Kingfisher: Cost-aware Elasticity in the Cloud

Upendra Sharma, Prashant Shenoy
University of Massachusetts
{upendra,shenoy}@cs.umass.edu

Sambit Sahu and Anees Shaikh
IBM Watson Research Center
{sambit,aashaikh}@us.ibm.com

***Abstract*—In this paper we present Kingfisher, a *cost-aware* system that provides efficient support for elasticity in the cloud by (i) leveraging multiple mechanisms to reduce the time to transition to new configurations, and (ii) optimizing the selection of a virtual server configuration that minimizes the cost. We have implemented a prototype of Kingfisher and have evaluated its efficacy on a laboratory cloud platform. Our experiments with varying application workloads demonstrate that Kingfisher is able to (i) decrease the cost of virtual server resources by as much as 24% compared to the current cost-unaware approach, (ii) reduce by an order of magnitude the time to transition to a new configuration through multiple elasticity mechanisms in the cloud, and (iii), illustrate the opportunity for further design alternatives which trade-off the cost of server resources with the time required to scale the application.**

I. Introduction

Today's cloud computing platforms support elastic on-demand allocation of server resources, while supporting a multitude of hardware configurations, at different price levels. Further, as application needs change over time, a number of elastic scaling mechanisms, ranging from replication to migration, are available to transition the application to a new hardware configuration. It is left to the application provider to choose a particular hardware configuration for her application and to choose a specific scaling mechanism to increase or decrease the application's provisioned capacity. In this paper, we develop a generalized provisioning framework for supporting elasticity in the cloud, which exploits the pricing differentials of various server configurations and chooses the most appropriate scaling mechanism to minimize transition overheads. Our paper makes the following key contributions:

Cost-aware elasticity. We present Kingfisher, a cost-aware system that integrates multiple elasticity mechanisms such as replication and migration and computes both a cost-optimized configuration for the desired capacity as well as a plan for transitioning the application from its current setup to its new configuration.

Prototype implementation and experimentation. We implement a prototype of our Kingfisher cloud provisioning engine, using the OpenNebula cloud toolkit [1], that incorporates our optimizations, and evaluate its efficacy on a laboratory cloud testbed.

Our experimental results (i) demonstrate that cost-aware elasticity can achieve up to 24% rental cost savings (for the pricing scheme shown in Table I), (ii) show that integrating multiple mechanisms such as migration and replication into a unified approach can double the cost savings, and (iii) demonstrate how our transition-aware approach can be employed to quickly provision capacity in scenarios where an application workload surges unexpectedly. Our experiments also show an order of magnitude reduction in the transition overhead.

II. Cloud Computing Background and Problem Definition

Amazon EC2 Cloud Platform - aws.amazon.com			
Server size	Configuration	Cost	$/core
Small	1 ECU, 1.7GB RAM, 160GB disk	$0.085 / hr	$0.085
Large	4 ECUs, 7.5GB RAM, 850GB disk	$0.34 / hr	$0.085
Med-Fast	5 ECUs, 1.7GB RAM, 350GB disk	$0.17 / hr	$0.034
XLarge	8 ECUs, 15GB RAM, 1.7TB disk	$0.68 / hr	$0.085
XLarge-Fast	20 ECUs, 7GB RAM, 1.7TB disk	$0.68 / hr	$0.034
NewServer's NS Cloud Platform - www.newservers.com			
Small	uni-core 2.8GHz, 1 GB RAM, 36GB disk	$0.11 / hr	$0.11
Medium	dual 3.2 GHz, 2 GB RAM, 146GB disk	$0.17 / hr	$0.085
Large	4-core 2.0GHz, 4GB RAM, 250 GB disk	$0.25 / hr	$0.063
Fast	4 core 3.0 GHz, 4 GB RAM, 600GB disk	$0.63 / hr	$0.158
Jumbo	8 core 2.0GHz, 8GB RAM, 1TB disk	$0.60 / hr	$0.075

TABLE I: Cloud server configurations and their prices. For EC2, ECU= 1.2 GHz Xeon or Optron circa 2007.

Our work assumes a cloud platform that rents computing capacity to its customers. The cloud platform is assumed to use a usage-based pay-as-you-go pricing model that allows servers to be rented on a fine time-scale (e.g., hourly). We assume that these servers can be allocated or deallocated on-demand by a customer for her application in order to elastically match capacity to fluctuating workload demand. From an application standpoint, these capacity changes can be made either via *replication*—by adding or removing replicas—or via *migration*—by altering the server configuration to a larger or a smaller server. If a specific cloud platform exposes a subset of these mechanisms (e.g., the EC2 cloud does not presently support live migration), then the system must take these constraints into account when provisioning resources.

We assume that the platform offers N different servers types for rent, each with a different hardware configuration and a different rental cost. The pricing of servers is assumed to be arbitrary and can increase sub-linearly with the number of cores per system (convex pricing) or super-linearly with cores (concave model). Table I depicts two different pricing models for two real cloud platforms—the pricing is convex for most popular choices (e.g., small, medium, large) and becomes arbitrary for higher-end configurations.

Further we assume a multi-tier cloud application whose workload demand change over time—due to incremental growth or sudden change in popularity. In such cases, the application will need to be reconfigured by dynamically altering its capacity. To do so, given a certain hardware configuration that is already in use, we must determine a new configuration

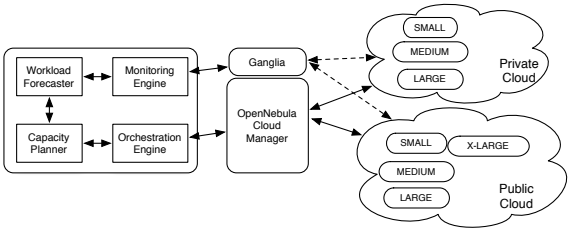


Fig. 1: Kingfisher Architectural Overview

that specifies how many cloud servers and of what types to use for each tier to sustain the new peak workloads at each tier. Furthermore, we must also specify a *plan* for morphing each tier from its current configuration to the new configuration using mechanisms such as migration, replication or shutdown. Thus, for our elastic provisioning decisions, we are interested in minimizing two types of costs: (i) the *rental cost* of the servers, which depends on the hardware configuration chosen by the system; and (ii) the *transition cost* (the latency overhead), which depends on the elasticity mechanism chosen for transition.

III. Kingfisher System Overview

Kingfisher is a system that supports elasticity in today's public and private cloud computing platforms. Kingfisher presently supports both Amazon's EC2 public cloud and Xen-based private clouds [2]. Kingfisher combines an application-centric provisioning engine with a cloud management platform. It assumes a virtualized cloud platform and provides support for virtual machine (VM) deployment, VM image management, in conjunction with elastic provisioning. Kingfisher uses a modified version of the OpenNebula toolkit to implement its cloud management mechanisms—e.g., to deploy/undeploy VMs on a set of servers in a private-cloud, and to reconfigure applications with more or less capacity.

Figure 1 depicts the high-level architecture of Kingfisher. Its key components include (i) the monitoring engine, which monitors the workload and resource usage on servers, (ii) workload forecasting subsystem, which uses the monitored workload to predict future demand, (iii) the capacity planner and orchestration engine, which implement the cost-aware elasticity algorithms that are discussed in the next Section. Additional details of these components can be found in [2].

IV. Cost-aware Elasticity Algorithms

In this section, we discuss Kingfisher's cost-aware elasticity algorithms in more detail.

A. Rental Cost-aware Provisioning

Given the estimated peak workload λ_i that must be sustained at each tier i, the goal of our approach[1] is to compute *which type* of cloud server to use and *how many* at each tier so as to minimize rental cost; the provisioned servers must have the collective capacity to service at least λ_i request/s while meeting tier's response time SLAs.

Our cost-aware provisioning algorithm involves two steps: (1) for each type of cloud server, compute the maximum request rate that the server can service at a tier, and (2) given these server capacities, compute a least-cost combination of servers that have an aggregate capacity of at least λ_i.

Step 1. Empirical Determination of Server Capacities. For each server configuration supported by the cloud platform (e.g., small, medium, large), we must first determine the maximum request rate that each hardware configuration can sustain for this application.

Kingfisher employs a systems approach to empirically derive capacities of different server types—it estimates the maximum server capacity by running the application on different hardware configurations, subjecting them to a gradually increasing synthetic workload, and determining the point where the server saturates (and begins violating SLAs or dropping requests). Such an empirical approach is more accurate than analytical queuing approaches [4] since capacities are computed using actual measurements on real hardware and can account for software artifacts since the actual application behavior is used when estimating capacities. In production environments, a system such as JustRunIt [5] can be used to clone virtual machines and run the cloned application on a sandboxed server in order to empirically profile and measure server capacities.

Step 2. Determining Server Configurations. Given a cloud platform with M different types servers (e.g., small, medium, large), let C_j and p_j denote that capacity (maximum request rate) and the rental cost of server type j. Let λ denote the peak workload request rate for which capacity needs to be provisioned at a tier. Then the problem of rental cost-aware provisioning is stated as $\text{minimize} \sum_{j=1}^{M} n_j p_j \;\; s.t. \sum_{j=1}^{M} n_j C_j \geq \lambda$, where n_j denotes the number of servers of each type that is chosen. This optimization problem can be formulated and solved as an integer linear program, as discussed later in this section. The ILP solution yields $(n_1, n_2, \ldots, n_M)$ — which tells the application provider how many servers of each type should be chosen for the application tier.

B. Transition Cost-aware Provisioning

Our transition cost-aware provisioning method is designed to address the scenario where the transition latency incurred in moving the application to new configuration is optimized. Thus, our provisioning approach is to estimate the latency of using different provisioning mechanisms – replication, migration and resizing as follows:

Local resizing: Local resizing involves using the hypervisor API on a machine to modify the resource allocation of a virtual machine (e.g., to give it more RAM or to allocate it additional cores or CPU shares). This can be done efficiently with minimal overheads.

Replication: Starting up a new instance (replica) of an application tier involves copying the machine image of the OS/application from central storage to the disk on the new

[1]The future peak workload can be estimated using any workload forecasting technique, e.g., time-series-based predictions [3].

server, starting up the OS and the application replica, and reconfiguring the application to make it aware of the new replica. The latency can be estimated as $\frac{D}{r}+b$, where D is the size of the disk image, r is the network bandwidth available for the copy operation and b is a constant representing the OS and application startup time.

Live migration: Live migration of a virtual machine from one server to another involves copying the memory state of the VM to new server while the application is running. Typically live migration mechanisms assume that the disk state of the VM is maintained on a shared file system. Hence, the latency of the live migration is $w \cdot \frac{R}{r}$, where R is the size of the VM's RAM, r is the network bandwidth available for the copy operation, and w is a constant that captures the mean number of times a memory page is (re)sent over the network (due to dirtying of pages during the migration process).

Shutdown-migrate. Migration can be "simulated" in a public cloud by suspending the application, converting its disk state into a new machine image, copying the machine image to a new server and restarting the image on the new machine. Since the disk state may need to be copied twice, once to construct a new machine image and then to copy the machine image to the new server , the latency of this approach is $2\frac{D}{r}+b$.

The transition-aware approach then attempts to minimize this overhead by preferring mechanisms that incur lower copying overheads (and hence, lower latencies). Like before, this can be formulated and solved as a ILP optimization problem as discussed next.

ILP formulation: Both rental and transition cost-aware provisioning problems can be stated using the following integer linear program (ILP). Let M denote the number of server types supported by the cloud platform; Let p_j denote the rental cost for server type j and let C_j denotes its maximum capacity. Let λ denote the peak workload for which the application needs to be provisioned, and let N denote the maximum number of servers that could be needed to satisfy λ (any large number can be chosen as N). Let T denote the number of the provisioning mechanisms supported by the platforms (e.g., replication, migration, resizing). Then the objective function for minimizing *r*ental cost is

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} p_{(j)} x_{ijk} \tag{1}$$

subject to the constraints

$$\sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} x_{ijk} C_j \geq \lambda; \quad \sum_{j=1}^{M} \sum_{k=1}^{T} x_{ijk} = 1, \forall i \tag{2}$$

The terms x_{ijk} is an integer variable in the ILP that can take values of of 0 or 1; A value of 1 indicates that server i is transformed into server-type j using a provisioning mechanism k (e.g., replicate or migrate); a value of 0 indicates that that option was not chosen by the ILP. The output of the ILP is set of values x_{ijk} that denotes which server types are chosen and also specifies a plan for transitioning for each server i to the new server type j using method k (replicate. migrate etc).

The ILP for *transition-aware* provisioning is identical to the previous one except for the optimization criteria which must minimize the transition cost rather than rental cost, and thus Equation (1) changes to: $\min \sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} m_{ijk} x_{ijk}$. Here m_{ijk} be the cost of transforming server i to server-j using mechanism k. This cost is estimated using the above equations that capture the overhead of replication, live migration etc. Like before, $x_{ijk} \in \{0, 1\}$ indicate whether the final solution will employ technique k to transition server i to server type j.

We have implemented a greedy-type heuristic with a worst case bound of 2 [6] for an approximate solution of the above ILP. The heuristic sorts $x_{i,j,k}$ in increasing order of p_j/C_j and then finds the smallest list of $x_{i,j,k}$'s which satisfy Eq. (2). Once an $x_{i',j',k'}$ has been chosen for a particular $i = i'$, we skip the remaining $x_{i',j,k}$; this ensures that we satisfy the constraint in Eq. (2).

V. Experimental Study of Elasticity: Methodology and Setup

We evaluate the efficacy of Kingfisher's elasticity mechanisms using a laboratory cloud testbed; results from our Amazon EC2 evaluation are in [2]. We conduct experiments with a number of mechanisms for achieving elasticity in the cloud, starting with cost-awareness with replication, and adding migration and transition-cost awareness. Our goal is to understand whether these mechanisms can further improve cost-aware elasticity support beyond the traditional replication-only approach. Our evaluation metrics are the overall rental cost of the virtual servers supporting the application deployment, the cost in terms of latency to change or scale the configuration, and the latency to achieve target application response time after a configuration change.

Cost-aware elasticity mechanisms:

Cost-aware vs Cost-oblivious with Replication: First, we consider replication as the only method for supporting elasticity - the typical method to provide elasticity. Here we compare between resource cost-oblivious (CO-R) and cost-aware (CA-R) approaches to illustrate the benefit of cost-aware approaches.

Migration: Second, we introduce VM migration in addition to replication as the means for supporting elasticity to investigate benefits of these mechanisms beyond replication based elasticity We refer them as CA-RM and CO-RM.

Transition cost-aware: Third, we account for transition cost, defined as the time taken to execute the configuration change to understand its effect on supporting elasticity. We compare the transition cost aware (TA-RM) and transition-cost oblivious (TO-RM) approach to explicitly account for such costs as part of elasticity study.

Experimental Testbed and Workload: We use a virtualized Xen/Linux-based laboratory cloud platform for our experiments. Our private cloud platform is built on two types of servers: 8-core 2GHz AMD Opteron 2350 servers and 4-core 2.4 GHz Intel Xeon X3220 systems. All machines run Xen 3.3 and Linux 2.6.18 (64-bit kernel). Our platform is assumed to support small and large servers, comprising 1, 2 and 4 cores, respectively.

We use the java implementation of TPC-W [7] for our experiments. TPC-W is a multi-tier web benchmark that represents an e-commerce web application comprising of a Tomcat application tier and a mysql database tier. We created a virtual appliance of TPC-W on Centos 5.2 using a modified version Tomcat-5.5.27 as the servlet container and mysql-5.0.45 as the backend database-server; our modified Tomcat server logs the service time of each request, in addition to other default per-request statistics. We also created a dispatcher appliance using the HAProxy load balancer; the dispatcher is used to distribute and load balance across all TPC-W replicas. We used the browsing workload of the TPC-W specification to to trigger the provisioning. We tested each approach on two types of workload patterns: 1.) smoothly increasing workload (*small-jump* workload) 2.) Sharply increasing workload (*large-jump* workload).

VI. Experimental Results

We now present our experimental results.

Profiling Server Capacities: We configured TPC-W with both tiers in a single VM, and ran this VM on various server instances of the private cloud, mentioned as above. We refer to the single-core system as "small," dual-core as "medium," and the quad-core as "large". In each case, we gradually increased the workload seen by the TPC-W application until the server saturated and began dropping requests. Fig. 2 plots the empirically derived capacities for various multi-core configurations on our Intel and AMD systems in our private cloud. The figure shows that server configurations on each processor have a very different capacity and in both cases they scale non-linearly.

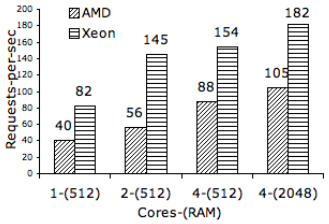
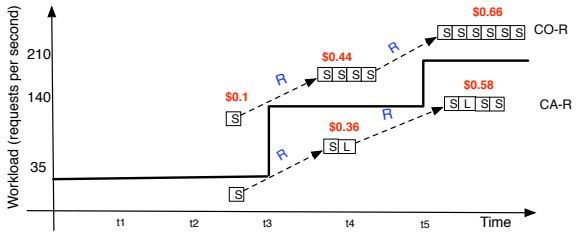
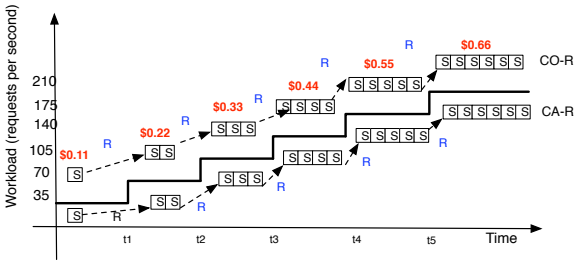


Fig. 2: Non-linear scaling behavior of TPC-W across multi-core servers configurations.

Cost-aware versus Cost-oblivious Provisioning: We first compare the cost-aware approach to a cost-oblivious approach (which ignores rental costs when provisioning servers) in a restricted setting where only "replication" is used to modify the deployment. We denote these two approaches as CA-R (cost-aware with replication) and CO-R (cost-oblivious with replication). In these experiments, for simplicity we used two types of server-classes, small and large, with the NS-cloud platform's pricing model, detailed in Table-I. We increase the request rate (λ) from 35 to 210 req/s. Fig. 3a depicts the server configurations chosen by the CA-R and CO-R approaches (and the resulting rental cost) when the workload increases sharply in a few large steps. *We see that, even for this relatively small deployment, cost-aware shows up to 12% reduced rental cost for the same provisioned capacity.*



(a) *large-jump* workload



(b) *small-jump* workload

Fig. 3: Cost-aware versus cost-oblivious provisioning

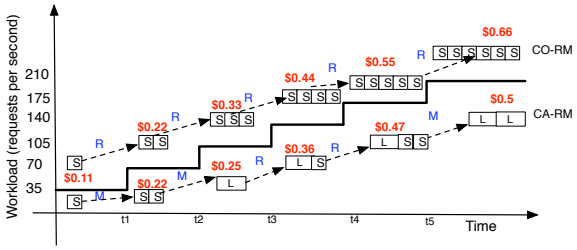
If the workload increases more steadily, as shown in Fig. 3b, both approaches choose *identical* configurations. With replication as the only elasticity mechanism, and slowly increasing workload, the cost-aware approach is not able to find opportunities for further cost improvement.

Benefits of adding Migration mechanism: We next consider the benefit of migration by enabling both mechanisms to modify the deployment. Our provisioning algorithms are able to consider a larger set of feasible configurations, which can yield higher savings in the rental cost. Figure 4b compares the two approaches as the workload grows in large jumps. The cost-aware approach (CA-RM) shows a benefit as high as 24% over cost-oblivious (CO-RM), twice the relative benefit as with using replication-based elasticity alone. For the steadily growing workload, shown in Figure 4a, the cost-aware algorithm shows a similar benefit over cost-oblivious. – *Thus, by adding the migration mechanism, cost-aware provisioning is able to improve the rental cost by 24%.*

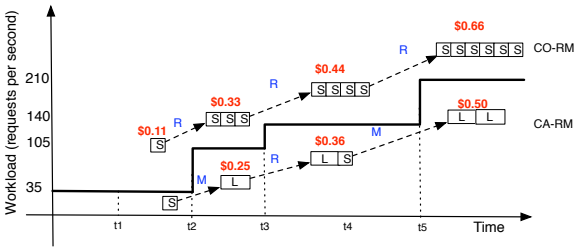
Figure 5 shows the changing request-rate applied to the TPC-W application, and the corresponding average response-time during the experiments. *By leveraging migration mechanism, the CA-RM approach is able to be much more responsive to provisioning requests*. For example, for the first large increase in workload, CA-RM chose a migration while CO-RM selected 2 replications, hence cost-aware finished the task in 10 sec as opposed to 1000 sec for cost-oblivious. This is because *live-migration* copies only the RAM-image of the VM, which is an order of magnitude faster than copying the disk image in replication.

Transition cost-aware Provisioning: Kingfisher's transition cost-aware approach can quickly provision additional capacity in the cloud when the workload surges suddenly by making elasticity decisions based on the time overhead. However, by focusing on rapid reconfiguration, transition cost-aware provisioning may not produce the minimal rental cost.

Figure 6 shows that the transition cost-aware approach is

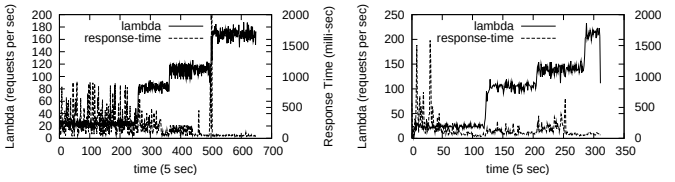


(a) small workload increase



(b) large workload jumps

Fig. 4: Benefits of using replication and migration in a unified provisioning approach.



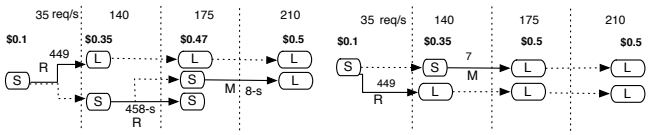
(a) Cost Oblivious (b) Cost Aware

Fig. 5: Application Performance during cost-aware and cost-oblivious provisioning for *large-jump* workload.

able to pick lower transition time configurations, while the other approach opts for a lower rental cost configuration but takes an order of magnitude more time. For example, transition cost-oblivious takes 458 seconds as opposed to 7 seconds for transition cost-aware, when the workload jumps from 140 req/s to 175. We note that the transition cost-aware approach opts to perform a single live-migration when workload jumps from 140 to 175 req/s. This is because live-migration copies only the RAM of the VM, and the algorithm opts for this because it tries to minimize the total data copying cost. This live-migration causes a slight over-provisioning and thus when the workload jumps from 175 req/s to 210 req/s no reconfiguration is necessary.

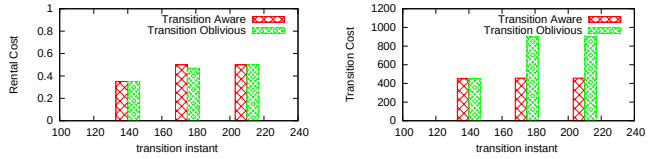
Figure 7(a) and (b) show the rental cost and transition cost for the scenario in Figure 6 . We observe that *by having migration as additional mechanism for elasticity, it is possible to provide elasticity more rapidly*.

Overall, the experiment demonstrates that copying of memory state during a live migration incurs lower latencies than copying of disk images during replication. Hence, live migration may be preferred, whenever feasible, to reduce transition costs.



(a) Transition-cost oblivious (b) Transition-cost aware

Fig. 6: Comparison of a transition-cost aware system with a transition-cost oblivious system.



(a) Rental Cost Comparison (b) Transition Cost Comparison

Fig. 7: Comparison of rental cost and time of transition of transition-cost aware and oblivious system

VII. Concluding Remarks

Since today's cloud platforms offer a plethora of different server configurations for rent and price them differently on a cost-per-core basis, we argued that these pricing differentials can be exploited by an application provider to minimize the rental cost of provisioning a certain capacity. We proposed a new cost-aware provisioning approach for cloud applications that can optimize either the rental cost for provisioning a certain capacity or the transition cost of reconfiguring an application's current capacity. Our approach exploits both replication and migration to dynamically provision capacity and uses an integer linear program formulation to optimize cost. We prototyped a cloud provisioning engine, using OpenNebula, that implements our approach and evaluated its efficacy on a laboratory-based Xen cloud. Our experiments demonstrated the cost benefits of our approach over prior cost-oblivious approaches and the benefits of unifying both replication and migration-based provisioning into a single approach.

Acknowledgements: This research was supported in part by NSF grants CNS-0855128, CNS-0916972, CNS-0720616, and an IBM faculty award.

References

[1] "Opennebula," http://www.opennebula.org.
[2] U. Sharma, P. Shenoy, S. Sahu, and S. Anees, "A system for elastic cost-aware provisioning in the cloud," in *University of Massachusetts, Technical Report UM-CS-2010-005*, May 2010.
[3] J. Hellerstein, F. Zhang, and P. Shahabuddin, "An Approach to Predictive Detection for Service Management," in *Proceedings of the IEEE Intl. Conf. on Systems and Network Management*, 1999.
[4] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An Analytical Model for Multi-tier Internet Services and Its Applications," in *Proc. of the ACM SIGMETRICS Conf.*, Banff, Canada, Jun. 2005.
[5] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner, "Justrunit: Experiment-based management of virtualized data centers," in *USENIX 09*, June 2009.
[6] J. Csirik, J. B. G. Frenk, M. Labbé, and S. Zhang, "Heuristics for the 0–1 min-knapsack problem," *Acta Cybern.*, vol. 10, no. 1-2, pp. 15–20, 1991.
[7] TPCW, "Java implementation," Website, http://tpcw.deadpixel.de.