Automated Control for Elastic Storage

Harold C. Lim   Shivnath Babu   Jeffrey S. Chase
Duke University
Durham, NC, USA
{harold, shivnath, chase}@cs.duke.edu

ABSTRACT

Elasticity—where systems acquire and release resources in response to dynamic workloads, while paying only for what they need—is a driving property of cloud computing. At the core of any elastic system is an automated controller. This paper addresses elastic control for multi-tier application services that allocate and release resources in discrete units, such as virtual server instances of predetermined sizes. It focuses on elastic control of the storage tier, in which adding or removing a storage node or "brick" requires rebalancing stored data across the nodes. The storage tier presents new challenges for elastic control: actuator delays (lag) due to rebalancing, interference with applications and sensor measurements, and the need to synchronize the multiple control elements, including rebalancing.

We have designed and implemented a new controller for elastic storage systems to address these challenges. Using a popular distributed storage system—the Hadoop Distributed File System (HDFS)—under dynamic Web 2.0 workloads, we show how the controller adapts to workload changes to maintain performance objectives efficiently in a pay-as-you-go cloud computing environment.

1. INTRODUCTION

Web-based services frequently experience rapid load surges and drops. Web 2.0 workloads, often driven by social networking, provide many recent examples of the well-known flash crowd phenomenon. One recent Facebook application that "went viral" saw an increase from 25,000 to 250,000 users in just three days, with up to 20,000 new users signing up per hour during peak times [1].

There is growing commercial interest and opportunity in automating the management of such applications and services. Automated surge protection and adaptive resource provisioning for dynamic service loads has been an active research topic for at least a decade. Today, the key elements for wide deployment are in place. Most importantly, a market for cloud computing software and services has emerged and is developing rapidly, offering powerful new platforms for *elastic* services that grow and shrink their service capacity dynamically as their request load changes.

Cloud computing services manage a shared "cloud" of servers as a unified hosting substrate for guest applications, using various technologies to virtualize servers and orchestrate their operation. A key property of this cloud hosting model is that the cloud substrate provider incurs the cost to own and operate the resources, and each customer pays only for the resources it demands over each interval of time. This model offers economies of scale for the cloud provider and a promise of lower net cost for the customer, especially when their request traffic shows peaks that are much higher than their average demand. Such advantageous demand profiles occur in a wide range of settings. In one academic computing setting, it was observed that computing resources had less than 20% average utilization [18], with demand spikes around project deadlines. This paper focuses on another driving example: multi-tier Web services, which often show common dynamic request demand profiles (e.g., [9]). Figure 1 depicts this target environment.

Mechanisms for elastic scaling are present in a wide range of applications. For example, many components of modern Web service software infrastructure can run in clusters at a range of scales; and can handle addition and removal of servers with negligible interruption of service. This paper deals with *policies* for elastic scaling based on automated control, building on the foundations of previous works [24, 34, 23, 22, 16] discussed in Section 7. We focus on challenges that are common for a general form of virtual cloud hosting, often called *infrastructure as a service*, in which the customer acquires virtual server instances from a cloud substrate provider, and selects or controls the software for each server instance. Amazon's Elastic Compute Cloud (EC2) is one popular example: the EC2 API allows customers to request, control, and release virtual server instances on demand, with pay-as-you-go pricing based on a per-hour charge for each instance. A recent study [2] reported that the number of Web-sites using Amazon EC2 grew 9% from July to August 2009, and has an annual growth rate of 181%.

We address new challenges associated with scaling the storage tier in a data-intensive cluster-based multi-tier service in this setting. We employ an integral control technique called *proportional thresholding* to modulate the number of discrete virtual server instances in a cluster. Many previous works modulate a continuous resource share allotted to a single instance [34, 23, 22]; cloud systems with per-instance pricing like EC2 do not expose this actuator.[1] We also address new challenges of *actuator lag* and *interference* stemming from the delay and cost of redistributing stored data on each change to the set of active instances in the storage tier.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

[1]Note to ICAC reviewers: our previous workshop paper [20] introduced proportional thresholding to control the application tier of a multi-tier service hosted in a cloud with per-instance pricing. This paper builds on that work by focusing on elastic control of the storage tier in that setting. The storage tier presents new challenges and is the missing element of an integrated cluster-based multi-tier solution.

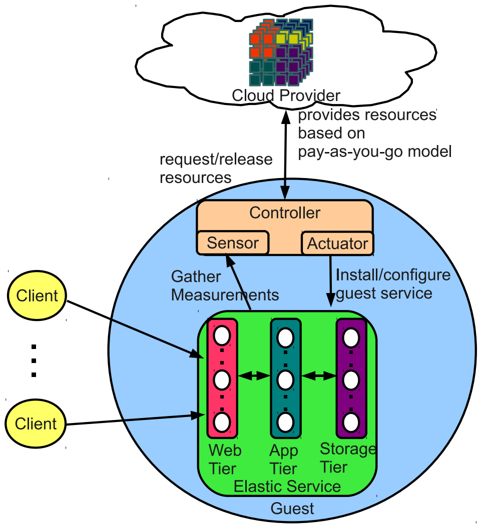


Figure 1: A multi-tier application service (guest) hosted on virtual server instances rented from an elastic cloud provider. An automated controller uses cloud APIs to acquire and release instances for the guest as needed to serve a dynamic workload.

While the discussion and experiments focus on cloud infrastructure services with per-instance pricing, our work is also applicable to multiplexing workloads in an enterprise data center. Some emerging cloud services offer packaged storage APIs as a service under the control of the cloud provider, instead of or in addition to raw virtual server instances for each customer to deploy a storage tier of their choice. In that case, our work applies to the problem faced by the cloud provider of controlling the elastic cloud storage tier shared by multiple customers.

We have implemented a prototype controller for an elastic storage system. We use the Cloudstone [27] generator for dynamic Web 2.0 workloads to show that the controller is effective and efficient in responding to workload changes.

2. SYSTEM OVERVIEW

Figure 1 gives an overview of the target environment: an elastic *guest* application hosted on server instances obtained on a pay-as-you-go basis from a cloud substrate provider. In this example, the guest is a three-tier Web service that serves request traffic from a dynamic set of clients.

Since Web users are sensitive to performance, the guest (service provider) is presumed to have a Service Level Objective (SLO) to characterize a target level of acceptable performance for the service. An SLO is a predicate based on one or more performance metrics, typically response time quantiles measured at the service edge. For any given service implementation, performance is some function of the workload and servers that it is deployed on; in this case, the resources granted by the cloud provider.

The purpose of controlled elasticity is to grow and shrink the active server instance set as needed to meet the SLO efficiently under the observed or predicted workload. Our work targets guest applications that can take advantage of this elasticity. When load grows, they can serve the load effectively by obtaining more server instances and adding them to the service. When load shrinks, they can use resources more efficiently and save money by releasing instances.

This paper focuses on elastic control of the storage tier, which presents challenges common to the other tiers, and additional challenges as well: state rebalancing, actuator lag, interference, and coordination of multiple interacting control elements. Storage scaling is increasingly important in part because recent Web 2.0 workloads have more user-created content, so the footprint of the stored data and the spread of accesses across the stored data both grow with the user community. Our experimental evaluation uses the Cloudstone [27] application service as a target guest. Cloudstone mimics a Web 2.0 events calendar application that allows users to browse, create, and join calender events.

2.1 Controller

We implement a *controller* process that runs on behalf of the guest and automates elasticity. The controller drives actuators (e.g., request/release instances) based on sensor measures (e.g., request volume, utilization, response time) from the guest and/or cloud provider. Our approach views the controller as combining multiple control elements, e.g., one to resize each tier and one for rebalancing in the storage tier, with additional rules to coordinate those elements. Ideally, the control policy is able to handle unanticipated changes in the workload (e.g., flash crowds), while assuring that the guest pays the minimum necessary to meet its SLO at the offered load.

For clouds with per-instance pricing, the controller runs outside of the cloud provider and is distinct from the guest application itself. This makes it possible to implement application-specific control policies that generalize across multiple cloud providers. (RightScale takes this approach.)

In general, these clouds present a problem of *discrete actuators*. As Figure 1 shows, the controller is limited to elasticity actuators exposed by the cloud provider's API. Cloud infrastructure providers such as Amazon EC2 allocate resources in discrete units as virtual server instances of predetermined sizes (e.g., small, medium, and large). Most previous work on provisioning elastic resources assume continuous actuators such as a fine-grained resource entitlement or share on each instance [24, 34]. We developed a *proportional thresholding* technique for stable integral control with coarse-grained discrete actuators.

Our approach to integrated elastic control assumes that each tier exports a control API that the controller may invoke to add a newly acquired storage server to the group (*join*) and remove an arbitrary server from the group (*leave*). These operations may configure the server instances, install software, and perform other tasks necessary to attach new server instances to the guest application, or detach them from the application. We also assume a mechanism to balance load across the servers within each tier, so that request capacity scales roughly with the number of active server instances.

2.2 Controlling Elastic Storage

The storage tier is a distributed service that runs on a group of server instances provisioned for storage and allocated from the cloud provider. It exports a storage API that is suitable for use by the middle tier to store and retrieve data objects. We make the following additional assumptions about the architecture and capabilities of the storage tier.

- It distributes stored data across its servers in a way that balances load effectively for reasonable access patterns, and redistributes (*rebalances*) data in response to *join* and *leave* events.
- It replicates data internally for robust availability; the replication is sufficient to avoid service interruptions across a sequence of *leave* events, even if a departing server is released back to the cloud before *leave* rebalancing is complete.
- The storage capacity and I/O capacity of the system scales roughly linearly with the size of the active server set. The tiers cooperate to route requests from the middle tier to a suitable storage server.

The design of robust, incrementally-scalable cluster storage services with similar goals has been an active research topic since the early 1990s. Many prototypes have been constructed including block stores [19, 25] and file systems [29, 12], key-value stores [11, 3], database systems [10], and other "brick-based" architectures. For our experiments, we chose the Hadoop Distributed File System (HDFS), which is based on the Google File System [12] design and is widely used in production systems.

As we have framed the problem, elastic control for a cloud infrastructure service presents a number of distinct new challenges.

Data Rebalancing: Elastic storage systems store and serve persistent data which imposes additional constraints on the controller. On adding a new node, a clustered Web server gives immediate performance improvements because the new node can quickly start serving client requests. In contrast, adding a new storage node does not give immediate performance improvements to an elastic storage system because the node does not have any persistent data to serve client requests. The new node must wait until data has been copied into it. Thus, rebalancing data across storage nodes is a necessary procedure, especially if the elastic storage system has to adapt and handle changes in client workloads.

Interference to Guest Service: Data rebalancing consumes resources that can otherwise be used to serve client requests. The amount of resources (bandwidth) to allocate to the rebalancing process affects its completion time as well as the degree of adverse impact on the guest application's performance during rebalancing. Note that overall improvement to system performance can be achieved only through data rebalancing. It may not be advisable to allocate a small bandwidth for rebalancing since it can take hours to complete, causing a prolonged period of performance problems due to suboptimal data placement. It may be better to allocate more bandwidth to complete rebalancing quickly while suffering a bigger intermediate performance hit. Finding the right balance automatically is nontrivial.

Actuator Delays: Regardless of the bandwidth allocated for rebalancing, there will always be a delay before performance improvements can be observed. The controller must account for this delay, or else it may respond too late or (worse) become unstable.

3. COMPONENTS OF THE CONTROLLER

Our automated control element for the elastic storage tier, which we call the *elasticity controller*, has three components:

- *Horizontal Scale Controller (HSC)*, responsible for growing and shrinking the number of storage nodes.
- *Data Rebalance Controller (DRC)*, responsible for controlling the data transfers to rebalance the storage tier after the number of storage nodes is grown or shrunk.
- *State machine*, responsible for coordinating the actions of the HSC and the DRC.

We present each of these components in turn and discuss how they address the challenges listed in Section 2.

3.1 Horizontal Scale Controller (HSC)

Actuator: The HSC uses cloud APIs to change the number of active server instances. Each storage node in the system runs on a separate virtual server instance.

Sensor: The HSC must base its elastic sizing choices on a feedback signal comprising of one or more performance metrics. There are a number of implementation choices of the performance metrics available to the HSC. A good choice of performance metric for the target environment satisfies the following properties: (i) the metric should be easy to measure accurately without intrusive code instrumentation because the HSC is external to the guest application, (ii) the metric should measure the tier-level performance, (iii) the metric should not have high variations, and (iv) the metric should strongly correlate to the measure of level of service (e.g, end-to-end response time) as specified in the client's service level objective (SLO).

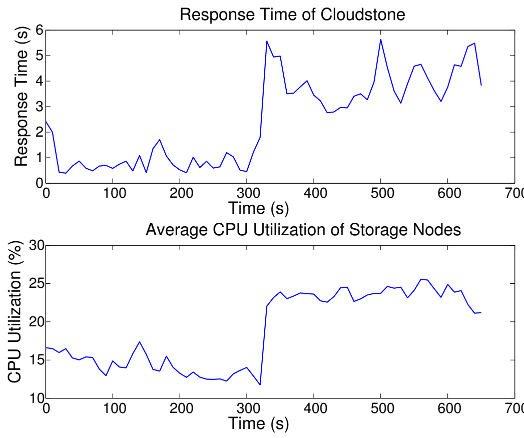


Figure 2: The response time and average CPU utilization of the storage nodes, under light and heavy workload. Correlation coefficient between response time and average CPU utilization is $.88$.

For our target application, we use CPU utilization on the storage nodes as our performance metric because it satisfies the aforementioned properties. Measuring CPU utilization does not involve intrusive code instrumentation and can be obtained from the operating system of the virtual machine. Moreover, tier-level metrics, such as CPU utilization, allow the controller to pinpoint the location of the performance bottleneck. Figure 2 shows that CPU utilization is a stable signal. When the performance bottleneck is in the storage tier, CPU utilization at the storage nodes is strongly correlated to end-to-end response time. It should be noted that for other target applications, one has to find a suitable performance metric that satisfies the previously mentioned properties and may differ from our choice of using CPU utilization.

Control Policy: We selected classical integral control as a starting point because it is self-correcting and provably stable in a wide range of scenarios, and has been used successfully in related systems [23, 24, 34]. Classical integral control assumes that the actuator is continuous and can be defined by the following equation.

$$u_{k+1} = u_k + K_i \times (y_{ref} - y_k) \tag{1}$$

Here, u_k and u_{k+1} are the current and new actuator values. K_i is the integral gain parameter [24]. y_k is the current sensor measurement. y_{ref} is the desired reference sensor measurement. Intuitively, integral control adjusts the actuator value from the previous time step proportionally to the deviation between the current and desired values of the sensor variable in the current time step. Since average CPU utilization is the sensor variable in HSC, y_{ref} is a reference average CPU utilization corresponding to a reference (target SLO) value of average response time. In our experiments, we chose a reference average response time of 3 seconds, which empirically gives a y_{ref} of 20% average CPU utilization.

Equation 1 assumes that the actuator u is a continuous variable. We show that directly applying this equation to discrete actuators can cause instability [20, 36]. Suppose u represents the number of virtual server instances allocated as storage nodes. For a change in the workload that causes y_k to increase, Equation 1 may set u_{k+1}

to 1.5 virtual server instances from $u_k = 1$. Since the HSC cannot request half a virtual server instance from the cloud provider, it allocates one full virtual server.

However, y_{k+1} may now drop far below y_{ref} because two virtual server instances are more than what is needed for the new workload. At the next time step, the controller may then decrease the number of virtual servers back to one, which raises back y_{k+2} above y_{ref}. This oscillatory behavior can continue indefinitely.

One solution is to transform y_{ref} into a target range specified a pair of high (y_h) and low (y_l) sensor measurements.

$$u_{k+1} = \begin{cases} u_k + K_i \times (y_h - y_k) & \text{if } y_h < y_k \\ u_k + K_i \times (y_l - y_k) & \text{if } y_l > y_k \\ u_k & \text{otherwise} \end{cases} \qquad (2)$$

The HSC will react only if $y_h < y_k$ (under-provisioned system) or $y_l > y_k$ (over-provisioned system). However, setting y_h and y_l statically can either lead to resource inefficiency (if the range is large) or to instability (if the range is small). The reason is the following: the impact of adding or removing a storage node to a system with N nodes is $\frac{1}{N}$. For example, adding an additional node to a one-node system divides the overall workload into two, thus reducing the average CPU utilization by 50%. However, going from 100 to 101 nodes will reduce average CPU utilization by less than 1%.

To address these problems, we developed *proportional thresholding* that combines Equation 2 with dynamic setting of the target range. We set $y_h = y_{ref}$, and vary y_l to vary the range. Since the performance impact of adding or removing a node becomes smaller as the size N of the system increases, the target range should have the following *Property I* to ensure efficient use of resources: $\lim_{N\to\infty} y_l = y_{ref} = y_h$.

Furthermore, to avoid oscillations as y_l is varied, the following *Property II* should hold: when the cluster size is reduced by one due to the sensor measurement falling below y_l, the new sensor measurement that results should not exceed y_h.

To capture these properties in our setting, we empirically model the relationship between the Cloudstone workload and CPU utilization (sensor values) per storage node.

$$CPU = f(workload); \text{thus, } workload = f^{-1}(CPU) \qquad (3)$$

The per-node workload *workload*$_h$ corresponding to the target $y_h = y_{ref}$ is: *workload*$_h$=$f^{-1}(y_h)$. Any per-node workload greater than *workload*$_h$ will result in a sensor measurement that exceeds y_h. Let *workload*$_l$ be the per-node workload at the point where HSC decides to reduce the current number of allocated storage nodes (*currVM*) by one. To ensure that Property II holds, we should have:

$$\begin{aligned} workload_l &= workload_h \times \frac{currVM - 1}{currVM} \\ y_l = f(workload_l) &= f\left(f^{-1}(y_h) \times \frac{currVM - 1}{currVM}\right) \end{aligned}$$

We parameterized the trigger condition by fitting a function to empirical measurements of the CPU utilization of HDFS datanodes at various load levels.

3.2 Data Rebalance Controller (DRC)

When the number of storage nodes grows or shrinks, the storage tier must rebalance the layout of data in the system to spread load and meet replication targets to guard against service interruption or data loss. The DRC uses a rebalancer utility that comes with HDFS to rebalance data across the storage nodes. Rebalancing is a cause of actuator delay and interference. For example, a new storage node added to the system cannot start serving client requests until some of the data to be served has been copied into it; and the performance of the storage tier as a whole is degraded while rebalancing is in progress.

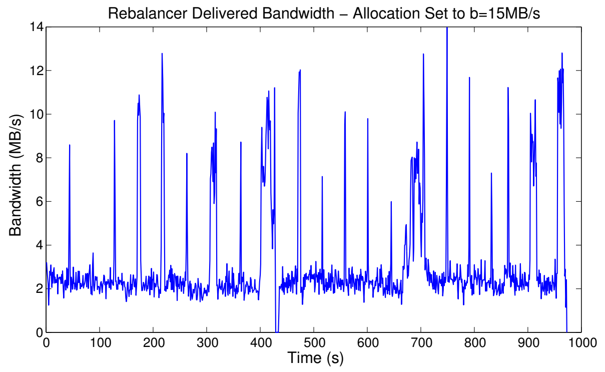


Figure 3: Delivered bandwidth of the HDFS rebalancer for b=15MB/s. Average bandwidth over entire run is 3.08MB/s.

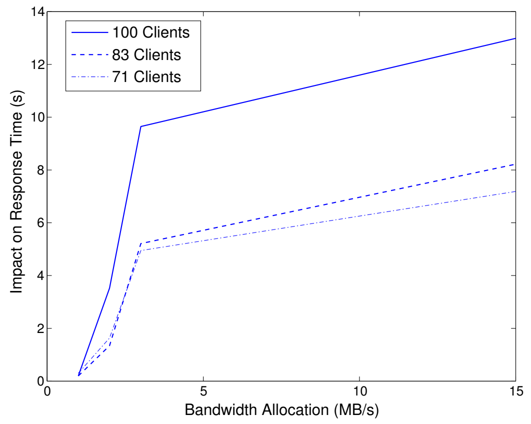


Figure 4: The impact on foreground client workload response time of rebalancer activity, as a function of bandwidth cap and client load level.

Actuator: The tuning knob of the HDFS rebalancer—i.e., the actuator of the DRC—is the *bandwidth* b allocated to the rebalancer. The bandwidth allocation is the maximum amount of outgoing and incoming bandwidth each storage node can devote to rebalancing. The DRC can select b to control the tradeoff between lag—i.e., the time to completion of the rebalancing process—and interference—i.e., performance impact on the foreground application—for each rebalancing action. Nominally, interference is proportional to b and lag is given by s/b where s is the amount of data to be copied.

We discovered empirically that the time to completion of rebalancing given by the current version of the HDFS rebalancer is insensitive to b settings above about 3 MB/s. The reason is that the rebalancer does not adequately pipeline data transfers, as illustrated in Figure 3. However, since HDFS and its tools are used in production deployments, and unreliable actuators are a fact of life in real computer systems, we decided to use the HDFS rebalancer "as is" for now and adapt to its behavior in the control policy.

Figure 4 shows the interference or *performance impact (Impact)* of rebalancing on Cloudstone response time, as a function of the bandwidth throttle (b) and the per-node load level (l). *Impact* is defined as the difference between the average response time with and without the rebalancer running. As expected, *Impact* increases as b and l increase. Running the rebalancer with b=1MB/s gives negligible impact on average response time.

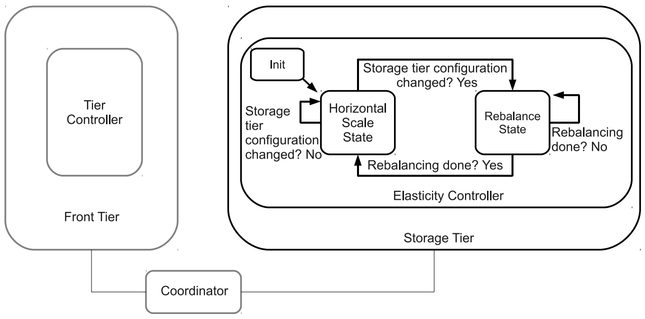


Figure 5: Block diagram of the control elements of a multi-tier application. Specifically, the diagram zooms in on the internal state machine of the elasticity controller of the storage tier and treats the other tier as a black box.

Sensor and Control Policy: From the data gathered through a planned set of experiments, we modeled the following using multivariate regression:

- The time to completion of rebalancing (*Time*) as a function of the bandwidth throttle (b) and size of data to be moved (s): *Time* $= f_t(b, s)$.
- The impact of rebalancing (*Impact*) as a function of the bandwidth throttle (b) and per-node workload (l): *Impact* $= f_i(b, l)$.

The goodness of fit is high ($R^2 \geq 0.995$) for both models. Values of s and l are used as sensor measurements by the DRC, and b is the tuning knob whose value has to be determined. The choice of b represents a tradeoff between *Time* and *Impact*. As previously stated, the controller must consider the lag (*Time*) to complete an adjustment and restore a stable service level, and the magnitude of the degradation in service performance (*Impact*) during the lag period.

To strike the right balance between actuator lag and interference, the DRC poses the choice of b as a cost-based optimization problem. Given a cost function *Cost* = $f_c($*Time,Impact*$)$ = $f_c(f_t(b, s), f_i(b, l))$, DRC chooses b to optimize *Cost* given the observed values of s and l. The cost function is a weighted sum: *Cost* = α*Time* + β*Impact*. The ratio of $\frac{\alpha}{\beta}$ can be specified by the guest based on the relative preference towards *Time* over *Impact*. Another alternative is to choose b such that *Time* is minimized subject to an upper bound on *Impact*.

3.3 State Machine

To preserve stability during adjustments, the HSC and DRC must coordinate to manage their mutual dependencies. The first dependency arises from the DRC's actuator lag. After a storage node has been added by the HSC, the benefit of the node starts to show only after rebalancing completes; which can take on the order of few tens of minutes for a few Gigabytes of data. The second dependency is due to noise introduced into the sensor measurements that a controller relies on, while the actions of the other controller are being applied. For example, the data copying and additional computations done during rebalancing impact the CPU utilization measurements seen by the HSC.

Ignoring these dependencies can lead to resource inefficient decisions, or much worse, unstable behavior due to oscillation. Consider a scenario where the HSC does not take the DRC's actuator lag into account. After adding a new storage node, the HSC may not see any changes in its sensor measurements, or the sensor measurements may show a decline in performance. This observation will cause the HSC to allocate more storage nodes unnecessarily to compensate for the lack of improvement in system performance. In turn, the completion time and impact of rebalancing could deteriorate further.

The elasticity controller uses the state machine shown in Figure 5 to coordinate the actions of the HSC and DRC. For illustrative purposes, Figure 5 also shows how the elasticity controller fits in an integrated cluster-based multi-tier control solution for a multi-tiered application. In this paper, we focus on the storage tier and treat the control elements for other tiers as a black box, since there has already been previous work on controlling other tiers (e.g., [20]). Section 6.3 provides further discussion on the problem of coordinating multiple per-tier control elements.

When the elasticity controller starts up, it goes from the *Init State* to the *Steady State*. In this state, only the HSC is active. It remains in this state until the HSC triggers an adjustment to the active server set size. When nodes are added or removed, the state machine transitions to the *Rebalance State*. The HSC is dormant in the *Rebalance State* to allow the previous change to stabilize and to ensure that it does not react to interference in its sensor measurements caused by data rebalancing.

The DRC, as described in Section 3.2, enters the *Rebalance State* after a change to the active server set size. It remains in this state until data rebalancing completes, after which the state machine returns to the *Steady State*. A form of rebalancing, called decommissioning, occurs on removal of a storage node to maintain configured replication degrees. HDFS stores n (a configurable parameter) replicas per file block, one of which may be on a node identified for removal. The replica of a block on a decommissioned node can be replaced by reading from any of the $n-1$ remaining copies. HDFS has an efficient internal replication mechanism that triggers when the replica count of any block goes below its threshold. Currently, the DRC does not regulate this process because HDFS does not expose external tuning knobs for it. In any case, we observed that this process has minimal impact on the foreground application.

4. IMPLEMENTATION

4.1 Cloud Provider

We use ORCA [14, 8] as our cloud infrastructure provider. ORCA is a resource control architecture and framework developed at Duke University. It provides a resource leasing mechanism which allows guests to lease a configurable amount of resources from a resource substrate. Our ORCA configuration allows multiple guests to share a common pool of physical servers using Xen virtualization technology [7]. In our current implementation, ORCA is set up running with an inventory of 30 physical machines. The resources leased to guests are encapsulated in a slice, which comes in the form of a set of virtual machines (VMs) instantiated from any of the physical machines in the inventory.

4.2 Cloudstone Guest Application

CloudStone: We modified and configured Cloudstone to run with GlassFish as the front-end application server tier, PostgreSQL as the database tier for structured data, and HDFS as a distributed storage tier for unstructured data such as PDF and image files. This required adding an HDFS class abstraction to Cloudstone to enable it to use HDFS storage APIs. We also added new parameter types to Cloudstone's configuration file so that users can easily configure and switch between different file systems without having to recompile the source code. In all, this involved adding 200 lines of code to the Cloudstone source code. The experiments use a block size in the storage tier of 800KB, which is the maximum size of binary files generated by Cloudstone. The HDFS replica count is set to three following best practices from production deployments.

HDFS: HDFS follows a master/slave architecture. The master, called HDFS namenode, contains the metadata of the files. The slave, called HDFS datanode, contains the actual data (files). There is usually one HDFS namenode and a variable number of HDFS datanodes. With its current implementation, HDFS does not ensure that storage nodes are request balanced, since its internal policy is based on disk usage capacity. However, Cloudstone's workload generator is designed such that structured and unstructured data are accessed in a uniform distribution, which naturally balances requests across all HDFS datanodes.

Finally, we modified HDFS so that the elasticity controller can select the bandwidth throttle b for the rebalancer tool. We created an RPC in the HDFS namenode that notifies all HDFS datanodes of the bandwidth limit.

4.3 Elasticity Controller

The controller is written in Java and contains 1500 lines of code. ORCA allows guests to use the resource leasing mechanisms through a controller plug-in module written to various toolkit APIs [35]. ORCA controllers implement an IController upcall interface that has a tick method, which is called at regular intervals. The control policy is clocked by this tick handler.

Each new lease request is attached with a handler that installs, configures, and launches the guest software on the leased server instances. Our handler installs and configures the HDFS datanode software package when a new virtual machine is instantiated and also performs the necessary shutdown sequence, such as shutting down the HDFS datanode, when the controller decides to decommission a virtual machine instance. The control system includes two other important components, described next.

Instrumentation: To get the sensor measurements mentioned in Section 3, we modified the HDFS datanode to gather system-level metrics such as CPU utilization. We included the Hyperic SIGAR library to the HDFS datanode. At periodic intervals, the HDFS datanode uses SIGAR to gather the system-level metrics and piggybacks this information on the regular heartbeat messages of the HDFS datanode to the HDFS namenode. We also modified the HDFS namenode and implemented a remote procedure call (RPC) that allows the controller to get the sensor measurements of all HDFS datanodes in a single data structure. With this implementation, the controller only needs to contact the HDFS namenode to get the sensor measurements of all nodes.

The controller has a separate thread that periodically, currently set to 10 seconds, makes an RPC to the HDFS namenode. The controller then processes the response of the RPC. It first calculates the average CPU utilization of all HDFS datanodes. Then, it applies an exponential moving average filter of six time periods to the average CPU utilization.

Subcontroller Modules: The controller has two subcontroller modules, corresponding to HSC and DRC, mentioned in Section 3. Each of these modules are running on a separate thread. As mentioned in Section 3, the coordination between these two subcontroller modules is guided by a finite state machine. The controller plug-in module installs event handlers that trigger notifications from the leasing core at specific points of a lease's life cycle. These handlers are an ORCA API designed for dynamic controllers. We use the onBeforeExtendTicket and onLeaseComplete handlers that are triggered just before a lease expires and after a new lease reservation is complete (e.g., a new datanode is instantiated). Since the feedback controller and the leasing mechanism run asynchronously on separate threads, these two handlers are synchronized through a common state variable representing the state of the controller's state machine. This state variable activates and deactivates the subcontroller modules.

5. EVALUATION

5.1 Experimental Testbed

Our experimental testbed consists of physical servers running on the same local network. To focus on the storage tier, the front-end application tier and database tier of Cloudstone are statically over-provisioned: the database server (PostgreSQL) runs on a powerful server with 8GB of memory and 3.16 GHz dual-core CPU, while the forward tier (GlassFish) runs in a fixed six-node cluster, where each node has 1GB of memory and a 2.8GHz CPU. The storage tier nodes are dynamically allocated virtual machine instances, with fixed settings of resource configuration, based on the control policy discussed in Section 3. We used weak virtual machine instances for the storage nodes to trigger responses from the controller at a smaller scale of workloads. The virtual machine instances have 30GB disk space, 512MB of memory and a 2.8GHz CPU, with a CPU cap set at 20%. Before each experiment, the HDFS tier is preloaded with at least 36GB worth of data (i.e., images and binary files used by Cloudstone). The Cloudstone workload generator is running on separate well-provisioned machines, and is never bottlenecked.

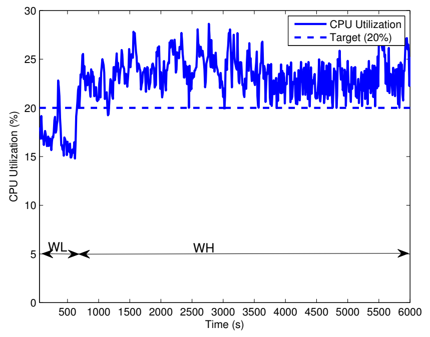
5.2 Controller Effectiveness

Internet workloads are known to show predictable long-term variations and highly unpredictable short-term fluctuations [31]. Long-term variations, usually predictable through models of past observations, can be handled by pre-provisioning resources in anticipation of the changes in workload behavior. However as mentioned in Section 1, unpredictable changes to the workload, such as flash crowds, happen often in practice. These changes cannot be anticipated by simply observing past observations, and are hard to deal with. We are interested in evaluating the effectiveness and adaptability of our controller under such unanticipated workload behavior. We first use Cloudstone to subject HDFS to dynamic workloads that represent sudden increases in load. We want to evaluate whether our controller is able to dynamically provision more resources to handle the client workload and to fix the SLO violations that arise.

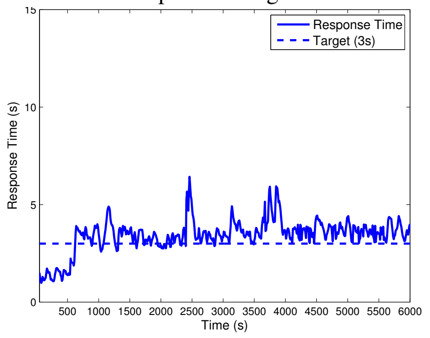
In this experiment, we programmed the load signal to first generate a small workload (load factor of 1.0). At around 600 seconds, the load factor is increased by 35%. We set the target response time to be three seconds, which corresponds to 20% CPU utilization of the storage nodes. The storage system is running with 10 HDFS datanodes.

Figure 6 shows the performance of Cloudstone with static resource provisioning and our control policy. With static provisioning, the system becomes under-provisioned for the increase in workload (see Figures 6(a) and (b)). Since resources are statically provisioned, the performance will continue to have SLO violations indefinitely until the workload goes back down. With our control policy, the controller detects the impact on performance of the increase in workload and decides to increase the storage cluster size by one (see Figures 6(c) and (d)). Figure 6(c) also shows the period, marked with an arrow and labeled as "Rebalance", when the rebalancing process is taking place. During this rebalancing process, the rebalance controller uses our rebalance model (with cost function parameters α=1 and β=2) to choose 2.2MB/s as the bandwidth to allocate to the rebalancer. At around the 2450^{th} time step, the response time and CPU utilization has gone down to below the target value due to the successful addition and integration of a new HDFS datanode.

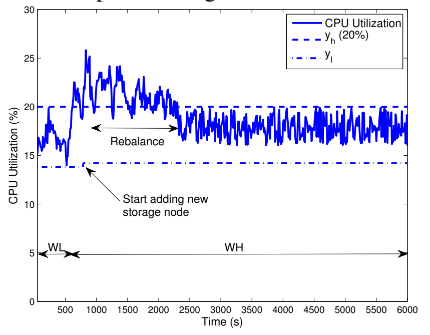
With our controller, although there is an impact of an average of four seconds in response time due to the rebalancing process, the system is able to adapt to the new workload and fix the SLO viola-



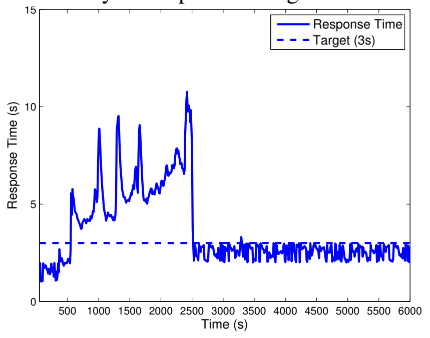
(a) Average CPU utilization of the HDFS datanodes with static provisioning



(b) Response time of the Cloudstone application with static provisioning

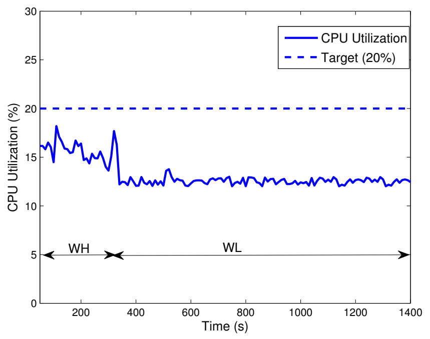


(c) Average CPU utilization of the HDFS datanodes with dynamic provisioning

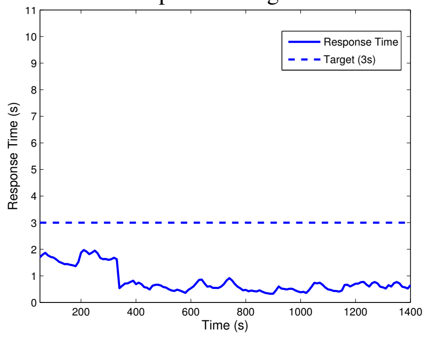


(d) Response time of the Cloudstone application with dynamic provisioning

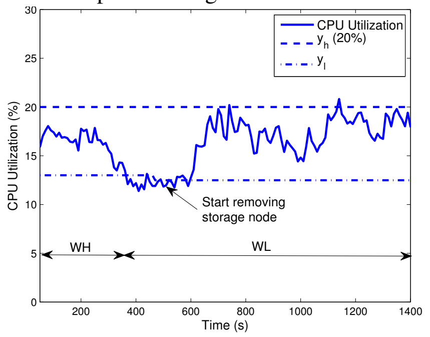
Figure 6: The performance of Cloudstone with static allocation (a,b) and our control policy (c,d), under an increase in workload volume. The time periods with low and high volume of workload are labeled as "WL" and "WH", respectively.



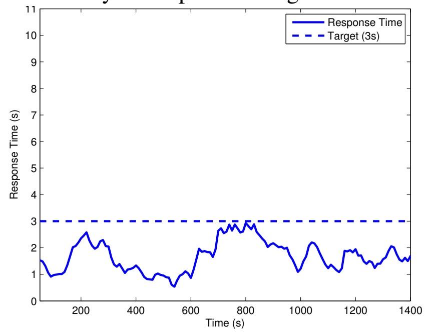
(a) Average CPU utilization of the HDFS datanodes with static provisioning



(b) Response time of the Cloudstone application with static provisioning



(c) Average CPU utilization of the HDFS datanodes with dynamic provisioning



(d) Response time of the Cloudstone application with dynamic provisioning

Figure 7: The performance of Cloudstone with static provisioning (a,b) and our control policy (c,d), under a decrease in workload volume. The time periods with low and high volume of workload are labeled as "WL" and "WH", respectively.

tions (Figure 6(d)). As discussed in Section 3.2, the noisy behavior of the response time is unavoidable due to the current implementation of the HDFS rebalancer. Furthermore, Figure 6(d) shows that the cost of data rebalancing will have to be paid under unpredictable workloads. In this experiment, we picked a rebalance policy that has a balanced tradeoff between the data rebalancer's completion time and impact. In section 5.4, we discuss how the α and β parameters of the cost function can be tuned by the guest to get the desired ratio of impact to completion time. The tuning will be done based on how much rebalance cost a guest is willing to absorb to fix SLO violations speedily.

5.3 Resource Efficiency

In the next experiment, we subject Cloudstone to a sudden decrease in workload from an initial workload that the system can handle without any SLO violations. We are interested to see whether our controllable elastic storage system meets our resource efficiency goal mentioned in Section 2. Figure 7 shows the behavior of our controller.

Similar to the previous experiment, we compare the performance of static thresholding with our control policy. In this experiment, the workload is decreased at the 370^{th} time step. Figures 7(a) and (b) show the CPU utilization and response time graph of the system with static provisioning. Since the resource configuration does not change in static provisioning, we also see a decrease in response time that is two seconds below the threshold for SLO violation. However under a prolonged decrease in workload, static provisioning will incur unnecessary resource costs because it is over provisioned for the current workload.

With our control policy, at the 420^{th} time step, our controller is able to detect and determine that the system is over-provisioned. The controller then releases the excess HDFS datanode and returns the resources to the cloud provider (see Figure 7(c)). As shown in Figure 7(d), even with a decrease in the size of the storage cluster, there still are no SLO violations.

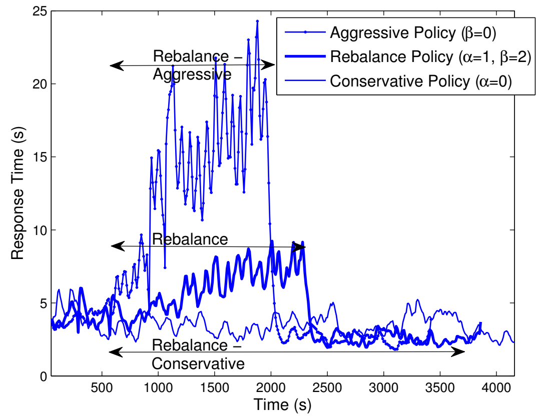


Figure 8: The response time of Cloudstone under different rebalance policies: Aggressive policy, our controller's rebalance policy, and conservative policy.

5.4 Comparison of Rebalance Policies

For illustrative purposes, we compared our rebalance policy with two other policies: aggressive and conservative. An aggressive policy allocates as much bandwidth as possible to the data rebalancer. On the other hand, a conservative policy allocates minimal bandwidth so that there is minimal impact on the response time of Cloudstone during rebalancing.

In this experiment, we drive a heavy workload to Cloudstone and then let the controller allocate a new storage node and start the rebalancing process. Figure 8 shows the response time of Cloudstone when the rebalancer is triggered. For each policy in the figure, the period reflecting the running time of the rebalancer is marked with an arrow and labeled as "Rebalance". An aggressive rebalance policy gives the shortest time to completion but also severely impacts the response time of Cloudstone. However, compared to our policy, it only gives around five minutes of improvement to the rebalancer's time to completion. Moreover, our policy gives 15 seconds less impact on the response time of Cloudstone than an aggressive rebalance policy. A conservative policy gives minimal impact on response time but takes more than twice as long to complete; which is also not good because it prolongs the period of SLO violations. Our controller's rebalance policy shows a balance between time to completion and the impact on Cloudstone. It should be noted that a conservative and aggressive rebalance policy can be attained by setting the α and β parameters respectively to zero.

6. DISCUSSION AND FUTURE WORK

6.1 Other Cloud Computing Models

In this paper, we focused on the infrastructure-as-a-service model of cloud computing (like Amazon EC2) where each guest runs a private storage service on virtual server instances leased from the cloud provider. Software-as-a-service is another popular model on the cloud (like Amazon S3) where the cloud provider offers a software service using a pay-as-you-go pricing model; rather than leasing out virtual resources. In this case, the control problem of storage elasticity does not arise for the guests because they don't control the storage infrastructure. However, the control problem has simply moved to the cloud provider. Our overall approach can be used by the cloud provider, but an additional challenge arises. The storage service will now be used and shared among multiple guests, each with its own performance objectives and data. The controller needs to make sure that there is performance isolation and differentiation across guests. It is worth noting a recent paper that discusses the problem of massive resource inefficiencies in emerging parallel systems [4]. Someone has to pay for this inefficiency—the cloud provider will have to pay in the software-as-a-service model unless they leverage elastic storage.

6.2 Data Rebalancing

Automated data rebalancing is a critical ingredient of a controllable elastic storage system. The kinds of rebalancing needed is specific to the storage system and application needs. In our target guest, for example, data rebalancing entails moving files to new storage nodes, replicating files for availability, and ensuring that the load is balanced across all nodes. On the other hand, collocating multiple data items is a crucial need during data rebalancing in a database system, e.g., collocating indexes along with the indexed data records. An ideal rebalancer should hide system-specific details, and expose appropriate tuning knobs so that the elasticity controller can invoke the rebalancer to meet service-specific needs on completion time and performance impact. The best way to implement a rebalancer is a nontrivial question.

HDFS Rebalancer: The current implementation of the HDFS rebalancer limits the performance of the elasticity controller. The bursty data transfer rates observed in Figure 3 and the noisy response time values in Figure 6(d) are unfortunate side-effects of this implementation. The implementation also causes high computational overhead. For example, the HDFS rebalancer creates separate socket connections between HDFS datanodes for each scheduled block transfer. Small block sizes can cause many open socket connections between datanodes. It should be noted that the issues with the HDFS rebalancer are tangential to the main point of this paper, which is addressing the challenges of automated control of an elastic storage tier.

6.3 Dealing with Multiple Actuators

One issue with having multiple actuators is that there will be sensor data interference and dependency. For example, we have shown that the data rebalancing process has an impact on sensor measurements. Thus, there is a need for coordination and synchronization among multiple actuators. In this paper, we used a *hierarchical* coordination scheme to coordinate between two actuators: cluster resizing and data rebalancing. This policy treats the two actuators as mutually exclusive, and data rebalancing always gets triggered after cluster resizing. One limitation of this policy is that if the workload changes while in the *Rebalance State*, the elasticity controller waits until the rebalancing process finishes before making another decision regarding the size of the cluster. In this situation, the controller can potentially be less responsive, i.e., longer time to adapt to workload changes.

As future work, we are looking into alternative policies for coordination between the two actuators. One possible approach is to run both actuators concurrently. We could develop techniques to filter out the impact of the rebalancing process on the sensor measurements. While the data rebalancer is running, the horizontal scale controller can then use the filtered measurements to determine whether further changes to the cluster configuration are necessary. These enhancements may make the controller more agile, which may be useful in rapid dynamic change of workloads, but we must balance stability and agility. Our current solution is simple and provably stable in that the controllers can never work at cross purposes.

Multi-tier Applications: We focused on controlling the storage tier of a multi-tier application, which is active only when the controller has determined that the bottleneck is in the storage tier. We can consider controlling a multi-tier application as dealing with multiple actuators. In this case, each tier can have an elastic number of resources (e.g., virtual server machines).

We are interested in finding the minimum amount of coordination among multiple actuators that still results in an effective and efficient control system. Treating each tier as an independent actuator with its own control policy can cause shifting of the performance bottleneck between tiers. Our proposed solution involves using an interlock to coordinate between tiers. A tier can only release resources when the interlock is not being held by the other tiers. The interlock is acquired by a tier when it detects that its resources are being overutilized. In our previous work [20], we have designed a controller for the front-end tier of Web applications. However we leave as a future work, the evaluation of the coordination policy between the front-end tier controller and our storage controller.

6.4 Adapting to Expected Load Changes

Currently, this paper only addresses the case of unpredictable workloads, in which the cost of rebalancing has to be paid by guests in order to fix SLO violations. As mentioned in Section 5.2, for the other type of workloads that exhibit reasonable load changes, the HSC controller can perform pre-provisioning of resources. With a predictable workload signal, we can use our models of time to completion of rebalancing (*Time*) and the impact of rebalancing (*Impact*) (refer to Section 3) to find when to trigger the actuators so that no SLO violations happen. The control policy then turns into a constrained optimization problem that minimizes the chosen bandwidth allocated to the rebalancer, while ensuring *Time* is earlier than the projected time of SLO violation and the sum of the projected growth in workload and *Impact* is smaller than the SLO threshold.

7. RELATED WORK

To our knowledge, we are the first to address the problem of automated control for elastic storage in the context of cloud computing. Specifically, no other work has focused on the combination of issues regarding discrete actuators, interference of the data rebalancing process on guest services, and actuator delays when designing a controller for elastic storage systems. SCADS [6] is a closely related work that deals with the problem of dynamically scaling a storage system. Its design uses machine learning to determine and predict resource requirements. Our controller employs a reactive policy. Moreover, we automate the data rebalancing process which is necessary for scaling the storage system.

Control of Computing Systems: There has been work on feedback control of computing systems [24, 34, 23, 22]. These works assume the availability of continuous actuators. Moreover, these works address the issue of control for non-clustered systems. For example, Wang et al. [34] dynamically adjust the CPU capacity of a guest virtual machine. This paper extends their work by designing a controller for clustered services. Specifically, our work uses the cluster mechanism of incrementally-scalable systems to dynamically provision cluster nodes.

In terms of automated control of computing systems in the context of cloud computing, Padala et al. [23] have also considered a decoupled architecture (between the guest and cloud provider) in the design of their control system. However, our work differs in that our control system also takes into account actuator constraints, which are inherent in all commercial cloud providers. For example, rather than adjusting CPU allocation, which commercial providers do not provide, our work regulates the number of instantiated virtual machines.

There has also been work addressing the problem of control of Web applications [30, 31, 16]. Urgaonkar et al. [30] use queuing theory to analytically model a multi-tier Internet application, and use this model to determine how many servers are needed in each tier. One difference with our work is that they use a centralized control architecture, which may not be feasible in the cloud context when the provider and guests are separate business entities. Yaksha [16] does not perform resource provisioning, rather it performs admission control to improve a Web application's performance.

Data Rebalancing: Previous works have addressed the data rebalancing problem in a storage system [21, 5, 26]. In these works, rebalancing data is performed in a way that it does not cause any violations to the foreground application's SLOs. Aqueduct [21] uses a feedback controller that throttles its bandwidth usage to ensure that the quality of service of the foreground application is not affected while data transfers (e.g., backups) are performed in the storage system; and only unused bandwidth is used. If there is only limited unused bandwidth, then this approach can take a long time to complete; which may not be acceptable in the context of controllable elastic storage systems. For our controller, rebalancing data is not treated as an optional procedure, but as a required procedure to fix SLO violations.

Actuator Delays: Soundararajan et al. [28] address the issue of actuator delays for a different control problem. They present a control policy for database replication on a static-sized cluster. Their controller waits for the replication process to complete before making a new decision. Our work addresses the problem of automated control of elastic storage systems while accounting for the delays brought by data rebalancing. Aside from waiting for the data rebalancing process to complete, our controller also finds the right balance between the time to completion and impact of data rebalancing. The use of proportional thresholding further distinguishes our work from [28].

Performance Differentiation for Storage Systems: There has been a long line of work that uses scheduling policies and admission control schemes to ensure performance guarantees and differentiation in storage systems [32, 13, 33]. For example, Jin et al. [15] present a share-based scheduling algorithm that enforces desired shares of resources for a storage service. Triage [17] uses a feedback controller to perform admission control by throttling client request rates to the storage system. These control schemes complement our work because we deal with allocating the right amount of resources to handle client workloads, while the aforementioned control schemes ensure that there is performance isolation between different sets of clients.

8. CONCLUSION

In this paper, we presented an automated controller for elastic storage systems in the context of cloud computing. The design addresses several challenges that exist due to the nature of the storage system and the cloud infrastructure. To address the issue of discrete actuators, our controller uses proportional thresholding to determine the size of the storage cluster. Moreover, the controller must treat data rebalancing as a first-class actuator. The controller uses a cost-optimization-based approach to determine the amount of bandwidth to use for rebalancing data as the cluster size grows or shrinks. A cost function is used to find the best tradeoff between the impact on the guest service and the time to completion of the rebalancing process. Finally, the controller uses a state machine to coordinate between multiple actuators and to be robust to actuator delays.

We evaluated our controller using a Web 2.0 benchmark application running on an experimental testbed that consists of a variable number of Xen virtual machines instantiated from an inventory of physical servers. The experimental results confirmed that our controller is able to dynamically provision coarse-grained resources (i.e., virtual machines) under unanticipated changes in the client workload. Our rebalance policy balances the performance impact and time to fix SLO violations. Furthermore, our controller maintains client SLOs while being very resource efficient.

9. REFERENCES

[1] Animoto's Facebook scale-up. `http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up`.

[2] State of the Cloud - August 2009. `http://www.jackofallclouds.com/2009/08/state-of-the-cloud-august-2009/`.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. of SOSP*, 2009.

[4] E. Anderson and J. Tucek. Efficiency Matters! In *Proc of HotStorage*, 2009.

[5] E. J. Anderson, J. Hall, J. D. Hartline, M. Hobbs, A. R. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An experimental study of data migration algorithms. In *Proc. of WAE*, 2001.

[6] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. Scads: Scale-independent storage for social computing applications. In *Proc. of CIDR*, 2009.

[7] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of SOSP*, 2003.

[8] J. Chase, L. Grit, D. Irwin, V. Marupadi, P. Shivam, and A. Yumerefendi. Beyond virtual data centers: Toward an open resource control architecture. In *Proc. of VCI*, 2007.

[9] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centres. In *Proc. of SOSP*, 2001.

[10] S. Das, D. Agrawal, and A. E. Abbadi. Elastras: An elastic transactional data store in the cloud. In *Proc. of HotCloud*, 2009.

[11] 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 *Proc. of SOSP*, 2007.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of SOSP*, 2003.

[13] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. In *Proc. of FAST*, 2009.

[14] D. E. Irwin, J. S. Chase, L. E. Grit, A. R. Yumerefendi, D. Becker, and K. Yocum. Sharing networked resources with brokered leases. In *Proc. of USENIX*, 2006.

[15] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proc. of SIGMETRICS*, 2004.

[16] A. Kamra, V. Misra, and E. M. Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *Proc. of IWQoS*, 2004.

[17] M. Karlsson, C. T. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage*, 2005.

[18] A. Kimball, S. Michels-Slettvet, and C. Bisciglia. Cluster computing for web-scale data processing. In *Proc. of SIGCSE*, 2008.

[19] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. of ASPLOS*, 1996.

[20] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh. Automated control in cloud computing: Challenges and opportunities. In *Proc. of ACDC*, 2009.

[21] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *Proc. of FAST*, 2002.

[22] 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 *Proc. of EuroSys*, 2009.

[23] 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 *Proc. of EuroSys*, 2007.

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

[25] Y. Saito, S. Frølund, A. C. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proc. of ASPLOS*, 2004.

[26] B. Seo and R. Zimmermann. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Transactions on Storage*, 2005.

[27] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, 2008.

[28] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *Proc. of EuroSys*, 2006.

[29] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. of SOSP*, 1997.

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

[31] B. Urgaonkar, P. J. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Proc. of ICAC*, 2005.

[32] S. Uttamchandani, L. Yin, G. A. Alvarez, J. Palmer, and G. A. Agha. Chameleon: A self-evolving, fully-adaptive resource arbitrator for storage systems. In *Proc. of USENIX*, 2005.

[33] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proc. of FAST*, 2007.

[34] Z. Wang, X. Zhu, and S. Singhal. Utilization vs. slo-based control for dynamic sizing of resource partitions. In *Proc. of DSOM*, 2005.

[35] A. Yumerefendi, P. Shivam, D. Irwin, P. Gunda, L. Grit, A. Demberel, J. Chase, and S. Babu. Towards an autonomic computing testbed. In *Proc. of HotAC*, 2007.

[36] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. What does control theory bring to systems research? *SIGOPS Operating Systems Review*, 2009.