A Cost-aware Elasticity Provisioning System for 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 design alternatives which trade-off the cost of server resources with the time required to scale the application.**

I. Introduction

Cloud computing has emerged as a new IT delivery model in which an organization or individual can rent remote compute and storage resources dynamically, using a credit card, to host networked applications "in the cloud." . The appeal of cloud computing lies in its usage-based pricing model – organizations only pay for the resources that are actually used, and can flexibly increase or decrease the resource capacity allocated to them at any time. This *elasticity* provided by Cloud computing can yield significant cost savings when compared to the traditional approach of maintaining an expensive IT infrastructure that is provisioned for peak usage – organizations can instead simply rent capacity, and grow and shrink it as the workload changes.

Cloud environments enable flexible, elastic provisioning by supporting a variety of hardware configurations and mechanisms to add or remove server capacity. However this flexibility also raises new challenges for application providers: (i) given several available resource configurations for a particular workload, which one to choose, and (ii) how best to transition from one resource configuration to another to handle changes in workload. The first decision arises from the availability of a number of server configurations, each with a different amount of virtual CPU cores, memory, and disk space to satisfy the same resource requirements. The array of available hardware configurations lead to a number of different ways to configure a typical multi-tier Web application. Further, these server configurations are typically not priced linearly with server capacity. For instance, a quad-core server may not be priced four times the price of four single-core servers. As shown in Table I, depending on the exact configuration, the price per core of a server may be higher or lower than the cost of a single-core system, and a careful selection of actual configuration may lower the total infrastructure cost. The second decision arises when adding more server capacity to accommodate an increase in the application request volume, for example. There is a similar array of choices in determining the new resource configuration (e.g., add a new replica or move the application to a larger server), as well as different costs or overheads based on the mechanism used to make the transition to the new target configuration.

In this paper, we present a new approach for dynamically provisioning virtual server capacity that exploits pricing models and elasticity mechanisms to select resource configurations and transition strategies that optimize the incurred cost. Our paper makes the following 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. Kingfisher's algorithms can take into account the price differentials in the per-core cost of different server types to minimize the *infrastructure cost* of provisioning a certain capacity. Kingfisher also minimizes the time to add extra capacity using different elasticity mechanisms (we call this time as *transition cost*). We formulate our provisioning problem as an integer linear program (ILP) to account for both infrastructure and transition cost for deriving appropriate elasticity decisions.

Prototype implementation and experimentation on public and private clouds. 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 both a private laboratory-based Xen cloud and the public Amazon EC2 cloud. Our experimental results (i) demonstrate that cost-aware elasticity can achieve 24% infrastructure cost savings for the same capacity in the private cloud, and up to 35% in EC2 over cost-oblivious provisioning approaches, (ii) demonstrate 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. In our experiments, we observed transition time improvements of 2x in the private cloud and up to 6x in EC2 using transition-aware elasticity.

The Case for Cost-aware Elasticity: While there has been significant research on dynamic capacity provisioning for data center applications, there are three key differences between the prior work and capacity provisioning in the cloud. First, some of prior work on dynamic provisioning has been *cloud provider centric*, where the data center provider attempts to maximize resource utilization by dynamically allocating a set of servers across hosted applications with varying workload demands (and attempts to statistically multiplex as many applications as possible on the data center). In contrast, the problem articulated in this paper requires a *customer-centric* view, where each customer ("application provider") individually optimizes their capacity needs by choosing the best server configuration that matches their needs. Cloud provider centric approaches attempt to maximize revenue while meeting an application's SLA in the face of fluctuating workloads, while a customer-centric approach attempts to minimize the cost of renting servers while meeting the application's SLA.

Second, the prior work on dynamic provisioning has not been *cost-aware*. By being cost-oblivious, prior approaches assume that so long as the desired capacity is allocated to the application, the choice of exact hardware configuration is immaterial. That is, the unit cost per core is assumed to be identical, making an N-core system equivalent, from a provisioning perspective, to an N-core systems with single cores. In the cloud context, however, the choice of the configuration matters, since the pricing per core is not uniform. Hence, Kingfisher must take the infrastructure costs of servers into account during provisioning.

Third, much of the prior work on provisioning has employed replication as the primary means to increase an application's capacity. The application is assumed to be replicable, and workload increases are handled by adding additional server instances to the application's pool of servers. An alternative method for capacity provisioning is to employ migration, where an application and its data are migrated to larger capacity server (e.g., a server with more cores) to handle workload growth. As we will show in this paper, Kingfisher considers both replication and migration when choosing the best method of transition the application from one capacity configuration to another.

II. Cloud Background and Problem Statement

Consider a cloud computing platform that offers N heterogeneous server configurations for rent, each with a different rental cost (infrastructure cost). The pricing of servers is assumed to be arbitrary. Thus, the pricing can be convex, where the cost per-core increases sub-linearly with the number of cores, or concave where more the cost of more capable servers increases super-linearly with the number of cores, or arbitrary where some other pricing formula is employed. As noted in Table I, both the EC2 cloud and the NewServer (NS) cloud platform employ a convex function for their most popular choices (e.g., small, medium, large) and the pricing model becomes arbitrary when the "high-CPU" or "fast CPU" configurations are taken into account.

Amazon EC2 Cloud Platform			
Server size	Configuration	Cost/hr	$/core
Small	1 ECU, 1.7GB RAM, 160GB disk	$0.085	$0.085
Large	4 ECUs, 7.5GB RAM, 850GB disk	$0.34	$0.085
Med-Fast	5 ECUs, 1.7GB RAM, 350GB disk	$0.17	$0.034
XLarge	8 ECUs, 15GB RAM, 1.7TB disk	$0.68	$0.085
XLarge-Fast	20 ECUs, 7GB RAM, 1.7TB disk	$0.68	$0.034
New Server's NS Cloud Platform			
Small	1-core 2.8GHz, 1 GB RAM, 36GB disk	$0.11	$0.11
Medium	2-core 3.2 GHz, 2 GB RAM, 146GB disk	$0.17	$0.085
Large	4-core 2.0GHz, 4GB RAM, 250 GB disk	$0.25	$0.063
Fast	4 core 3.0 GHz, 4 GB RAM, 600GB disk	$0.53	$0.133
Jumbo	8 core 2.0GHz, 8GB RAM, 1TB disk	$0.60	$0.075

Table I
CLOUD SERVER CONFIGURATIONS AND THEIR PRICES. FOR EC2, 1 ECU= 1.2 GHZ XEON OR OPTRON CIRCA 2007.

We assume that these servers can be allocated or deallocated on-demand by a customer for her application. From an application standpoint, these capacity changes can be made either via *replication*—by adding or removing replicas—or via *migration*—by moving the application 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 our approach must take these constraints into account when provisioning resources. We assume that an application is distributed with k interacting components (e.g., k tiers in multi-tier applications); each tier has an SLA associated with it that must be met by provisioning sufficient capacity to service that tier's workload.

Given such a cloud platform, the goal of our work is to develop a system that supports elasticity for applications by (i) choosing the most cost-effective elasticity mechanism (e.g., replication, migration) when adding or removing capacity, and (ii) choosing the most cost-effective server configuration. The elasticity problem arises both when initially provisioning/deploying an application in the cloud as well as during any subsequent reconfiguration.

Initial Provisioning: Assuming an application with k independent components/tiers, let λ_i denote the peak estimated workload seen at tier i. Then, the initial deployment problem is one of determining *how many* cloud servers to provision for each tier and of *what type* such that the infrastructure cost is minimized and a peak workload of λ_i can be sustained at each tier while meeting per-tier response time SLAs. Since the desired capacity can be satisfied using multiple hardware configurations , the goal is to choose the cheapest configuration that meets the needs of each tier. We compute the initial configuration and deploy the application.

Subsequent provisioning: Once an application has been deployed on the cloud, its workload demands may change over time—due to incremental growth or sudden change in popularity. In such cases, the application will need to be reconfigured by dynamically increasing (or decreasing) the capacity at each tier. The problem of subsequent re-provisioning is one where, given a certain server configuration that is already in use, we must determine a new configuration that specifies how many cloud servers and of what types to use for each tier to sustain the new peak workloads of λ'_i at tier i. Furthermore, we must also specify a *plan* for morphing each tier from its current configuration to the new configuration using mechanisms such as resizing, migration or replication. Thus, for subsequent provisioning decisions, we are interested in minimizing two types of costs: (i) the *infrastructure cost* of the servers, and (ii) the *transition cost*, defined as the latency, to move the current to the new configuration.

Depending on the scenario, a customer may be interested in optimizing the infrastructure cost, the transition cost or some combination of the two. For instance, steady growth in workload volume can be handled by computing a new configuration that minimizes the infrastructure cost of servers. In contrast, a sudden surge in workload—caused by a flash crowd—will require additional capacity to be brought online as quickly as possible. In this scenario, it is more important to reduce the latency to bring additional capacity online even if it implies choosing a configuration that incurs a somewhat higher infrastructure cost. Such a transition cost aware approach must consider different configurations that offer the same capacities and pick the one that offers the fastest migration path.

III. Cost-aware Elasticity Algorithms

Any dynamic provisioning algorithm involves two steps: (i) *when* to invoke the provisioning algorithm, and (ii) *how* to provision capacity so as to minimize infrastructure or transition cost.

When to provision? The provisioning algorithm can be triggered in a proactive or a reactive manner. A proactive approach uses workload forecasting techniques to determine when the future workload will exceed currently provisioned capacity and invokes the algorithm to allocate additional servers before this capacity is exceeded [2]. In contrast a reactive approach uses thresholds on resource utilization or on SLA violations to trigger the need for additional capacity. A combination of predictive and reactive approach is also employed to handle the prediction inaccuracy and also to avoid oscillations in provisioned capacity due to oscillations[1] in workload [13]. The issue of proactive or reactive invocation is orthogonal to that of cost-aware provisioning, and hence in this paper we choose perfect forecaster, i.e. a forecaster that knows the workload in advance. Next we discuss *how* to provision for optimizing infrastructure/transition cost.

A. Infrastructure Cost-aware Provisioning

Given the estimated peak workload $\lambda_1, \lambda_2, \lambda_k$ that must be sustained at each tier, the goal of our approach is to compute which type of cloud server to use and how many at each tier so as to minimize infrastructure cost; the provisioned servers must have the collective capacity to service at least λ_i request/s at tier i 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 configuration can sustain for this application. This information is used in the subsequent step by our provisioning algorithm to determine how many servers of a particular type will needed to service the peak workload λ_i. Clearly, the maximum request rate (i.e., the server capacity) depends on the nature of the application, its workload mix and the server type.

One possible approach for estimating the maximum workload that can be serviced by a particular server type is to employ queuing theory [3], where the server is modeled as a queuing system and queuing theoretic results are used to derive a relationship between the request rate, service times of requests, and the response time SLA. This approach can not account for software artifacts that limit the application capacity from scaling with the number of cores, causing the queuing-based model to overestimate the capacity of multi-core systems.

To overcome this drawback, we employ a systems approach that uses empirical profiling—Kingfisher 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 . Such an empirical approach is more accurate 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. The approach, however, requires an application provider to carefully set up and profile the application on various hardware configurations supported by the cloud platform, and such profiling is more involved than the simple measurements required by the queuing approach. We note, however, that a system such as JustRunIt [4] that can clone virtual machines and run the cloned application on a sandboxed server can be exploited to reduce the overheads of such an empirical approach. Once

[1] inaccurate workload prediction can lead to a rapid oscillation of the workload forecast between its increase and decrease

the maximum request rates of the various servers supported by the cloud platform have been determined, this information is subsequently used by the provisioning algorithm.

Step 2. Determining Server Configurations. Consider a cloud platform with M different types of servers (e.g., small, medium, large). Let C_j and p_j denote that capacity (maximum request rate) and the infrastructure cost of server type j. Let λ denote the peak workload request rate for which capacity needs to be provisioned at a tier. The problem of infrastructure cost-aware provisioning is stated as

$$\text{minimize} \sum_{j=1}^{M} n_j p_j, \tag{1}$$

such that

$$\sum_{j=1}^{M} n_j C_j \geq \lambda, \tag{2}$$

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. Notice that the ILP can handle both the capacity increase and capacity decrease.

B. Transition Cost-aware Provisioning

While our infrastructure cost-aware provisioning algorithm minimizes recurring infrastructure costs, it does not account for (or optimize) the transition latency to move from the old configuration to the new—a factor that depends on the size of the application's disk and/or memory state. In many cases, this latency is important, especially when additional capacity needs to be added to the application quickly (e.g., during a flash crowd).

To be able to handle such scenarios, the provisioning approach must be able to estimate the latency of using different provisioning mechanisms, such as replication, migration and resizing. By taking into account the latency of such mechanisms, a configuration that minimizes such overheads is chosen. We estimate the overhead of these mechanisms 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 (the latency is on the order of tens of milliseconds). Hence, local resizing is always the most desirable option to scale a VM's capacity. However, since the physical server may lack sufficient idle capacity for resizing, the algorithm must frequently resort to other options.

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 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 (memory pages that are dirtied during the copy phase are iteratively resent). 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. While live migration is a implemented in most popular hypervisors such as Xen and VMware, some public clouds such as Amazon's EC2 do not currently expose this option. 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[2], 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 the lower copying overheads (and hence, lower latencies). Like before, this can be stated and solved as an ILP optimization problem as discussed next.

C. An ILP-based Elasticity Algorithm

Both infrastructure 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 infrastructure cost[3] 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 *infrastructure cost* is

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} p_j x_{ijk} \tag{3}$$

[2] In Amazon's EC2, the disk state must be uploaded to its S3 storage system as a machine image and then copied over to the new server, resulting in two copy operations

[3] Price changes are handled currently by updating the pricing parameters and recomputing the provisioning solution.

subject to the constraints

$$\sum_{i=1}^{N}\sum_{j=1}^{M}\sum_{k=1}^{T} x_{ijk} C_j \geq \lambda \tag{4}$$

$$\sum_{j=1}^{M}\sum_{k=1}^{T} x_{ijk} = 1, \forall i \tag{5}$$

The terms x_{ijk} is an integer variable in the ILP that can take values 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). If this is the first time the application is being deployed onto the cloud, the current configuration is empty; for subsequent (re)provisioning, the plan specifies how the current configuration is to be morphed into the new configuration (e.g., using replication, migration etc); note that the cost, p_j, in (3), is independent of the mechanism k, which means that all reconfiguration mechanisms are considered equal as long as they provide the same final capacity. However, this formulation becomes useful in capturing the transition cost as described below.

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 infrastructure cost, and thus Equation (3) changes to:

$$\min \sum_{i=1}^{N}\sum_{j=1}^{M}\sum_{k=1}^{T} m_{ijk} x_{ijk}. \tag{6}$$

Here m_{ijk} is the cost of transforming server i to server-type j using mechanism k. This cost is estimated using the mechanism-models mentioned in section III-B that capture the overhead of replication, live migration etc[4]. Like before, $x_{ijk} \in \{0, 1\}$ indicate whether the final solution will employ technique k to transition server i to server type j.

Although for a small problem (with nodes less than 10) a perfect solution can be obtained by solving the above formed ILP, as the size of the problem increases finding the optimal solution becomes hard. We have implemented a greedy-type heuristic with a worst case bound of 2 for an approximate solution of the above ILP [5]. The basic idea of the heuristic is to sort $x_{i,j,k}$ in increasing order of p_j/C_j and then find the smallest list of $x_{i,j,k}$'s which satisfy Eq. (4). 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. (5).

IV. Kingfisher System Implementation

We have implemented a prototype of Kingfisher, 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. 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, create/terminate instances on Amazon's EC2, and to reconfigure applications with more or less capacity. We use the XML-RPC APIs exposed by OpenNebula deploy, terminate, or reconfigure servers allocated to an application.

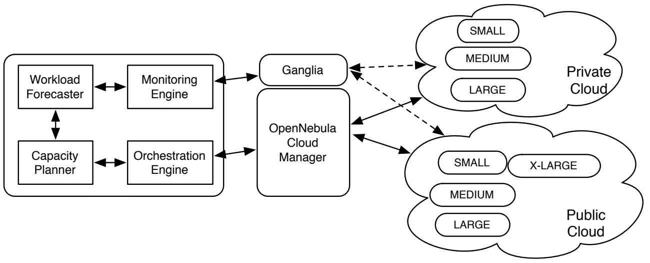


Figure 1. Architecture of our Kingfisher prototype

The architecture of Kingfisher and its relationship to the cloud orchestration framework is shown in Figure 1. We briefly describe below the key components of our architcture, the details are given in [6].

Monitoring engine: Our monitoring engine can track the application-workload and system resources. The monitoring data is stored in a round-robin database. We have implemented our monitoring engine by enhancing Ganglia [7]. Each VM image is pre-configured with the reporting agent; thus, when new virtual machines are dynamically deployed, the Ganglia server automatically recognizes new servers and begins to monitor them without the need for any additional configuration. In scenarios where the cloud platform provides monitoring capabilities (e.g., Amazon EC2 CloudWatch), our monitoring engine can directly query the cloud platform APIs, rather than Ganglia databases, to obtain these metrics.

Workload Forecasting: The workload forecasting component in Kingfisher uses the workload statistics gathered by the monitoring engine to derive estimates of future workloads. We use the open-source R statistical package to implement workload forecasting. In our experiments (in Section V), we focus on evaluating the cost benefits of Kingfisher, hence we assume a perfectly accurate forecaster but any other forecaster can be seamlessly used in its place.

Capacity planner: The capacity planner is the heart of Kingfisher's provisioning engine. It implements our ILP-based algorithm for optimizing the infrastructure cost for

[4] Using the model of mechanisms described in section III-B, we pre-compute a matrix, say $M' = [m'_{ijk}]_{i,j=1...M;k=1...K}$, which represents the cost of migration from server-type i to server-type j using mechanism k. We use M' to compute $m_{i,j,k}$ of (6)

an application or the transition cost of moving to a new configuration. We employ an *lpsolve*, an open-source LP solver that is invoked via a JNI interface from Kingfisher.

Our ILP-based planner requires several pieces of input before it can begin computing cost-optimized configuration for an applications. First, the various types of servers supported by the cloud platform and their infrastructure prices need to be specified. Second, all provisioning mechanisms supported by the cloud platform (e.g., migration, replication etc) must be specified, and a model for estimating the cost/overhead of each mechanism must also be specified. Finally, the empirically derived application capacities for each server hardware type must be specified.

Given these configuration parameters, Kingfisher's planner can be invoked by specifying (i) the tier-specific peak request rate λ for which capacity must be provisioned, (ii) the current configuration for the application, which can be empty if this is the initial deployment of the application, and (iii) the optimization objective, which can be infrastructure cost or transition cost.

Orchestration engine: Once an initial or new configuration has been computed, Kingfisher's orchestration engine instantiates the configuration using the transition plan. This component uses the interfaces exposed by the cloud management platform to resize VMs, startup new instances, or migrate existing VMs. The orchestration engine merely specifies the server type to use (e.g., small, medium, large) for each configuration step, and leaves the problem of placement of these VMs onto physical servers to the cloud manager. Thus, the management platform (OpenNebula or EC2) is assumed to track which physical servers are available to create a VM of the desired type for the application. Migrations were implemented by the VM-manipulation capabilities provided by the underlying hypervisor or by EC2.

V. Experimental Study for Elasticity: Methodology and Setup

In our experimental investigation for cost-aware elasticity, we consider two environments: (i) Private Cloud - a setup based on our prototype design in a lab setting, and (ii) Public Cloud - conducting our study on Amazon EC2 with some adaptation of our prototype. 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 infrastructure 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.

A. Cost-aware elasticity mechanisms

Cost-aware vs Cost-oblivious with Replication: First, we consider replication-only as the method for supporting elasticity - the typical method that is available in public clouds to support 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 migration in addition to replication as the means for supporting elasticity to investigate benefit from such additional mechanisms beyond base level replication based elasticity[5]. 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.

B. Experimental Testbed and Workload

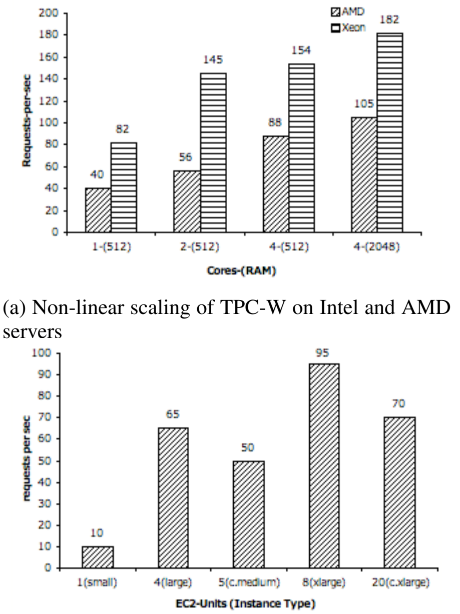
For the private cloud, leveraging the prototype we discussed earlier, we use a laboratory-based cloud system built on virtualized Xen/Linux-based cluster, while our evaluation on the public cloud uses Amazon's EC2. We use the java implementation of TPCW [8] for our experiments. TPC-W is a multi-tier web benchmark that represents an e-commerce web application comprising of a Tomcat application tier and a mysql database tier. The workload used to trigger the provisioning the algorithm was browsing mix of the TPC-W specification; that was generated using TPC-W clients. We have tested each approach on two types of workload patterns: 1) smoothly increasing workload (*small-jump* workload) 2) Sharply increasing workload (*large-jump* workload).

C. Profiling Server Capacities

Earlier, we have argued that real-world applications will not scale linearly with the number of cores due to software artifacts and differences in processor hardware across different systems. As our decision algorithms will have to determine the amount of resource required to meet the desired application level performance, we resort to empirical profiling to determine the application's capacity on each server type.

We configured TPC-W with both tiers in a single VM, and ran this VM on various server instances of both private and public cloud. In the case of public cloud we used the following EC2 server instances: m1.small (S), c1.medium

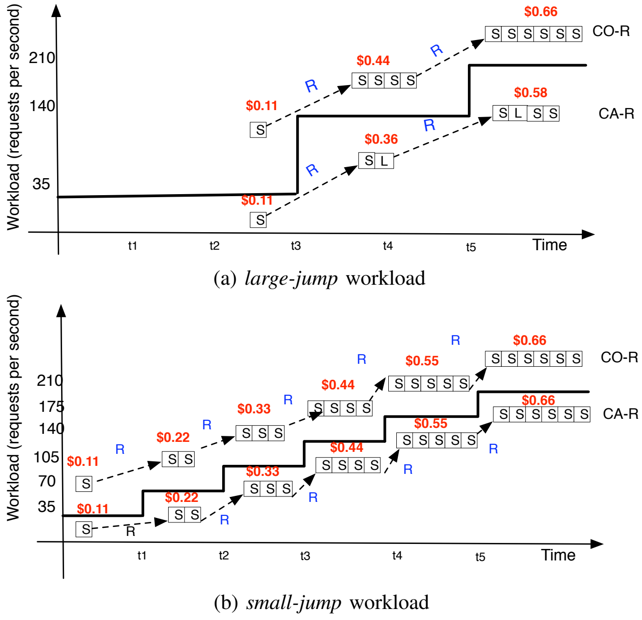
[5] We implemented the migration of a server-instance to another instance using the *live-migration*, *vcpu-set* and *mem-set* facilities of Xen to perform migration. *Live-migration* migrates a virtual machine (server-instance) to a new host-machine (which has more CPU and MEM), while *vcpu-set* and *mem-set* change the number of virtual-cpus and memory of the virtual-machine.



(a) Non-linear scaling of TPC-W on Intel and AMD servers

(b) Non-linear scaling behavior of TPC-W on EC2-instances

Figure 2. Profiling server instances for private and public cloud



(a) *large-jump* workload

(b) *small-jump* workload

Figure 3. Cost-aware versus cost-oblivious provisioning

(M), m1.Large (L), c1.xlarge (XL) and m1.xlarge (XLM); these instances have 1, 4, 8, 5 and 20 EC2 compute units (ECUs), respectively. On the private cloud, it was not possible to have instances equivalent to those of public cloud, nonetheless, we created 1, 2 and 4 core systems; we refer to single-core system as "small" dual-core as "medium" while 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. 2a plots the empirically derived capacities for various multi-core configurations on our Intel and AMD systems on our *private cloud*. It is quite apparent that server configurations on each processor have a very different capacity and in both cases they scale non-linearly. Fig. 2b plots the derived capacities for various EC2-instances.

VI. Evaluation on a Private Cloud

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 (64bit kernel). Our platform is assumed to support small and large servers, comprising 1, 2 and 4 cores, respectively. These are constructed by deploying a Xen VM on the above servers and dedicating the corresponding number of cores to the VM (by pinning the VM's VCPUs to the cores).

We created a virtual appliance of TPC-W on CentOS 5.2. We have used 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.

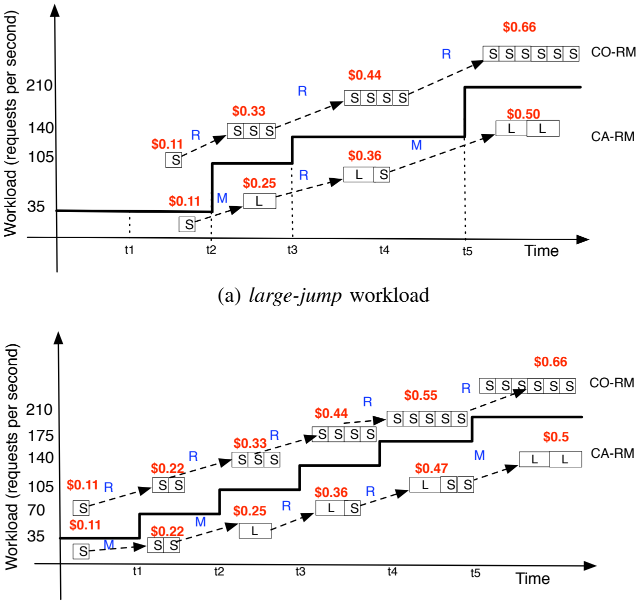
A. Cost-aware versus Cost-oblivious Provisioning

We first compare the cost-aware approach to a cost-oblivious approach (which ignores infrastructure 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 infrastructure 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 infrastructure cost for the same provisioned capacity.

If the workload increases more steadily, as shown in Fig. 3b, both approaches choose *identical* configurations, i.e., an increasing number of small servers. 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.

B. Benefits of adding Migration mechanism

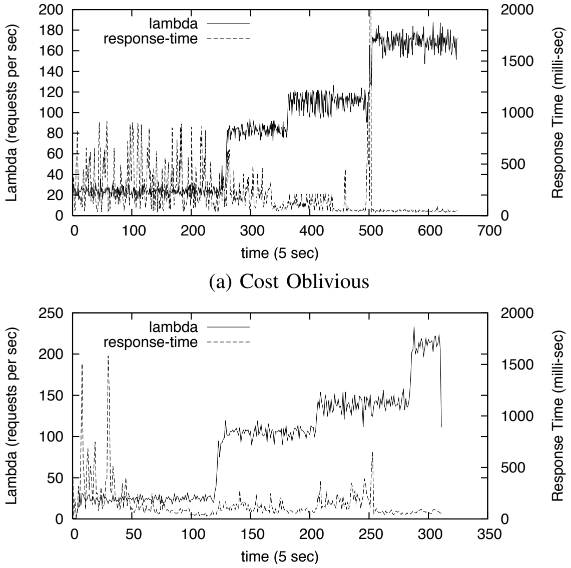
We next consider the benefit of the cost-aware approach compared to cost-oblivious when migration is added as an additional elasticity mechanism to allow relocation of an



(a) *large-jump* workload

(b) *small-jump* workload

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



(a) Cost Oblivious

(b) Cost Aware

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

application to a more cost-effective server configuration. 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 infrastructure cost. Figure 4a 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 4b, the cost-aware algorithm shows a similar benefit over cost-oblivious. Recall that with replication alone, the cost-aware approach produced an equivalent solution as cost-oblivious for the slowly increasing workload – in this case, by adding the migration mechanism, cost-aware provisioning is able to improve the infrastructure cost by 24%. Though the cost-oblivious approach uses both migration and replication as well, its choices are frequently more expensive than those of the cost-aware approach.

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 elasticity, 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.

C. Transition cost-aware Provisioning

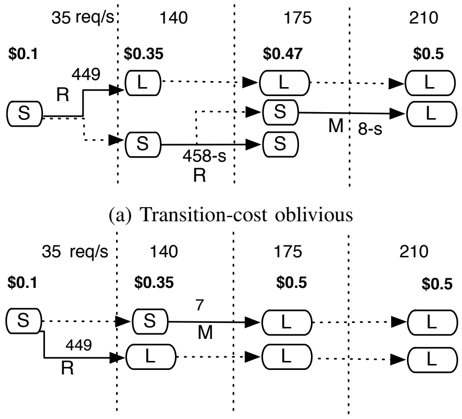
Our experiments thus far have focused on optimizing infrastructure cost and have ignored the overhead of transitioning the application deployment from one configuration to another. By making elasticity decisions based on the time overhead of various options, Kingfisher's transition cost-aware approach can quickly provision additional capacity in the cloud when the workload surges suddenly. However, by focusing on rapid reconfiguration, transition cost-aware provisioning may not produce the minimal infrastructure cost.

To demonstrate the benefits of our approach, we increased the TPC-W application workload in a series of large steps. At each step, we invoked Kingfisher's transition cost-aware provisioning and compared the decisions made by this approach with its infrastructure cost-aware provisioning method (i.e., which ignores the transition cost when making decisions). We assumed a cloud platform with two server types, small (S) and large (L), with infrastructure costs of $0.11 and $0.25 per hour, respectively (as in Table-I).

Figure 6 shows that the transition and infrastructure costs resulting from the chosen configuration after each workload step (i.e., from 35 req/s to 175 req/s). The transition cost-aware approach is able to pick lower transition time configurations, while the other approach opts for a lower infrastructure cost configuration but takes an order of magnitude more time. For example, when the workload increases from 140 to 175 req/s, the transition cost-oblivious approach performs a replication requiring 458s, while transition cost-aware opts for migration to a large server which requires 7 seconds, but results in a slightly higher infrastructure cost. Over the course of the experiment, the figure shows that transition cost-oblivious chooses replication twice, while transition cost-aware replicates once, resulting in a much quicker response at the expense of some added infrastructure cost.

Figure 7(a) and (b) show the applied workload on the TPC-W application and the average response time as the workload increases. Between 150 and 200 seconds the workload increases to 175 req/s and, after a small spike in response time corresponding to the migration to a large server, the transition cost-aware solution settles to the target response time. In contrast, in (a) the transition cost-oblivious approach take significantly longer to reach the desired response time as the replication operation proceeds.

The experiment demonstrates that since copying memory state during live migration incurs lower latencies than copying disk images during replication live migration may be preferred, whenever feasible, to reduce transition costs. However, migration is not always feasible (e.g., if the application is already on the largest possible server) and replication may be needed in such cases.



(a) Transition-cost oblivious

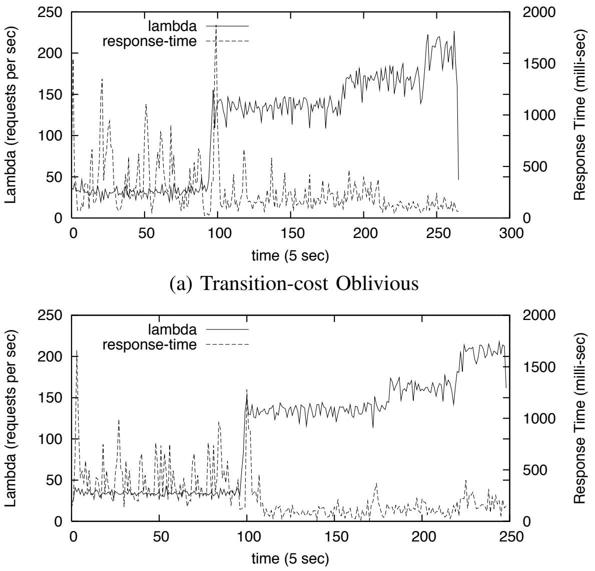
(b) Transition-cost aware

Figure 6. Comparison of a transition-cost aware system with a transition-cost oblivious system. Solid lines denote a configuration change, while dotted lines indicate no change.

D. Impact of the Pricing Model

Prior experiments have assumed a convex pricing model where the cost-per-core decreases as the number of cores increases. Since our ILP can handle arbitrary pricing models, we demonstrate how different pricing models can impact the choice of the configuration.

We consider the TPC-W application and wish to deploy it on a cloud platform with different initial capacities (workload, and corresponding required capacity, is increased from λ to 6λ). We assume that the cloud supports three types of servers, small, medium and large. For comparison, we also show the results of the cost-oblivious approach, which always chooses small servers regardless of the pricing model. First, we assume a convex pricing model, which resembles those employed in current clouds. In this case, larger servers have lower cost-per-core, causing our approach to prefer medium and large servers over small ones, when possible. Next, we employ concave pricing model, where the cost-per-core increases for larger systems. In this case, since the small server has the cheapest price per code, our cost-aware approach uses only small servers to provision capacity, effectively choosing the same configuration as cost-oblivious. Finally, we choose an arbitrary pricing model, where the medium server is the cheapest, and the large server is the next cheapest on a cost-per-core basis. This causes our approach to prefer medium servers when possible and occasionally some large server instances.



(a) Transition-cost Oblivious

(b) Transition-cost Aware

Figure 7. Workload and Response Times of a transition-cost aware system with a transition-cost oblivious system.

Provisioning Algorithm	λ	3λ	4λ	5λ	6λ
Convex pricing model (S=0.11,M=0.15,L=0.25)					
Cost Aware	S	2M	M,L	2L	4M
Cost Oblivious	S	3S	4S	5S	6S
Concave pricing model (S=0.11,M=0.24,L=0.5)					
Cost Aware	S	3S	4S	5S	6S
Arbitrary pricing model (S=0.11,M=0.15,L=0.44)					
Cost Aware	S	2M	S,2M	2S,2M	4M

Table II
PROVISIONING WITH DIFFERENT PRICING MODELS

VII. Evaluation on Public Cloud: Amazon EC2

In this section we conduct our experimental evaluation on public cloud using Amazon EC2. We compare cost-aware with cost-oblivious elasticity methods for both infrastructure and transition costs. We need careful examination of the migration support in Amazon EC2 as the steps and the associated costs are quite different based on whether image provisioning is based on EBS or instance-store based [9]. We compare three different scenarios for transition cost-aware approach - these differ in the way the transition cost is accounted as well as the storage is used for image provisioning.

Note that Amazon EC2 supports eight EC2-instance types [9]. We have used 5 of these server-types of EC2, namely S, M, L, XL and XLC, each of which are profiled offline and the results are shown in Fig. 2b. EC2 allows creation of instances of each of these server types; these instances can be created either from instance-store or from EBS-volume snapshots, where an EBS-volume is a persistent storage. Amazon offers snapshoting capability on these EBS-volumes and these snapshots can be used to create new EC2-instances.

A. Determining Transition Costs in EC2

Kingfisher's transition-aware provisioning method needs to accurately account for the overheads of different replication / migration mechanisms available in EC2. We conducted a sequence of experiments to empirically determine these costs that we require for Kingfisher provisioning step.

We determine the transition costs for both EBS and instance-store based provisioning approach as the associated process and costs are quite different. EC2 provides two mechanisms from starting up a new new replica: (1) using an EBS-volume image (2) using the instance-store. Unlike private cloud, which supports live migration, the EC2 system supports only *shutdown-and-migrate* on EBS-volume based instances, while on instance-store based EC2-instances it only supports *replication*. Nevertheless, it is possible to simulate a migrate operation for instance-store based instances (i.e. those created using instance store) in two different ways. If the application does not maintain any state on its local disk (e.g., if the persistent state is stored on the S3 and on a separate EBS-volume, which is mounted on EC2-instance during instance-creation time), then we can emulate migration by starting a new instance on a larger server (via replication) and simply shutting down the old server and attaching the disk state to the new server (called *replicate-shutdown*). In contrast, if the state of the local disk needs to be migrated as well, then a *shutdown-copy-migrate* operation can be performed, where an application is shutdown, a machine image of its disk state is created and uploaded to S3, and a new replica is started with this image; on EBS-snapshot based instances, one can stop the instance and restart it as a different EC2-instance; we call this as *stop-and-start* operation.

In order to capture the cost of each of the provisioning operation, we break down the each operation into its component steps and capture the cost of each of the component steps. The *shutdown-copy-migrate* option, in a non-EBS volume instance involves following five steps 1.) copy the complete disk-image 2.) compress it 3.) uploading it onto S3 4.) register it as an AMI[6] 5.) create an instance using this new AMI. Table III(a) shows the time taken to complete each component steps for different size-images. Note that the total time is linearly varying with the size of compressed image. Similarly for EBS, there are three distinct steps. Table III(b) depicts the time it takes to take a snapshot of volume which contains data which cannot be compressed any-further. The time to take the snapshot of an EBS volume can also be modeled as a linear function of size of compressed image size. As shown in IIIc, the time it takes to boot an instance from EBS-snapshot is nearly constant– our measured average value is 85 sec The average instance registration time is 7 sec. The *replicate-shutdown* option incurs a similar overhead as that of a pure replicate operation. In our experiments we have used the time to be 800 sec (since our instance gets compressed to 3GB). Finally, the *stop-and-start* operation is estimated to have mean overhead of 65 sec.

Volume Size (GB)	Compressed Image (GB)	Snapshot	upload time(s)	boot time (s)
10	1.22	675	175	190
10	1.60	710	210	246.5
10	2.34	927	310	345
10	2.99	1160	314	407.1
10	3.08	1308	435	424
10	3.54	1466	490	494.3

(a) Time measurements of steps involved in shutdown-copy-migrate operation

Volume Size (GB)	Used Space	Compressed Image (GB)	Zone	Snapshot time
10	2	2	us-east-1a	491
10	4	4	us-east-1a	915
10	6	6	us-east-1a	2064
10	8	8	us-east-1b	2596

(b) Time Measurements of taking snapshot of an EBS volume

Volume Size (GB)	Used-up space	Zone	Startup Time (s)
10	5	us-east-1a	82.7
10	6	us-east-1a	84
10	7	us-east-1a	82
10	8	us-east-1b	85.7
10	9	us-east-1a	88

(c) Time measurements of start-up time of an image from EBS-volume

Policy	λ	2λ	3λ	6λ
CO-RM	4S(.34)	S,L (.425)	2L (.68)	3L,2S (1.19)
CA-RM	4S (.34)	2M (.34)	S,2M (.425)	4M,S (.765)
TA-RM-1	4S (.34)	2S,M (.34)	S,2M (.425)	XL,L,M (1.19)
TA-RM-2	4S (.34)	2S,M (.34)	S,2M (.425)	S,4M (0.765)
TA-RM-3	4S (.34)	2S,M (.34)	S,2M (.425)	3M,L (0.85)

(d) Provisioning with different methods ($\lambda = 35$). Choice of provisioning mechanism for each transition, i.e. from $\lambda \rightarrow 2\lambda, 2\lambda \rightarrow 3\lambda$ and $3\lambda \rightarrow 6\lambda$, are described in section VII-C

Table III
MEASUREMENTS AND PROVISIONING ON EC2

[6] An Amazon Machine Image (AMI) is a virtual machine image which is used by EC2 to create server instances

B. Infrastructure-cost aware Provisioning

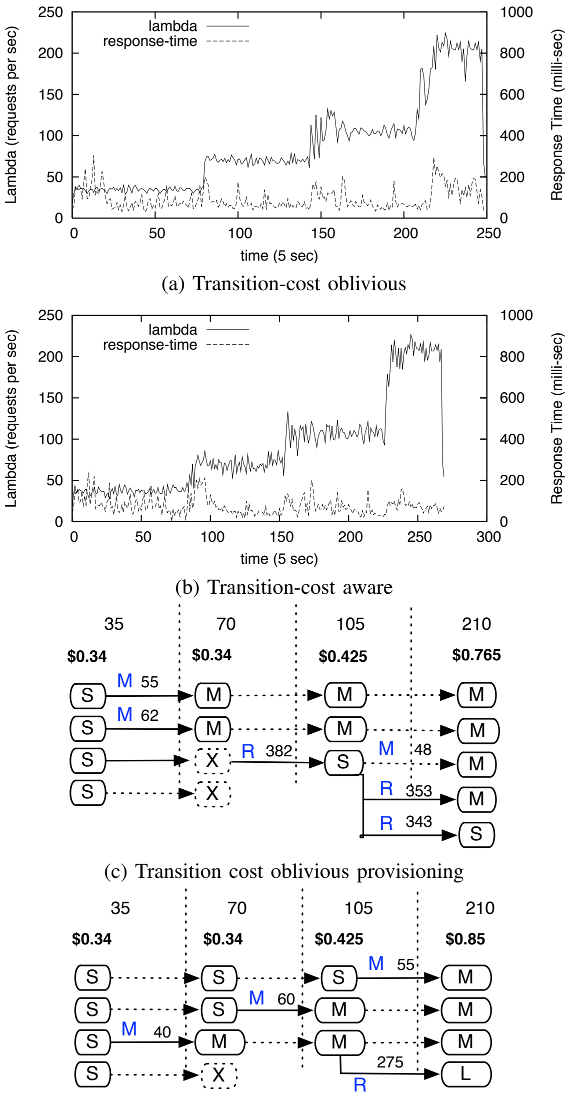
To evaluate the efficacy of Kingfisher in taking infrastructure and transition costs into account, we repeated our TPC-W experiment on the public EC2 cloud. We assume an initial configuration of four small servers serving an initial workload of $\lambda = 35$. The infrastructure cost of servers is summarized in Table I and transition cost is discussed above. Like before we varied the workload in steps and Table III(d) depicts the configurations generated by the cost-oblivious and Kingfisher's cost-aware methods. The cost-aware (CA-RM) method is able to provision the same capacities at 35% lower cost.

C. Transition-cost aware Provisioning

Using the empirically determined transition costs, we next evaluate transition cost-aware elastic provisioning. We consider three transition cost scenarios based on usage pattern and constraints in EC2: (i) TA-RM-1, which only takes into account the number of transitions and cost of each transition and also the infrastructure cost of final configuration; (ii) TA-RM-2: that considers transition costs and final infrastructure costs for non-EBS instances in EC2, and (iii) TA-RM-3 that distinguishes between 32-bit small EC2 instances, and 64-bit larger EC2 instances, and assumes that 32-bit and 64-bit applications are not mixed across the corresponding server types.

As shown in Table III(d), when workload jumps to 2λ, TA-RM-* chooses to perform only one *stop-and-start* operation as opposed to two chosen by CA-RM; notice that both configurations have the same dollar cost however CA-RM policy tries to maximize capacity, while TA-RM-* schemes minimize the number of reconfigurations. When the workload increases from 2λ to 3λ, the CA-RM method resorts to *replication*, while the TA-RM-* chooses the faster *stop-and-start* provisioning. In the final step, CA-RM chooses to perform two replications, however, TA-RM-1 initiates two *stop-and-start* operations for faster provisioning. Since TA-RM-2 provisions non-EBS instances, it chose the faster *replication* option (over the slower *shutdown-copy-migrate*). TA-RM-3, on the other-hand, performs a *stop-and-start* from S to M instances and then initiates another *replication*.

Figure 8 show the result of provisioning experiment conducted using Kingfisher for TA-RM-3 scheme. Figure 8a and Figure 8b show the response-times of the of the corresponding configurations, indicating the responsiveness of the system using the end-to-end response time of the configuration under workload. The benefit of transition-cost aware approach is apparent from Figure 8c,8d: in the first and last step it approximately takes the same time[7], however in the second jump the transition-cost aware system achieves the new configuration in 60 sec as opposed to 382 sec.



(a) Transition-cost oblivious

(b) Transition-cost aware

(c) Transition cost oblivious provisioning

(d) Transition cost-aware provisioning

Figure 8. Comparison of a transition-cost aware system with a transition-cost oblivious system.

VIII. Related Work

Our work focuses on optimizing the use of elasticity mechanisms and is applicable in commercial cloud service offerings (exemplified by Amazon EC2 and others) and cluster management systems such as OpenNebula or Eucalyptus. In particular, this study is the first work to propose cost-aware provisioning in a cloud, along with algorithms to optimize how additional mechanisms beyond replication can be leveraged to support elasticity.

There is a significant amount of related work, however, in the area of dynamic capacity provisioning in data centers, grids, or compute clusters, starting with earlier work such as [10] and [11]. Much of this work is platform-centric, while our work considers a customer-centric view

[7] The large variation in similar operations is because the copy operation is dependent on the load on the backend network and disk systems of EC2

of provisioning and resource optimization. Other work has considered migration as a means of dynamic provisioning [12], while we consider replication with different types of migrations and assign cost to each of them. There is also an extensive body of work on dynamic provisioning of web applications using analytic models [13], [14], [15], [16]. Classical feedback control theory has also been used to model the bottleneck tier for providing performance guarantees for web applications [17], [18]. The approach in [18] formulates the application tier server provisioning as a profit maximization problem and models application servers as M/G/1/PS queueing systems. The work in [3] provides a model-driven approach for adapting resources for a multi-tier application. Finally, machine learning techniques have also been used for provisioning, such as the k-nearest neighbor approach to provision the database tier [19].

In contrast to these efforts, our work automates the process of characterizing the workload mix and uses empirical models as a basis for provisioning system capacity. Further, while we employ analytic models of infrastructure and transition costs, our approach involves full prototype implementation and experiments on an actual Linux cluster.

IX. 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 infrastructure cost of provisioning a certain capacity. We proposed a new cost-aware provisioning approach for cloud applications that can optimize either the infrastructure 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. We also presented a case study of how our approach can be employed in a public cloud such as Amazon EC2. In future we plan to extend kingfisher by integrating it with systems which employ queuing theory based model for capacity estimation for provisioning on cloud.

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] J. Hellerstein, F. Zhang, and P. Shahabuddin, "An Approach to Predictive Detection for Service Management," in *Proceedings of the IEEE Intl. Conf. on SNM*, 1999.

[3] 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.

[4] 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.

[5] 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.

[6] U. Sharma, P. Shenoy, S. Sahu, and S. Anees, "Kingfisher: A System for Elastic Cost-aware Provisioning in the Cloud," Dept. of CS, UMASS, Tech. Rep. UM-CS-2010-005, May 2010.

[7] "Ganglia monitoring system," http://ganglia.sourceforge.net/.

[8] TPCW, "Java implementation," Website, http://tpcw.deadpixel.de.

[9] A. EC2, "Amazon ec2," Website, http://aws.amazon.com/ec2.

[10] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-based scalable network services," in *SOSP'97*, December 1997, pp. 78–91.

[11] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, "Managing energy and server resources in hosting centers," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 103–116, 2001.

[12] L. Grit, D. Irwin, , A. Yumerefendi, and J. Chase, "Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration," in *VTDC*, November 2006.

[13] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Adaptive and Autonomous Systems (TAAS), Vol. 3, No. 1*, pp. 1–39, March 2008.

[14] C. Stewart and K. Shen, "Performance Modeling and System Management for Multi-component Online Services," in *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, May 2005. [Online]. Available: http://www.usenix.org/event/nsdi05/tech/full_papers/stewart/stewart.pdf

[15] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, Washington, DC, USA, 2007.

[16] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, "1000 islands: Integrated capacity and workload management for the next generation data center," in *ICAC*, 2008, pp. 172–181.

[17] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach," *IEEE Trans on PDS*, vol. 13, no. 1, pp. 80–96, 2002. [Online]. Available: citeseer.csail.mit.edu/abdelzaher01performance.html

[18] D. Villela, P. Pradhan, and D. Rubenstein, "Provisioning Servers in the Application Tier for E-commerce Systems," in *Proceedings of the 12th IWQoS*, June 2004.

[19] J. Chen, G. Soundararajan, and C. Amza, "Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers," in *ICAC'06*, Jun. 2006, pp. 231–242.