Kingfisher: A System for Elastic Cost-aware Provisioning in the Cloud

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

Sambit Sahu, Anees Shaikh
IBM Research
{sambits,aashaikh}@us.ibm.com

January 26, 2010

Abstract

Cloud computing platforms allow application providers to rent server capacity to run hosted applications and to dynamically vary the rented capacity to match demand. Today's cloud platforms offer a plethora of different server configurations for rent and price them differently on a cost-per-core basis. Furthermore, cloud platforms support different replication and migration mechanisms to support elastic provisioning of servers. In this paper, we present Kingfisher a cost-aware provisioning system 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 system exploits both replication and migration to dynamically provision capacity and uses an integer linear program formulation to optimize cost. We have implemented a prototype of our Kingfisher cloud provisioning system and have evaluated its efficacy on a laboratory-based private Xen cloud as well on the public Amazon EC2 cloud. Our experiments demonstrate the efficacy of Kingfisher in elastically provisioning servers within private and public clouds that see varying application workloads. Our results demonstrate the ability of Kingfisher in reducing server rental costs and reconfiguration overheads over prior cost-oblivious approaches.

1 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." Fundamentally, cloud computing enables application providers to allocate resources purely on-demand – on an as-needed basis – and to adjust the amount of resources to match workload demand. 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* capability 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.

The IT consumption model in cloud environments enables flexible and elastic provisioning; that is, it supports a variety of configurations and also provides mechanisms to add or remove compute-capacity. This opens-up the opportunity to efficiently manage ones resources, but, also brings new challenges for application providers. The two key ones being: (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. This is complicated by the fact that these server configurations are typically not priced linearly with server capacity. For instance, a quad-core server is not priced four times the price of four single-core servers. As shown in Table 1, 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. The array of available hardware configurations lead to a number of different ways to configure a typical multi-tier Web application and a careful selection of final configuration can prove cost-effective. Hence, application providers must take the cost into account when determining a choice of the initial server configuration for their applications. The application provider is faced with the second decision 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, 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 cloud pricing models and elasticity mechanisms to select resource configurations and transition strategies that optimize the cost incurred by customers. Our paper makes the following contributions:

Cost-aware provisioning. We present Kingfisher, a system for cost-aware provisioning in the cloud that takes into account the price differentials in the per-core cost of differ-

ent server types to minimize the rental cost of provisioning a certain capacity as well as the transition cost of reconfiguring a deployed application when adding more capacity. We formulate the provisioning problem as an integer linear program (ILP) to account for both rental and transition cost for deriving appropriate elasticity decisions. Our ILP-based approach integrates multiple mechanisms such as replication and migration into Kingfisher and computes both a cost-optimized configuration for the desired capacity as well as plan for transitioning the application from its current setup to its new configuration.

Prototype implementation and experimentation on public and private clouds. We implement a prototype of our Kingfisher cloud provisioning engine, using the OpenNebula cloud toolkit [10], that incorporates our optimizations, and evaluate its efficacy on both a private laboratory-based Xen cloud and the public Amazon EC2 cloud. Our results (i) demonstrate the cost savings that can be realized using our approach over prior cost-oblivious provisioning approaches, (ii) demonstrate the benefits of integrating multiple mechanisms such as migration and replication into a unified approach, and (iii) demonstrate how our transition-aware approach can be employed to quickly provision capacity in scenarios where an application workload surges unexpectedly.

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 *platform-centric*, where the approach attempts to maximize resource utilization from the provider perspective by dynamically allocating a set of servers across hosted applications with varying workload demands (and attempting to statistically multiplex as many applications as possible on the platform). In contrast, the problem articulated in this paper requires a *customer-centric* view, where each customer ("application provider") individually optimizes their capacity usage by choosing the least-cost server configuration that matches their needs. Platform-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 a N-core system equivalent, from a provisioning perspective, to a N systems with single cores. In the cloud context, however, the choice of the configuration matters, since the pricing per core is not uniform. Thus, from a pricing standpoint, a quad-core system is not the same as four single-core systems, even though they may be roughly equivalent from a capacity perspective. Hence, Kingfisher is designed to be *cost-aware*, and takes the rental costs of servers into account – by choosing the least cost configuration that fulfills an application's needs.

Table 1: Cloud server configurations and their prices. For EC2, 1 ECU= 1.2 GHz Xeon or Optron circa 2007.

Amazon EC2 Cloud Platform			
Server size	Configuration	Cost	Cost per core
Small	1 ECU, 1.7GB RAM, 160GB disk	$0.10 / hr	$0.085
Large	4 ECUs, 7.5GB RAM, 850GB disk	$0.4 / hr	$0.34
Med-Fast	5 ECUs, 1.7GB RAM, 350GB disk	$0.8 / hr	$0.17
XLarge	8 ECUs, 15GB RAM, 1.7TB disk	$0.2 / hr	$0.0.68
XLarge-Fast	20 ECUs, 7GB RAM, 1.7TB disk	$ 0.8 /hr	$0.68
NS Cloud Platform			
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
Jumbo	8 core 2.0GHz, 8GB RAM, 1TB disk	$ 0.38 / hr	$0.048
Fast	4 core 3.0 GHz, 4 GB RAM, 600GB disk	$0.53 / hr	$0.133

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, relying solely on replication or migration to provision capacity is restrictive from cost-optimization standpoint. In order to fully exploit the pricing differentials of different cloud server configurations, Kingfisher considers both replication and migration when choosing the best method of transitioning the application to its new configuration.

2 Problem Formulation

In this section, we present the system model assumed in our work and formulate the problem of cost-aware provisioning for the cloud that we address in this paper.

2.1 Cloud Platforms: System Model

Consider a cloud computing platform that offers compute servers to run hosted applications. We assume that the platform offers N heterogeneous server configurations for rent, each with a different rental 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, causing some servers to be cheaper and others more expensive on a per-core basis when compared to uni-core systems. As noted in Table 1, 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.

Customers are assumed to be able to provision servers using a self-service portal, allowing users to request and terminate cloud servers without human intervention. The cloud platform is assumed to have a virtually infinite pool of servers, allows customers to request an arbitrary number of servers for their applications; servers may be requested and terminated at any time and billing is based on the amount of time for which each servers was in use (e.g., based on the number of hours for which each server was used).

We assume that the cloud platform employs virtualization—each physical server is assumed to run a hypervisor that control the allocation of physical resources on the machine. One or more virtual machines may be mapped on to each physical server, and the hypervisor is used to allocate a certain number of CPU cores, memory and disk to each VM. Thus, cloud platform allocates virtual servers of different size to customers, where each virtual server resembles a hardware configuration used in the pricing model.

Because the cloud platform is virtualized, it offers the ability to flexibly provision capacity/resources to a customers. This may be achieved in a number of different ways. In the simplest case, an existing virtual server may be *resized* by increasing (or decreasing) the number of CPU cores and/or memory allocated to it. This can be done on-the-fly by simply modifying the resource allocation of the VM in the underlying hypervisor. Instead of resizing locally on the current server, it is also possible to *migrate* an existing VM to a larger physical server and allocate it more resources on the new server. Depending on the cloud platform, such migration-based resizing can done in a live fashion, while the application is running, or by shutting down the VM, copying the disk state to the new server, and starting it up there. The customer can also provision additional capacity by starting up new virtual servers and *replicating* the application on these new servers. In our work, we assume that the customer is free to use any provisioning mechanism — resizing, migration or replication — that is exposed by the underlying cloud platform. We note that a cloud platform may only expose a subset of these mechanisms (e.g., the EC2 cloud does not presently support live migration) and our approach must take these constraints into account when provisioning resources on behalf of the customer.

2.2 Problem Statement

We are broadly concerned with the problem of a customer who wishes to deploy an application onto cloud servers and vary the capacity allocated to the application to match its workload demand. Unlike some statistical multiplexing approaches that perform application placement and dynamic resource provisioning to *maximize* revenue for the platform, we take a customer-focused cost *minimization* view of the problem. Thus, given certain cloud server configurations and prices, our approach must choose, on behalf of the customer, the least-cost configuration that provisions sufficient capacity to satisfy the application's workload and SLA needs.

We assume that an application is distributed with k interacting components. A multi-tier web application is the most common example of such an architecture. Components may or may not be replicable. If a component is replicable, then the capacity allocated to that component can be varied either by migrating that component to a larger (or smaller) server or by dynamically starting up additional replicas of that component on new server. If a component is not replicable, then its capacity can be varied only by resorting to local resizing or migration to a different hardware configuration. In multi-tier web applications, for instance, the front tiers are often replicable, while the back-end database tier is not; thus either replication or migration can be used to provision capacity to front tiers, while only migration/resizing can be used for the back-end database.

Given these assumptions, the provisioning problem can be stated as follows.

Initial Deployment: Assume an application with k independent components/tiers. Let λ denote the maximum incoming workload for which capacity needs to be provisioned at each tier. Assume that this peak incoming workload of λ request/s imposes a workload of $\lambda_1, \lambda_2, \ldots \lambda_k$ request/s at each tier i.[1] 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 rental 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 (e.g., by picking 8 uni-core servers, 4 dual-core or two quad-cores—all of which provision 8 cores for a tier), the goal is to choose the cheapest configuration that meets the needs of each tier. Thus, the pricing model for different cloud servers will drive the choices made by the provisioning approach.

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 hardware 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 workloads of $\lambda'_1, \lambda'_2, ..\lambda'_k$. Furthermore, we must also specify a *plan* for morphing each tier from its current configuration to the new configuration

[1] In the simplest case where each incoming request triggers a single request at each tier, $\lambda_1 = \lambda_2 = \lambda_k = \lambda$. However, this need not be the case, for instance if caching is employed at tiers, or when a request triggers multiple requests at a downstream tier (e.g., database queries). For generality, we assume that λ_i's can be different.

using mechanisms such as resizing, migration or replication. We are interested in minimizing two types of costs: (i) the *rental 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 rental 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 rental 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 rental 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.

3 Cost-aware Provisioning In the Cloud

Any dynamic provisioning algorithm involves three steps: (i) determining *when* to invoke the provisioning algorithm, (ii) determining *how much* capacity to provision, and (iii) determining *how* to choose a configuration that minimizes rental or transition cost. We discuss each issue in turn.

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 . In contrast a reactive approach uses thresholds on resource utilization or on SLA violations to trigger the need for additional capacity. The issue of proactive or reactive invocation is orthogonal to that of cost-aware provisioning, and hence, we choose a simple threshold-driven reactive approach in this paper.

How much to Provision? The problem of how much capacity to provision involves estimating how much peak workload will be seen by the application in the future. Workload forecasting techniques can be employed to derive these estimates. Specifically, the rate of workload growth over time can be used to estimate the overall workload volume that will be seen in the future. In this paper, we employ a simple ARIMA time-series predictor to capture workload trends and estimate the future workload; however, any workload forecasting technique is compatible with our provisioning approach that we present next.

3.1 Rental 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 rental cost; the provisioned servers must have the collective capacity to service at least λ_i request/s at tier i while meeting the 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. Determining 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 and its workload mix. While faster processors and/or more cores allow a server to service higher request rates, the increase in capacity is not linear with the number of cores or processor speed. For example, a server with four cores will typically not be able to service four times the requests serviced by a single core system. This non linear scaling occurs due to software artifacts (e.g., threading, locking, configuration thresholds) in the application and also due to I/O activity. Further differences in the processor families across server configuration can also yield non-linear increases in capacity. For example, the small, medium and large server configurations depicted in Table 1 all have different processor speeds, and consequently, will not scale linearly with the number of cores.

There are two possible approaches for estimating the maximum workload that can be serviced by a particular server type: queuing-based and empirical. In the queuing approach, a server is modeled as a queuing system (e.g., a G/G/1 system) and queuing theoretic results are used to derive a relationship between the request rate, service times of requests, and the response time SLA. For instance, for a G/G/1 single core server, the Kingman's approximation [9] yields such a relationship ($C < \left[\bar{s} + \frac{{\sigma_a}^2 + {\sigma_s}^2}{2(r/3 - \bar{s})}\right]^{-1}$), where C denotes the maximum request rate, $\bar{s}$ denotes the mean service time of a request; σ_a^2 and σ_s^2 denote the variance in the inter-arrival times and the service time, and r denotes the desired response time SLA of the tier. The advantage of this approach is that the capacities for different server types can be derived by measuring some simple workload characteristics. The limitation though is that queuing theory 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 an alternate empirical approach that estimates server capacity by actually running the application on different hardware configurations and 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 empir-

ical 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 that the simple measurements required by the queuing approach. We note, however, that a system such as JustRunIt [17] 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 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 when determine how many servers of each type are needed for the application.

Step 2. Determining Server Configurations.

Consider a cloud platform with M different types servers (e.g., small, medium, large). Let C_i and p_i denote that capacity (maximum request rate) and the rental cost of server type i. 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

$$\text{minimize} \sum_{i=1}^{M} n_i p_i \;\; s.t. \sum_{i=1}^{M} n_i C_i \geq \lambda \tag{1}$$

where n_i denotes the number of servers of each type that is chosen. The goal of our approach is to determine $(n_1, n_2, \ldots, n_M)$ — which tells the application provider how many servers of each type should be chosen for the application tier.

Before presenting our provisioning algorithm, we present the intuition for how the rental cost can be minimized. We note that there are many different ways to provision for the desired capacity λ. For example, if the application scaled perfectly with cores, and if we wanted to provision 8 cores, we could do so by using 8 uni-core (small) servers, 4 dual-core (medium), 2 quad-core (large), or one oct-core (jumbo) server. While all of these configurations are identical in terms of capacity, the cost of renting these servers is different; hence, the cost-optimal solution is to choose the server type that yields that highest capacity per unit rental cost. Since cloud providers have currently priced their offering using a convex pricing function—where the cost per core falls for larger systems—the cheapest solution in this simple example is to pick an oct-core system.

In the more general case, applications don't scale linearly with capacity, and the pricing model can be arbitrary. We have assumed that C_i is the empirically measured capacity of server type i and p_i is its rental cost. The ratio $\frac{C_i}{p_i}$ yields the capacity per unit rental cost for server type i— larger ratios indicate that the server yields a higher capacity for a lower rental cost. Thus, a cost-aware provisioning approach should prefer server configurations with the highest $\frac{C_i}{p_i}$ values.

A simple greedy approach then is to order cloud servers by their $\frac{C_i}{p_i}$ ratios, and then choose as many servers with the highest $\frac{C_i}{p_i}$ as possible and then provision in the "residual" capacity with the next best server with a capacity that is at least equal to the residual capacity and so on. To illustrate, suppose the cloud platform supported quad-, dual- and uni-core systems and used a convex pricing model such as indicated in Table 1. If we then wanted to provision 7 cores for an application, the approach would first pick a quad-core system, and use one dual and one uni-core system to fill the residual capacity, yielding 7 cores in total. This is the basic intuition behind our approach. We note, however, that this simple greedy approach is not optimal. For example, in the above example, it is cheaper (and optimal) to pick two quad-core servers, a solution that over-provisions by one core but is still cheaper than renting one quad, one dual and one uni-core server.

Our cost-aware algorithm uses an integer linear program (ILP) that uses the above $\frac{C_i}{p_i}$ intuition when computing a solution. The ILP formulation can consider alternatives, in addition to those chosen by the above greedy approach, when determining a solution, and thereby account for scenarios such as those in the above example (where over-provisioning slightly yields a cheaper solution).

ILP: Our cost-aware provisioning problem 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_i denote the rental cost for server type i and let C_i 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 rental cost is

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} p_{(j)} x_{ijk} \tag{2}$$

subject to the constraints

$$\sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} x_{ijk} C_j \geq \lambda \tag{3}$$

$$\sum_{j=1}^{M} \sum_{k=1}^{T} x_{ijk} = 1, \forall i \tag{4}$$

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

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 type 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 cho-

sen and also specifies a plan for transitioning from the current configuration i to the new server type j using method k (replicate. migrate etc). If this 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).

Since ILP is a NP-hard problem, we use the "standard approach" of approximating it as a linear program and solving the resulting LP using a linear programming library. A linear program will yield non-integral values of x_{ijk}, which we then round up or down to the closest integer (here 0 or 1), while satisfying the ILP constraints in Eq. 4 and 5.

3.2 Transition Cost-aware Provisioning

Our rental cost-aware provisioning algorithm computes both a new configuration for the application as well as a plan to transition the application from its current to the new configuration. A certain latency will be incurred for this transition—due to the large amounts of data that must be copied during a replication or a migration. Depending on the size of the application's disk and/or memory state, such transitions can take on the order of tens of seconds to tens of minutes. In many cases, this latency is tolerable if the transition to a higher capacity configuration is implemented during a planned maintenance down-time for the application. However, there can be scenarios where this transition latency incurred to move the application to the new configuration is important, especially when additional capacity needs to be added to the application quickly. The application provider may even be willing to pay a higher rental cost if the transition latency can be minimized—by choosing a configuration that minimizes the transition latency rather than the rental cost.

Our transition cost-aware provisioning method is designed to address such a scenario. The goal is to determine a configuration that can service the workload surge and to transition to that configuration as quickly as possible. Thus a configuration that has the least transition latency rather than the least rental cost must be computed.

To do so, our 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 (e.g., if it is faster to live migrate a particular application than to start a new replica, then configuration that involves migration from the existing setup is chosen). The overhead of these mechanisms can be estimated 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). Since this is akin to using OS system calls to modify resource allocations, 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 to permit such 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 of these operations is dominated by the overhead of copying the disk image, which can be several gigabytes in size, to the local disk of the new server. 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 the 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 and need not be copied. 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 (private clouds such as those constructed using Eucalyptus and OpenNebula do support live migration, however). 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. Unlike live migration, such an approach only migrates the disk state of an application and does not migrate memory state; it also incurs an application down-time. However, it does enable an application to be moved to a larger system with its disk state intact. 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 $2\frac{D}{r} + b$.

From the above discussion, it is clear that, except for local resizing, the latency of a transition mechanism is directly proportional to either the disk state or the memory state of the virtual server. The transition-aware approach attempts to minimize this overhead by preferring mechanisms that incur the lower data copying overheads (and hence, lower latencies). Like before, the optimization problem can be formu-

[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

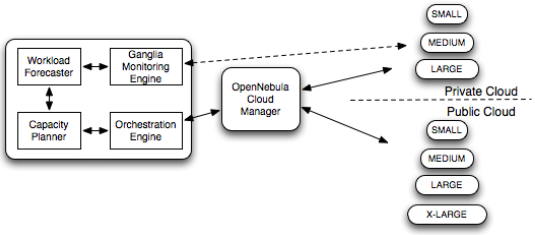


Figure 1: Architecture of our Kingfisher prototype

lated as an integer linear program. The ILP is identical to the previous one except for the optimization criteria which must minimize the transition cost rather than rental cost, and can be stated as follows:

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

subject to the same constraints as before. 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} are integer variable of the ILP that take values zero or one, and indicate whether the final solution will employ technique k to transition server i to server j.

4 Kingfisher Implementation

We have implemented a prototype of the Kingfisher cloud provisioning system that allows each application provider (i.e., customer) to independently optimize their provisioning objectives (e.g., optimize either rental or transition cost while provisioning sufficient capacity for the application). Kingfisher's application-centric provisioning engine interfaces with a cloud management platform that manages the cloud environment with support for VM deployment, basic image management, etc. In this work, we used a modified version of the OpenNebula toolkit to implement Kingfisher's cloud management platform. We use OpenNebula to deploy/undeploy VMs on/from a set of servers (private-cloud), or create/terminate instances on Amazon's EC2. We use the XML-RPC APIs exposed by OpenNebula deploy, terminate, or reconfigure servers allocated to an application.

The architecture of Kingfisher and its relationship to the cloud orchestration framework is shown in Figure 1. Our system consists of following key components:

Monitoring engine: Since OpenNebula does not implement any sophisticated monitoring capabilities, the Kingfisher architecture includes a monitoring engine to track both application-level workloads and system-level resource usage for virtual machines and physical servers. Our monitoring engine is implemented based on the Ganglia [7] monitoring system. Ganglia consists of a server component (gmetad) that aggregates monitoring statistics from various machines, and a reporting agent (gmond) which runs inside each virtual machine. By default, Ganglia monitors only system-level metrics such as CPU, disk, memory and network usage. We enhanced ganglia to also monitor application workloads by enhancing the reporting agent to track application logs in real-time, and report statistics such as number of requests serviced and their service times seen over a reporting period (currently every 15s in our system). 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.

Ganglia stores monitored statistics in a custom round-robin database (RRD); our monitoring engine can track resource usage and application workload statistics by periodically querying this database and determining whether any user-specified thresholds have been exceeded (e.g., thresholds on SLA violations, request drops, or resource utilization). Finally, 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. More specifically, we extract a time-series of prior workload observations from the monitoring engine and model it as a ARIMA time-series. We use the ARIMA forecasting libraries in $R$to predict the future peak workload. The forecasted workload allows Kingfisher to plan a transition to a new configuration when it detects that the capacity of the current configuration may be exceeded. In our experiments (in Section 5), we focus on evaluating the cost benefits of Kingfisher, hence we assume a perfectly accurate forecaster.

Capacity planner: The capacity planner is the heart of Kingfisher's provisioning engine. It implements our ILP-based algorithm for optimizing the server rental cost for 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 rental 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 rental cost or transition cost. The planner then uses lpsolve to solve the LP approximation of our ILP, and uses a heuristic to convert the LP solution into an integer solution. Recall from the previous section, the solution specifies both a new configuration of servers for the application and a transition plan for transforming the current configuration to the new one.

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.

Live migration is implemented by simply initiating a migration to a new server type. For replication, the machine image is specified and a deployment request for a new server type is made. Shutdown-and-migrate involves shutting down the application, capturing a new machine image using the current disk state, and then specifying this machine image along with the request for a new server type.

5 Experimental Evaluation

In this section, we evaluate the efficacy of Kingfisher on a private as well as a public cloud. For the private cloud, 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 [12] 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.

5.1 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 the VM (by pinning the VM's VCPUs to the cores).

We created a virtual applicance 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.

5.1.1 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. For instance, an application will see different capacities due to differences in the processor clock frequencies, hardware caches sizes, or even the RAM on machines. Such hardware heterogeneity is common since most large clouds, including ones such as EC2, employ multiple hardware server configurations. Even with identical hardware, software artifacts often prevent linear scale-up with number of cores. To deal with this heterogeneity when making provisioning decisions, Kingfisher resorts to empirical profiling to determine the application's capacity on each server type.

To demonstrate these effects, we configured TPC-W with both tiers in a single VM, and ran this VM on 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. Figure 2 plots the empirically derived capacities for various multi-core configurations on our Intel and AMD systems. As shown, on private cloud, dual-core VMs on Xeon-servers have a capcity 1.77 time of that of a single core, while quad-core VMs have 1.8 times the capacity of a single-core system, but with increased RAM it becomes 2.2 times. However in the case of AMD, the scaling is very different. As can be seen, there are also differences between Intel and AMD servers with the same number of cores.

In our subsequent experiments we have used only AMD single and quad-core systems as s̈malländ l̈arge; we assume that these empirically derived capacity profiles are made available to Kingfisher to aid its provisioning decisions.

5.1.2 Cost-aware versus Cost-oblivious Provisioning

Our first provisioning experiment compares our cost-aware approach to a cost-oblivious approach (which ignores rental costs when provisioning servers). Our baseline cost-oblivious approach uses the same ILP formulation as Kingfisher but with a linear cost model (where an n-core servers cost n times as much as a single core system, effectively causing the ILP to ignore rental costs). It is important to note that, for a fair comparison, our cost-oblivious approach does take our empirically derived capacity profiles into account. In contrast, Kingfisher's cost-aware approach will account for both non-linear

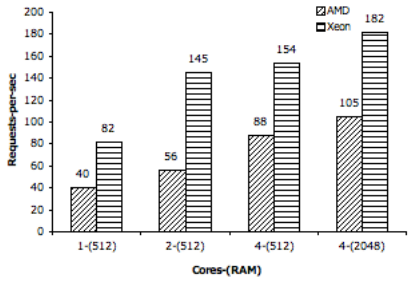
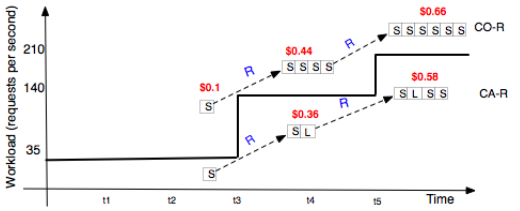
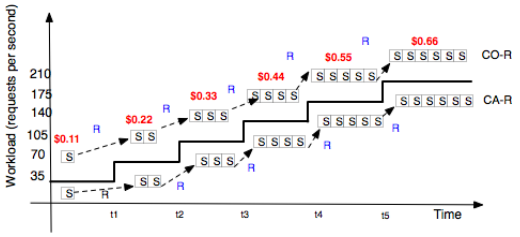


Figure 2: Non-linear scaling behavior of TPC-W on multi-core servers



(a) large workload jumps



(b) small workload jumps

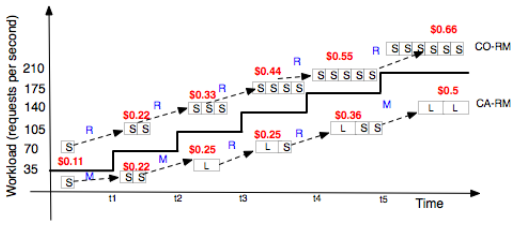
Figure 3: Cost-aware versus cost-oblivious provisioning

capacity scaling and pricing differentials across servers.

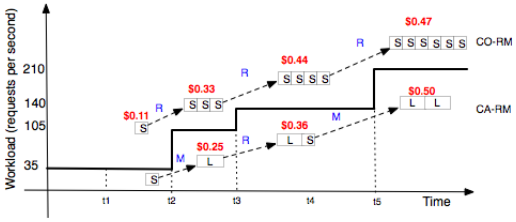
Since many prior provisioning cost-oblivious approaches are based on replication as the primary means to increase capacity, we restrict both approaches to replication-based provisioning. We denote these two approaches as CA-R (cost-aware with replication) and CO-R (cost-oblivious with replication). For ease of exposition we have used only two types of server-classes, namely small and large and used the NS-cloud platform's pricing model, mentioned in Table-1, for the same. On our TPCW-application, we increase the request rate from 35 to 210 req/s in moderate steps, causing the provisioning algorithm to be invoked each time. Figure 3(a) depicts the servers configurations chosen by the CA-R and CO-R approaches and the rental cost of each configuration. As shown, the cost-oblivious approach chooses an increasing number of small servers each time, while the cost-aware approach chooses a combination of small and large servers, resulting in a lower hourly rental cost (while provisioning the same capacity).

Next we repeat the above experiment with a workload that increases in small steps. As shown in Figure 3(b), both approaches choose *identical* configurations—an increasing number of small servers. Since the workload increases in small steps, one extra small server suffices to service this workload increase after each step, which is what both approaches choose. Further, we have (artificially) restricted our cost-aware approach to use only replication. Consequently, it is unable to consider replacing a set of small servers with a cheaper quad-core server (which requires migration). This experiment also motivates the need to use both replication and migration to obtain optimal results.

5.1.3 Benefits of using migration and replication

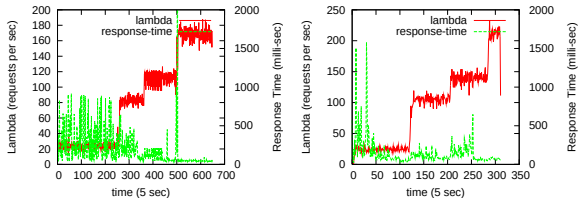


(a) small workload increase



(b) large workload jumps

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.

Next, we repeat the previous experiment, but allow both approaches to use both migration and replication. By allowing our approaches to use both mechanisms, our provisioning algorithms are able to consider a larger set of feasible configurations, which can yield higher savings in the rental cost. As shown in Figure 4(a), even though the workload increases

in small steps as before, our cost-aware approach now outperforms the cost-oblivious approach. This is because, it chooses to use migration to transition one or more small servers to a single large server, extracting the benefits of a low cost-per-core pricing of quad-core servers. For a fair comparison, we let the cost-oblivious approach use both migration and replication as well; consequently, unlike Figure 3(b), it chooses occasionally chooses a large server as well; but its choices are frequently more expensive than those of the cost-aware approach.

Figure 4(b) compares the two approaches for scenario where the workload increases in large steps. As shown, by using the best possible mechanism for each step (i.e., migration or replication), the cost-aware approach is able to pick lower-cost configurations. Figure 5 shows the workload on the TPC-W application and the avrage response-time along the course of its evolution. It should be observed that The transition-cost aware system quickly achieves the desired configuration and thus the response-times reduce early along the curve.

5.1.4 Transition cost-aware Provisioning

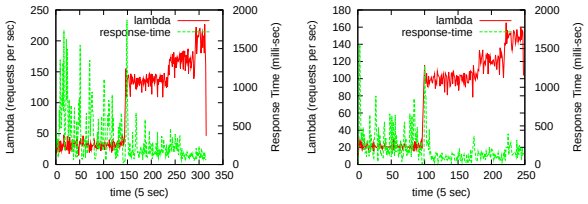
Our experiments thus far have focused on optimizing rental cost and have ignored the overhead of transitioning the application from one configuration to another. Kingfisher's transition cost-aware provisioning approach is useful for quickly provisioning capacity (e.g., in scenarios where the workload surges suddenly); however, our approach may not necessarily minimize rental costs since its focus is on quick, rather than cheap, provisioning.

To demonstrate the benefits of our approach, we subjected our TPCW web application to a workload that increased in large steps, as depicted in Figure 6(a),6(b). As each step, we invoked Kingfisher's transition cost-aware provisioning and compared the decisions made by this approach with its rental cost-aware provisioning method (which ignores the transition cost when making decisions, and is henceforth referred to as a transition cost-oblivious approach). In this case, we assumed a cloud platform with two servers small (S) and large (L), with rental costs of $0.11 and $0.25 per hour, respectively (as in Table-1).

Figure 7 depicts our results. Figure 7(a) and 7(b) plot the configurations chosen by the transition cost-oblivious and transition-cost-aware approaches and their corresponding rental costs. Also shown are the actual latencies incurred in our cloud platforms to complete the migration and replication steps. As shown in the figure, after the second workload increase, the cost-aware approach chooses a migration to a large server, while the cost-oblivious servers starts up two small replicas. The migration latency is only 7 seconds while that for replication is an order of magnitude higher. The lower latency, however, is achieved at the expense of a higher rental cost ($0.5 versus $0.47 per hour). In the third step, the cost-aware approach does not perform any reconfiguration, while the cost oblivious approach performs migration. In this case, the latency of the replicate operations is comparable, but we note that the cost-oblivious approach incurs twice the data copying cost (due to the need to copy the machine image to two small servers), while the cost-aware approach reduces the data copying cost, again at the expense of a somewhat higher rental cost. It should be noted that since we have considered only two types of servers, for ease of exposition, the result shows a small benefit; our subsequent EC2 experiments consider a larger number of server types, allowing for a larger number of reconfiguration possibilities.

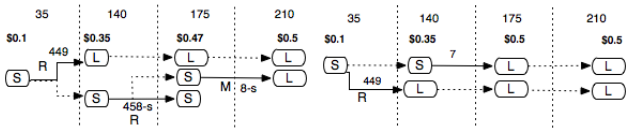
The response-times (request drops) seen by the two approaches are shown in Figure 6(a) and (b). Since the transition latency of a migrate operation is lower than that for replication, the response time stays high for longer duration in the transition-cost-oblivious approach; the response times are comparable in the first and third step since both approaches resort to replication in first step and in third step live-migration is very quick.

Overall, the experiment demonstrates that copying of memory state during a live migration incurs lower latencies than copying of disk images during replication (typically memory state is smaller than disk image sizes, and memory-to-memory transfers are faster than disk-to-disk copying). Hence, live migration may be preferred, whenever feasible, to reduce transition costs. The experiment also demonstrates that 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 (e.g., in the second step).



(a) Transition-cost Oblivious (b) Transition-cost Aware

Figure 6: Workload and Response Times of a transition-cost aware system with a transition-cost oblivious system.



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

Figure 7: Comparison of a transition-cost aware system with a transition-cost oblivious system.

5.1.5 Impact of the Pricing Model

Prior experiments have assumed a convex pricing model where the cost-per-core reduces with increasing number of

cores in the system. 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 (the desired capacity is varied from λ to 6λ). We assume that the cloud supports three types of servers, small, medium and large. First, we assume a convex pricing model, which resembles the ones employed in today's 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 a 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. Finally, we choose an arbitrary pricing model, where the medium server is the cheaper, 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 chooses some large server instances. For comparison, we also show the results of the cost-oblivious approach, which always chooses small servers regardless of the pricing model.

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 2: Provisioning with different pricing models

5.2 Evaluation on the Public EC2 Cloud

Amazon EC2 supports seven EC2-instance types [5], namely small, large, xlarge, high-cpu-medium, high-cpu-xlarge, high-mem-dxlarge, high-mem-qxlarge. We chose the first five configurations / instance-type for our experiments. EC2-instances can be created either from the instance-store or from EBS-volume snapshots, where an EBS-volume is a persistent storage which is charged at a rate of $0.1-perGB-month. Amazon offers snapshoting capability on these EBS-volumes; as mentioned, these snapshots can be used to create new EC2-instances. Besides this, an EC2-instance created using EBS-snapshots can be stopped and started into another EC2-instance type (which is involves only the stopping and starting-time). In order to test our provisioning system we created a replica of our private cloud TPCW-images (of size 10GB), both on the instance-store as well as on EBS-volumes. We used HAProxy as a load balancer; we used a dedicated high-cpu-medium instance as a load-balancer.

5.2.1 Profiling Server Capacity

Like in the private cloud, Kingfisher must profile TPC-W's performance on each server type before it can make provisioning decisions. To do so, we chose first EC2 instance types, namely small (m1.small), large (m1.large), xlarge (m1.xlarge), high-cpu-medium (c1.medium) and high-cpu-xlarge (c1.xlarge); these instances have 1, 4, 8, 5 and 20 EC2 compute units (ECUs), respectively. Like in private cloud, we empirically estimated the capacity of each of these instances and Figure 8 plots the derived capacities for various EC2-instances. Our results show several interesting insights. First, like in the private cloud, it shows that TPC-W scales non-linearly with the number of ECU (logical cores). Second, it also demonstrates that the performance is dependent not only on the number of cores but also the RAM and I/O performance on each server. For example, it shows that the performance of a 8 ECU xlarge server is better than a 20 ECU/core high-cpu-xlarge server. This is because the former has a larger amount of RAM (15GB versus 7GB), which appears to impact application performance more than the number of logical cores / ECUs [5]. A similar result is true for the large and the Med-fast server type, where the latter has one more logical core but the former has significantly more RAM, giving it a performance edge.

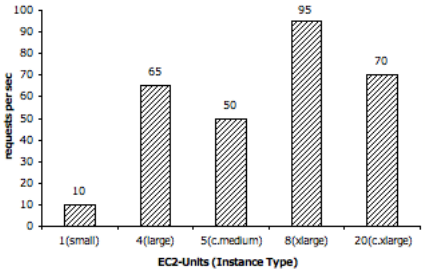


Figure 8: Non-linear scaling behavior of TPC-W on EC2-instances

As we will show next, Kingfisher is able to easily take these capacity profiles, resulting from arbitrary hardware differences, as well as the non-linear pricing model uses by Amazon into account when making provisioning decisions.

5.2.2 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, so that they can be fed into Kingfisher's optimization algorithm. We present a summary of these results below.

EC2 provides two mechanisms from starting up a new new replica: (1) using an EBS-volume image or (2) using the instance-store. Unlike our 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 the machine image, which contains the old disk state; 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.

We conducted experiments to quantify the overheads of these operations. Table-3 depicts the latency to start-up an instance. There are two mechanisms using with an EC2 instance can be started, namely 1.) one using the instance-store (i.e S3) 2.) using an EBS volume image. From our observations, the time to create an instance from and instance-store is linearly proportional to the size of compressed image, however in the second case its nearly constant;

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 cost of the composite step is nothing but the sum of the cost of individual 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 5.) create an instance using this new AMI. Table-3 shows the time taken to complete each of these steps for different size-images. It can be seen from the table that the the total time is linearly varying with the size of compressed image. The procedure of doing shutdown-copy-migrate on an EBS-volume based instance has three steps: (i) take a snapshot of EBS-volume, (ii) register an AMI using the snapshot, and (iii) create and instance. We conducted two separate experiments for the sake of convenience. Table-4 depicts the time it takes to take a snapshot of volume which contains data which cannot be compressed any-further. It can be approximately modeled as a linear function of compressed image The time to take the snapshot of an EBS volume can also be modeled as a linear function of size of compressed image size. The time it takes to boot an instance from EBS-snapshot is nearly constant; our measured average value is 65 sec. The reason for this is that EBS-service firstly copies the volume's required-blocks, for starting-up the image, and thus the machine boots quickly but the rest of the disk is lazily copied over to the new volume. 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 involves following three steps 1.) stop the instance 2.) change its attributes 3.) start the instance. We found that its a constant for an image (irrespective of the instance-type). The mean overhead for these steps is 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

Table 3: 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

Table 4: 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

Table 5: Time measurements of start-up time of an image from EBS-volume

5.2.3 Rental- and Transition-cost aware Provisioning in EC2

To evaluate the efficacy of Kingfisher in taking rental and transition costs into account, we repeated a version of 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 rental cost of servers is summarized in Table-1) and transition cost is as discussed above. Like before we varied the workload in steps and Table-6 depicts the configurations generated by the cost-oblivious and Kingfisher's cost-aware methods. Like before, the cost-aware (CA-RM) method is able to provision the same capacities at a lower rental cost.

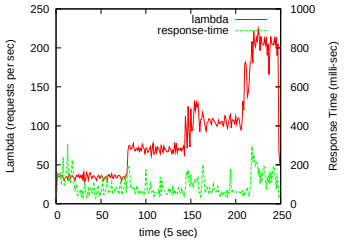
Since these solutions do not account for the reconfiguration/transition costs, we repeated this experiment for three flavors of transition-aware provisioning that capture the real-life constraints specific to EC2: (i) TA-RM-1, which only takes

into account the number of transitions and cost of each transition and also the rental cost of final configuration; (ii) TA-RM-2: that considers transition costs and final rental 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.

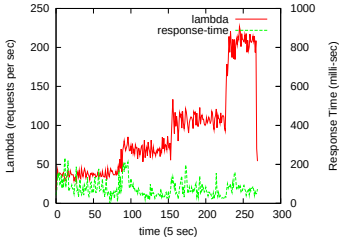
In order to compare CA-RM configurations with that of TA-RM, we let Kingfisher choose the least transition-cost scheme to achieve the final configuration suggested by CA-RM. Initially, TA-RM-* chooses to perform only one migration stop-and-start as opposed to 2-migrations as chosen by CA-RM; also notice that both the 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 resort 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 TA-RM-2, it chooses the faster replication option (over the slower stop-and-start option). TA-RM-3, on the other-hand, performs a stop-and-start migration from small to medium instances and then initiates another replication.

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)

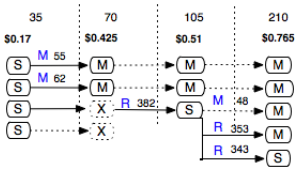
Table 6: Provisioning with different methods; here $\lambda = 35$



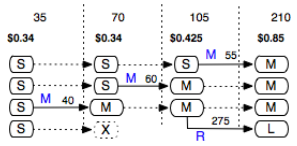
(a) Transition-cost oblivious



(b) Transition-cost aware



(c) Transition cost oblivious provisioning



(d) Transition cost-aware provisioning

Figure 9: Comparison of a transition-cost aware system with a transition-cost oblivious system.

Figure-9(d) and Figure-9(c) depict the configurations chosen by TA-RM-3 scheme and also the time to execute the operation. Figure-9(a) and Figure-9(b) show the response-times of the of the corresponding configurations, showing how the configuration responds to the workload. The benifit of transfer-cost aware system are apparent as it finished early.

6 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 [6] and [3]. Much of this work is platform-centric, while our work considers a customer-centric view of provisioning and resource optimization. Other work has considered migration as a means of dynamic provisioning [8], 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 [14, 11, 16, 18]. An online measurement approach to estimate capacity model parameters was proposed in [2]. The approach employs a queueing model that partitions capacity of a shared server among multiple hosted applications. Classical feedback control theory has also been used to model the bottleneck tier for providing performance guarantees for web applications [1, 15]. The approach in [15] 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 [13] 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 [4].

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 rental and transition costs, our approach has involved a full prototype implementation and experiments on an actual Linux cluster.

7 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. We also presented a case study of how our approach can be employed in a public cloud such as Amazon EC2.

References

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[2] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of Eleventh International Workshop on Quality of Service (IWQoS 2003)*, June 2003.

[3] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. *ACM SIGOPS Operating Systems Review*, 35(5):103–116, 2001.

[4] J. Chen, G. Soundararajan, and C. Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *IEEE International Conference on Autonomic Computing (ICAC)*, pages 231–242, June 2006.

[5] A. EC2. Amazon ec2. Website. `http://aws.amazon.com/ec2`.

[6] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France*, pages 78–91, December 1997.

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

[8] L. Grit, D. Irwin, , A. Yumerefendi, and J. Chase. Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In *In the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November 2006.

[9] L. Kleinrock. *Queueing Systems, Volume 2: Computer Applications*. John Wiley and Sons, Inc., 1976.

[10] Opennebula. `http://www.opennebula.org`.

[11] 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.

[12] TPC. the tpcw benchmark (java implementation). Website. `http://tpcw.deadpixel.de`.

[13] 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, June 2005.

[14] 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*, pages 1–39, March 2008.

[15] 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.

[16] 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.

[17] 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.

[18] 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*, pages 172–181, 2008.