CloudFlex: Seamless Scaling of Enterprise Applications into the Cloud

Yousuk Seung
ysseung@cs.utexas.edu

Terry Lam
vtlam@cs.ucsd.edu

Li Erran Li
erranlli@research.bell-labs.com

Thomas Woo
woo@research.bell-labs.com

***Abstract*—This paper proposes and studies a system, called CloudFlex, which transparently taps cloud resources to serve application requests that exceed capacity of internal infrastructure. CloudFlex operates as a feedback control system with two key interacting components: *load balancer* and *controller*. We focus on operational optimality and stability of the system, highlight the tradeoffs between cost and responsiveness, and address important design considerations such as choke point detection that are critical in avoiding pathological system operations. For evaluation, we develop a prototype of CloudFlex on our testbed comprising servers of our enterprise data center and Amazon EC2 instances.**

I. Introduction

Cloud computing paradigm offers a novel approach for utility computing with unprecedented resource f exibility, agility, and scalability [5]. A number of recent market research reports (e.g., Garner [6]) have predicted that cloud computing is poised for signif cant enterprise adoption within the next two to f ve years. In this paper, we address an important barrier to enterprise cloud computing adoption: how to transition seamlessly from a pure enterprise computing resource model to one that uses both enterprise internal and cloud-based resources.

We design a system called *CloudFlex* to provide this mechanism. At its core, CloudFlex leverages a feedback control system that monitors system performance, and distributes load onto a heterogeneous set of resources.

We present a study toward the design of a system that allows enterprise applications to be seamlessly extended into a cloud. In particular, the paper makes the following contributions:

- We are the f rst to study load balancing and auto-scaling in a hybrid mixture of internal and cloud resources.
- We propose and study novel scaling algorithm designs in a control theoretic framework with strong emphasis on stability and optimality, and balancing tradeoff between cost and responsiveness.
- We implement our design using state-of-the-art open source components, and evaluate performance on a real-world cloud provider (Amazon EC2).
- We address important issues for a complete system design and real-world deployment that include considerations for security, system choke points, etc.

II. CloudFlex Design

In this section we describe the main modules of the CloudFlex architecture.

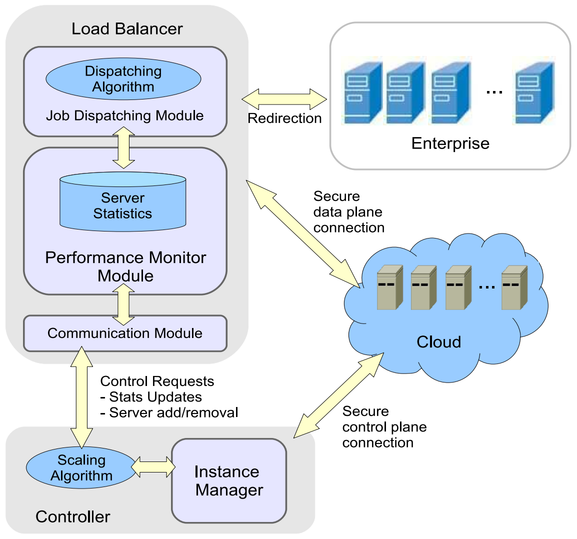


Fig. 1. CloudFlex system architecture

A key design principle of CloudFlex is the separation of the control and data planes. Figure 1 shows the building blocks of a CloudFlex system with two fundamental components. The *controller* provides auto-scaling intelligence by making decisions on resource acquisition/release in the control plane. The *load balancer* monitors performance and dispatches jobs in the data plane.

Load balancer: Our design of the load balancer leverages the multi-tiered architecture commonly deployed in enterprise applications. In such an architecture, interactions occur mostly between tiers and individual server instances are typically hidden behind a reverse-proxy with a single IP address. In particular, the load balancer includes three major modules. A *performance monitoring module* collects server performance data. A *communication module* periodically polls on the server statistics and status updates from the controller and also allows the controller to register callback functions upon certain conditions. Finally, a *job dispatching module* decides which server a request should be dispatched to, based on the information fed by the communication module.

Controller: Inside the controller, a *scaling algorithm* decides on resource scaling given current demand and performance statistics. The details of the scaling algorithm are discussed in Section IV. The controller then manages VM instances through an *instance manager*. Note that only the instance manager is aware of cloud provider specifi API and we can easily extend this to multiple cloud providers by having different “adapters” that speak the specifi cloud API of a cloud provider.

III. Design challenges

In this section, we address several important technical challenges in the design of CloudFlex.

Choke points: A *choke point* refers to a dynamic performance bottleneck that can shift from one place to another as the system scales out. For example, consider a three-tiered application architecture. Tier-2 servers run business logic and call on tier-3 database servers. Responses from tier-2 servers are collected and presented by tier-1 servers. Assuming the current bottleneck is the business logic computation, by scaling out tier-2 servers to the cloud, the overall system performance may improve until tier-3 database servers fail to keep up with the ever expanding set of application servers. At this point, further scalability gain can be achieved only with scaling out tier-3. Therefore, an important part of any auto-scaling design is the ability to detect such scaling choke points. We discuss our choke point detection technique in Section IV-E.

Cloud resource responsiveness: There is always a nonnegligible delay between the request and the acquisition of a cloud resource. This can be viewed as the “boot time” for cloud resources, and it implicitly serves as a cap for the burst of requests that can be handled. In our algorithm, due to our cost model (Amazon charges on an hourly basis), we do not always release servers immediately, and thus can reuse a server (if possible) before it terminates.

Load balancing: The micro-operations of a load balancer can be tricky to implement and get right. We highlight two examples here. First, naively selecting the “best” server would direct burst of requests to the “best” server, potentially causing serious thrashing. To avoid this problem we distribute load equally among internal nodes and external servers (but not among all), assuming that statistically requests are equally heavy and internal/external servers are equally capable. Second, in deciding to scale back and release a server, it is important to ensure that all existing connections would properly terminate before the VM is actually released. We call this the *bleeding* process and the CloudFlex assures that all terminating nodes have enough time to *bleed* before actually terminate.

IV. CloudFlex Algorithm

In this section we describe our system setup and the details of our CloudFlex algorithm.

A. System setup

The CloudFlex three-tiered architecture is shown in Figure 2. In such a system, typically, tier-1 and tier-2 can be scaled horizontally by adding more servers, while tier-3 scales vertically by adding resources to existing servers.

Our focus is on horizontal scaling algorithms for tier-1 and tier-2 servers. We do not attempt to scale tier-3 due to two reasons. First, tier-3 is tightly coupled with the application and hold sensitive confidentia information which makes enterprises reluctant to migrate tier-3 servers to the cloud. Second, limited bandwidth to the cloud and very large data size also hinders tier-3 migration to the cloud.

B. Application performance metric

Our performance metric is system response time, being define as the interval at the load balancer between accepting a client request and receiving its corresponding server response. Note that it is better than other indirect metrics (e.g. CPU load) since our requests are HTTP-based and hence tight system response time immediately guarantees application responsiveness.

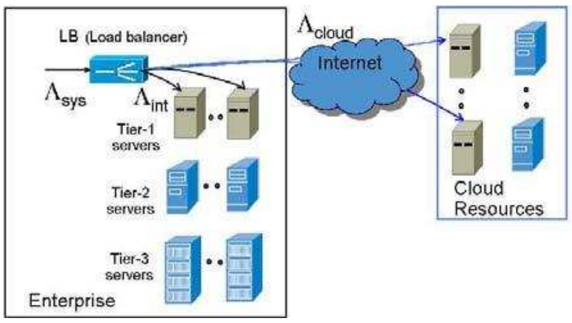


Fig. 2. A standard infrastructure for web service applications with three tiers of servers.

C. System state modeling

We model each VM’s system dynamics in terms of the relationship between its capacity, load and the resulting response time. Our system is based on the estimation per time window t. Note that the duration of each time window can be adjusted dynamically.

Response time vs load at each server: Let $\mathtt{qRespTime_i(pct,t)}$ be the $\mathtt{pct}$%-tile response time of completed application requests measured between the load balancer and application server i during the time interval indexed by t. Let $\lambda_\mathtt{i}(\mathtt{t})$ be the request arrival rate of server i at time t. Denote the maximum resource capacity of server i at the time interval t as $\mathtt{cap_i(t)}$. Note that $\lambda_\mathtt{i}(\mathtt{t})$ and $\mathtt{cap_i(t)}$ correspond to the birth (arrival) rate and death (service) rate in a queuing model at server i respectively.

We use a regression model (a function f) to fi the relationship between the response time and the arrival rate as follows.

$$\mathtt{qRespTime_i(pct,t)} = \mathtt{f}(\mathtt{cap_i(t)}, \lambda_\mathtt{i}(\mathtt{t})) \quad (1)$$

In our regression model, we fi a linear model between $\mathtt{cap_i(t)}$, $\lambda_\mathtt{i}(\mathtt{t})$ and $\frac{1}{\mathtt{qRespTime_i(pct,t)}}$. We use the simple least mean square estimation to estimate parameters $\mathtt{a}$ and $\mathtt{b}$ below.

$$\frac{1}{\mathtt{qRespTime_i(pct,t)}} = \mathtt{a} \cdot \lambda_\mathtt{i}(\mathtt{t}) + \mathtt{b} \cdot \mathtt{cap(t)} \quad (2)$$

Our empirical formula shows that $\frac{1}{\mathtt{qRespTime_i(pct,t)}}$ is a quadratic function of λ_i. Typically, the response time behaves as shown in Figure 3. The $\mathtt{pct}$%-tile response time increases as the request arrival rate increases. The increase is drastic as the arrival rate approaches capacity. Th_{resp} denotes a desirable quantile response time and λ_i^* is the maximum allowable load to meet the threshold Th_{resp} at server i.

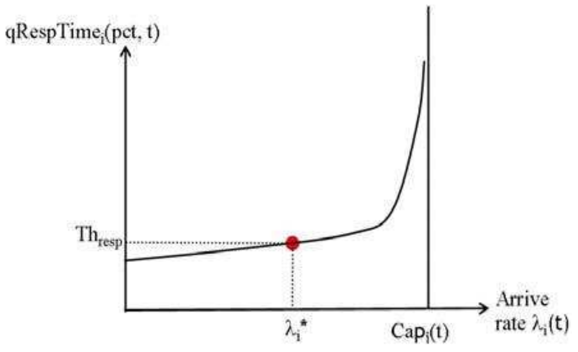


Fig. 3. Response time vs. arrival rate at server i.

In our algorithm, we assume homogeneous VMs and internal nodes. Thus we can drop subscript i and denote λ_{cloud}^* to be the maximum allowable load for any cloud server such that the pct%-tile response time is within Th_{resp}. Similarly, we defin λ_{int}^* for identical internal servers.

System operating regions: Given that systems can have bottlenecks once we scale up tier-1 or tier-2, we do not scale up indefinitel . We need to infer the system state and make sure the system is operating in good regions. We assume the system designer will provide an input graph depicting the operating regions as shown in Figure 4. The X-axis is the arrival rate λ_i of server i.

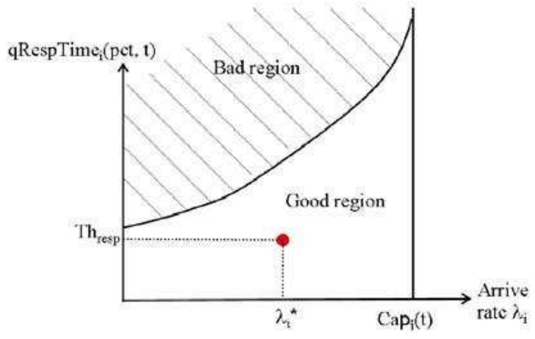


Fig. 4. Operating region of all servers

D. Scaling algorithm

We design the scaling algorithm based on feed-back control framework. The observed system output is the application performance metric, i.e. the response time. Our target is to keep the exponential moving average of pct%-tile response time below a threshold. We denote this metric at time t as $\mathtt{qRespTime_{sys}(pct,t)}$. Note that this is the overall system response time. The controller needs to decide the number of VMs added or released based on the changes of response time $\delta_{\mathtt{qRespTime}}(\mathtt{t})$ $=\mathtt{qRespTime_{sys}(pct,t)}$-$\mathtt{qRespTime_{sys}(pct,t-1)}$. We discuss three types of controllers.

```
if $\mathtt{qRespTime_{cloud}(pct^*,t)} > Th_{resp}$
  Let $\Delta S(t) = \frac{\Lambda_{cloud}(t)}{\lambda^*(t-1)} - numCloudS(t-1) - numMarkDelS(t)$
  if $\Delta S(t) + numMarkDelS(t) < 0$
    Mark $|\Delta S(t) + numMarkDelS(t)|$ active servers for deletion
  else
    for $j = 1$ to $\Delta S(t) + numMarkDelS(t)$
      if !detect_choke_point(numCloudS(t)),
        reactivate or create server $j$
        distribute $\lambda^*_{cloud}(t)$ to server $j$
      else break;
      endif
    endfor
  endif
  $numCloudS(t) = numCloudS(t-1) + j$
  if detect_choke_point(numCloudS(t)), //system choking
    instruct load balancer to drop the excessive load,
    $(\Delta S(t) + numMarkDelS(t) - j) * \lambda^*_{cloud}(t)$
  endif
```

Fig. 5. CloudFlex scaling algorithm: P controller

Controller with binary feedback: If we cannot infer detailed system parameters such as λ_{cloud}^*, we will rely on binary feedback on whether the target response time is met. There are two general approaches: Multiplicative Increase - Additive Decrease (MIAD) and Additive Increase - Additive Decrease (AIAD). MIAD is costly and suitable for applications with strict requirement on response time target. On the other hand, AIAD is cheap and suitable for applications with high tolerance on response time target.

Controller with λ_{cloud}^* estimation(P controller): We now describe our controller design based on λ_{cloud}^* estimation.

Our approach is based on the relationship of the arrival rate of the system $\Lambda_{sys}(t)$, average system response time $\mathtt{qRespTime_{sys}(t)}$ at the current time interval t, and the load at all individual server in the previous time period $t-1$ (i.e. $\lambda_i(t-1)$ for each server i).

Given the total arrival rate to the cloud servers $\Lambda_{cloud}(t)$, assuming a server in the cloud can still handle $\lambda^*_{cloud}(t-1)$, we can calculate the number of additional $\Delta' S(t)$ servers needed to add or shutdown as follows.

$$\Delta' S(t) = \frac{\Lambda_{cloud}(t)}{\lambda^*_{cloud}(t-1)} - numcloudS(t-1) \qquad (3)$$

Since there are $\mathtt{numMarkDelS(t)}$ number of servers marked for deletion that are still alive at time t (i.e. these servers are in the bleeding process), we only need to add or shutdown $\Delta S(t) = \Delta S'(t) - numMarkDelS(t)$. Our detailed algorithm for scaling is shown in Figure 5.

PI controller: Our P controller may be slow in catching the demand during a period where requests arrive in increasing rate. To deal with this situation, an integral component to accelerate the convergence process can be added. The resulting controller is a PI controller. The integral component is designed to be $I_{cum}(t) = K\sum_{t'=t-\tau}^{t}(\Lambda_{cloud}(t') - \Lambda_{cloud}(t'-1))$

where K and τ are tuning parameters to control how fast the system should respond to past cumulated load.

$$\Delta S(t) = \frac{\Lambda_{cloud}(t) + I_{cum}(t-1)}{\lambda^*_{cloud}(t)} - numCloudS(t-1) \quad (4)$$

E. Detecting choke point

We do not start all the new servers immediately and instead try to detect if the system is choked first If server j does not cause bottleneck, we load it up with $\lambda^*_{cloud}(t-1)$ requests per second. We do this one by one. If enough VMs are operating in bad regions, the choke point detection algorithm will return true. We base our decision on the fraction of servers operating in pre-define bad regions. If such a fraction exceeds a threshold γ then we say the system is choking.

F. Partitioning load between internal and cloud

Let numIntS and $\lambda_{int}{}^*(t)$ Let numIntS be the number of internal servers and $\lambda_{int}{}^*(t)$ be the load that corresponds to our threshold response time Th_{resp} for internal servers. We have,

$$\Lambda_{int}(t) = \lambda_{int}{}^*(t) \times numIntS \quad (5)$$

If $\Lambda_{int}(t)$ stays unchanged for several previous time windows but the response time of $qRespTime_{int}(t)$ of internal servers is still above or below Th_{resp}, then it means that our $\lambda^*_{int}(t)$ estimation is not accurate. We will reduce or add $\lambda^*_{int}(t+1)$ by δ_1 where δ_1 can be x% of $\lambda^*_{int}(t)$ for the next time window $t+1$. Subsequently, $\delta_i = \delta_1/i$.

V. Emulation Experiments

We firs evaluate the effica y of CloudFlex design in an emulation environment where the workloads are tightly controlled. Due to space constraints, we only describe the evaluation of the choke point detection algorithm.

A. Emulation Design

The emulation system has two components: load generator and simulated servers. The load generator creates multiple threads, each of which sends requests to the server simultaneously. Each thread emulates a client and thus the number of threads reflect the emulated load. Additionally it calculates the amount of time the server sleeps to simulate choking as

$$\frac{\text{number of active threads}}{\text{desired number of threads chokes the system} - 1} \quad (6)$$

The server is implemented as an Apache DSO module that locally executes heavily CPU-intensive operations and sleeps for the amount of time given as an HTTP argument. We used P controller described in Section IV-D.

B. Choke points simulation

We start with linear increase in the load as shown in Figure 6. The number of handled requests scales up linearly until the system reaches a choke point and our choke point detection algorithm successfully detects the choke point and stops scaling up cloud nodes. In the middle of the curve, we observe a slight decrease in throughput due to additional workload that exceeds the desired system capacity.

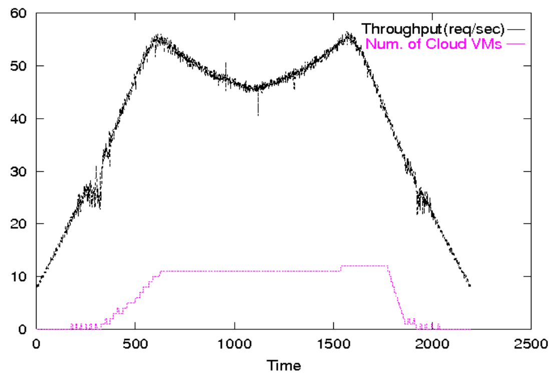


Fig. 6. System throughput vs number of cloud VMs

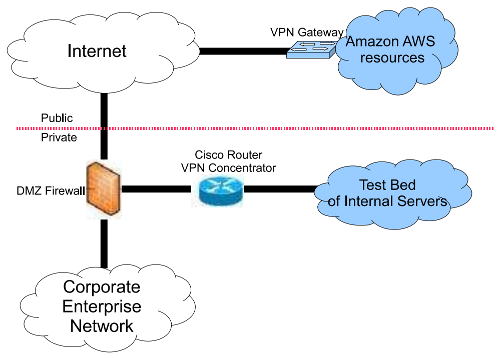


Fig. 7. Physical test bed for CloudStone deployment. Cloud resources are integrated into enterprise network with Amazon VPC. VPC resources are isolated from the Internet.

VI. CloudStone experiment

In this section we describe our Cloudstone experiment setup and the results.

A. Experiment Setup

We collaborate closely with our IT department to develop a testbed that reflect the enterprise environment. The testbed leverages our enterprise production resources inside our network. We virtualize the servers with Linux-based VMWare Workstation and set up Amazon Virtual Private Cloud (VPC) [1] to integrate enterprise and cloud resources securely and place the entire testbed behind our corporate fir wall for security. Figure 7 shows the topology of our testbed.

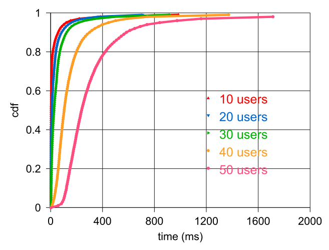
We deploy CloudStone [10], a benchmark for evaluating the performance of a web service. CloudStone includes three main components: a Faban Markov-chain-based load generator, an Olio Web 2.0 social networking application, and a structured database. We use P controller described in Section IV-D.

B. Maximum allowable load λ^*_{int} and λ^*_{cloud}

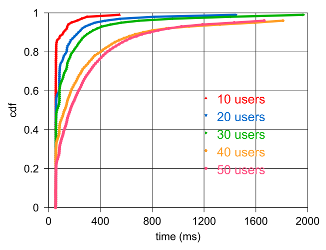
In this experiment, we attempt to determine the maximum allowable load for both internal and cloud physical servers. In particular, we pick a server (either internal or cloud worker) and configur the Faban load generator to gradually increase the workload toward it. The tuning knob is in terms of the number of concurrent users. Figure 8 plot the cdf of service response time versus different workloads for internal and cloud workers.

We observe the tradeoff between service response time versus workload as in our model in Figure 3. For example, with threshold $Th_{resp} = 1000$ms and percentile of 95%, the allowable loads of internal and cloud nodes are 50 and 30 users respectively. By examining the service request logs, we found that these are equivalent to $\lambda^*_{int} = 80$ requests/sec and $\lambda^*_{cloud} = 60$ requests/sec. Note that the requests are not homogeneous, i.e. certain requests are more expensive than others; however we experimentally observe that these parameters are conservative and reasonably consistent.

We measure the RTT between any two internal and cloud nodes to be 27 ms. In Figure 8b, the minimum response time of a cloud worker is at least 54 ms, i.e. $2 \cdot RTT$. We found the reason is that both the load balancer and the database stay inside the enterprise network; any requests directed to the cloud would incur an additional RTT to query the database.



(a) Internal worker



(b) External worker

Fig. 8. CDF of service response time of a single worker

C. Controller evaluation

Figure 9 shows that our controller converges in terms of the number of cloud usage. The total number of internal and cloud nodes are 5 and 20 respectively. In particular, with 100 concurrent users, the system scales out to 3-4 cloud nodes within 4 minutes. With 200 concurrent users, the system uses 8-9 cloud nodes after 6 minutes.

VII. Related work

Load balancer design from L2 to L7 for server, fir wall and cache is a mature technology [7]. However, load balancer typically assumes a fi ed set of servers in normal conditions. Load balancing and Auto-scaling exclusively using cloud VMs has been offered as a service by Amazon [3], [2] and Rightscale [9]. Lim et al [8] and Demberel et al [4] have proposed controller designs for applications using only cloud resources. Cloud-aware load balancing and scaling using both enterprise internal and cloud resources has not been studied as far as we know. Using both internal and external resources introduce new challenges such as secure connection between enterprise and cloud, and its throughput penalty; performance difference between enterprise servers and cloud servers. Whether to use internal or external servers depends

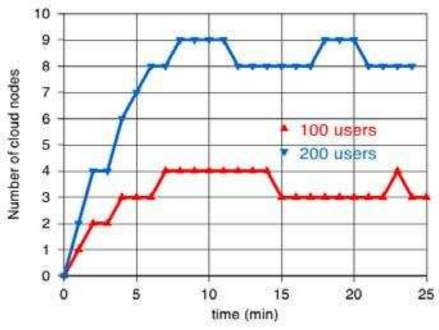


Fig. 9. Convergence in usage of cloud nodes

on the performance and cost tradeoff. We also detect system choke points and dropping requests when the system reaches a choke point. Our control target is response time of application request while [4] uses CPU load.

VIII. Conclusion

We have built a system called CloudFlex. CloudFlex enables enterprise to seamlessly use both internal and cloud resources for handling application requests. Our system addresses issues with system choke points, cloud resource responsiveness, and load balancing. Our scaling algorithm is general and does not depend on application details; it is based on general feedback control principles. We have built a testbed that consists of internal machines and Amazon EC2 instances. Our evaluation shows that CloudFlex react effectively to changing system loads; it properly scales out to the cloud as load increases and scales back in as load fades. CloudFlex operates in a stable manner with no observed instability.

IX. Acknowledgement

We are very grateful to the indepth discussion on convergence with Sem Borst and Sasha Stolyar at Bell Labs.

References

[1] Amazon AWS. Amazon virtual private cloud.
[2] A. AWS. Amazon auto-scaling. http://aws.amazon.com/autoscaling.
[3] A. AWS. Amazon elastic load balancing. http://aws.amazon.com/elasticloadbalancing.
[4] A. Demberel, J. Chase, and S. Babu. Reflect ve control for an elastic cloud application: An automated experiment workbench. In *First Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2009.
[5] M. A. et al. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, UC Berkeley, 2009.
[6] G. Inc. Gartner says worldwide cloud services market to surpass $68 billion in 2010. http://www.gartner.com/it/page.jsp?id=1389313, 2010.
[7] C. Kopparapu. *Load Balancing Servers, Firewalls, and Caches*. Wiley, 2002.
[8] H. Lim, S. Babu, J. Chase, and S. Parekh. Automated control in cloud computing: Challenges and opportunities. In *Workshop on Automated Control for Data Centers and Clouds (ACDC)*, June 2009.
[9] Rightscale. Rightscale adaptable automation engine. http://www.rightscale.com/products/features/adaptable-automation-engine%.php annote = .
[10] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.