RESOURCE PROVISIONING EXPLOITING COST AND PERFORMANCE DIVERSITY WITHIN IAAS CLOUD PROVIDERS

A thesis submitted in fulfilment of the requirements for the
degree of Master of Philosophy in the The Faculty of Engineering and IT at
The University of Sydney

Luke M. Leslie

October 2013

I have examined this thesis and attest that it is in a form suitable for examination for the degree of Master of Philosophy.

(Albert Y. Zomaya) Primary Supervisor

I have examined this thesis and attest that it is in a form suitable for examination for the degree of Master of Philosophy.

(Young Choon Lee) Associate Supervisor

I have examined this thesis and attest that it is in a form suitable for examination for the degree of Master of Philosophy.

(First Reader)

I have examined this thesis and attest that it is in a form suitable for examination for the degree of Master of Philosophy.

(Second Reader)

Abstract

Infrastructure as a Service (IaaS) Cloud Computing platforms such as Amazon EC2 allow clients access to massive computational power in the form of virtual machines (VMs) known as instances. Amazon hosts three different instance purchasing options, each with their own service level agreement covering availability and pricing. In addition, Amazon offers access to a number of geographical regions, zones, and instance types from which to select. In this thesis, the problem of utilizing Amazon's Spot and On-Demand instances is analyzed in the context of (1) maximizing profit while fulfilling user requests for instances, and (2) minimizing cost while executing deadline constrained jobs. To this end, two approaches are presented that are designed to exploit the cost and performance diversity among different instance types and availability zones, and among the various Spot markets they represent.

To fulfill user requests for instances RAMP, a Reliability-Aware Profit-Maximizing Resource Provisioner, is developed. RAMP employs novel strategies designed to calculate the expected profit of using a specific Spot or On-Demand instance through a comparison of the reliability of that instance. To execute deadline-constrained jobs, RAMP is extended to develop RAMC-DC, a Reliability-Aware Cost-Minimizing Resource Provisioner for Deadline Constrained Jobs. RAMC-DC allocates the most cost effective instance through strategies designed to promote interchangeability of instances among short jobs, determine the likelihood of completion of long jobs on specific instances, and compare the estimated costs of possible allocations that satisfy availability and reliability constraints. In addition, RAMC-DC achieves fault tolerance through comparisons of the price dynamics across instance types and availability zones, and through an examination of three basic checkpointing methods incorporating the migration of VM states to different availability zones.

Evaluations using Amazon Spot price history demonstrate that both RAMP and RAMC-DC achieve a large step toward low-volatility, high cost-efficiency resource provisioning. While achieving early-termination rates as low as 2.2%, RAMP can completely offset the total cost when charging the user just 17.5% of this On-Demand price. Moreover, the increases in profit resulting from relatively small additional charges to users are notably high, i.e., 100% profit compared to the resource provisioning cost with 35% of the equivalent On-Demand price. Furthermore, RAMC-DC can maintain deadline breaches below 1.8% of all jobs, achieve both early-termination and deadline breach rates as low as 0.5% of all jobs, and lowers total costs by between 80% and 87% compared to using only on-demand instances.

Acknowledgements

I would like to thank my supervisor, Prof. Albert Y. Zomaya, and my associate supervisor, Dr. Young Choon Lee, for their continual support and guidance.

Contents

List of Tables

List of Figures

List of Algorithms

Implementations

Chapter 1

Introduction

1.1 Cloud Computing

The term *Cloud Computing* is a broad label describing both the application services that are available through the internet, and the software and hardware that make up the *Cloud* that underpins these services (Armbrust et al. [2010]). Although not a new concept, Cloud Computing has emerged as a way to provide highly elastic computing services to consumers on an on-demand, self-service basis that allows fine-grain pricing of services with very low barriers to entry. In addition, Cloud Computing providers are able to alleviate many risks generally faced by consumers in the deployment of web applications including security, maintenance, and scalability, and are able to provide guarantees regarding the performance and reliability of offered services. Due to the multitenancy attribute of Cloud Computing, Cloud providers are also generally in a prime position to leverage economies of scale, and therefore can pass these savings on to the consumers, further reducing the costs associated with deploying applications in the Cloud.

A distinction is made between four types of Cloud deployments: *Public Clouds*, which provide application, storage, computing, and other services to the public; *Private Clouds*, which provide them to a closed environment such as a university or business; *Community Clouds*, which provide infrastructure services within a specific community; and *Hybrid Clouds*, which are a combination of multiple Clouds, either Public, Private, or Community (Mell and Grance [2011]). Public Cloud Computing providers have generally offered services through three basic service models: *Infrastructure-as-a-Service* (IaaS), *Software-as-a-Service* (SaaS), and *Platform-as-a-Service* (PaaS) (Mell and Grance [2011]). Figure 1.1 provides a broad overview of what these three service models offer to the consumer.

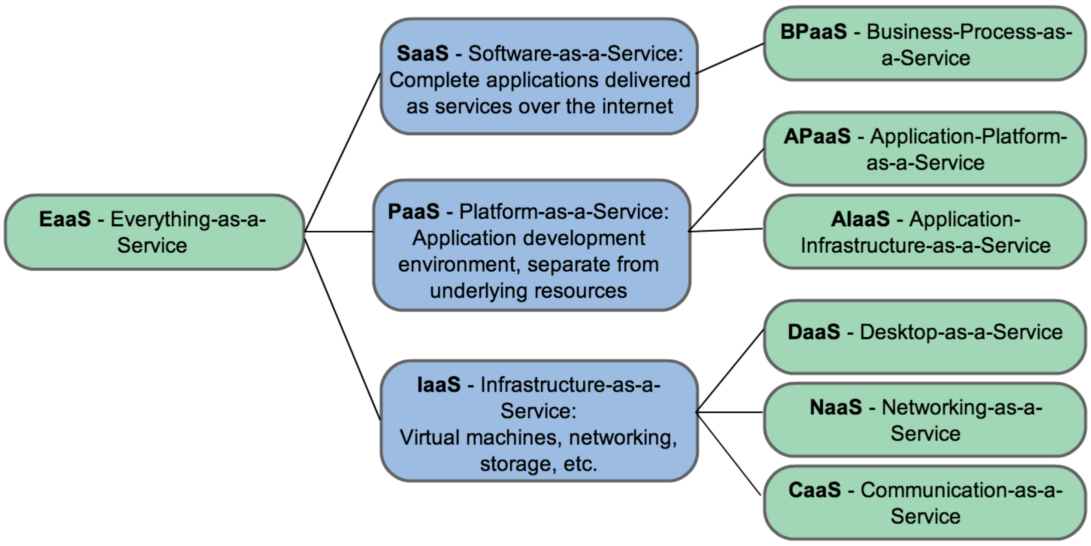

EaaS - Everything-as-a-Service
SaaS - Software-as-a-Service: Complete applications delivered as services over the internet
PaaS - Platform-as-a-Service: Application development environment, separate from underlying resources
IaaS - Infrastructure-as-a-Service: Virtual machines, networking, storage, etc.
BPaaS - Business-Process-as-a-Service
APaaS - Application-Platform-as-a-Service
AIaaS - Application-Infrastructure-as-a-Service
DaaS - Desktop-as-a-Service
NaaS - Networking-as-a-Service
CaaS - Communication-as-a-Service

Figure 1.1: The three basic Cloud service models.

IaaS providers, such as Amazon EC2 and Google Compute Engine, differ from SaaS and PaaS providers by allowing users direct access to the infrastructure they require; within IaaS platforms, users generally pay hourly rates to gain access to virtual machines known as instances, as well as paying for storage services such as Amazon S3. On the other hand, SaaS providers, such as Salesforce.com, let users lease access to software and data hosted on the cloud, while PaaS providers, such as Google App Engine, let users lease application development platforms. Recently, two other variations: Network-as-a-Service (NaaS), and Communications-as-a-Service (CaaS) were officially added to the primary three models, and there are many other sub-variations of the main three models (International Telecommuniation Union [2012], see Figure 1.1).

Many Cloud Computing providers, such as Amazon EC2, provide a means to acquire pay-as-you-go computing power and data storage in a manner similar to publicly available utilities such as gas and electricity. Due to the similarity between such pay-as-you-go computing services and commonly available utilities, these services are commonly known as *Utility Computing*. Utility Computing cloud providers can provide highly cost-efficient deployment platforms through three major characteristics (Armbrust et al. [2010]):

1. Consumers are given access to virtually limitless computing resources through the illusion of infinite capacity. Users can meet whatever demand they face by leasing and releasing resources as needed. [1]

2. Startup ventures do not need to accrue any of the fixed costs that have existed in the past. All hardware and maintenance costs are borne by the cloud provider. Instead, consumers pay only for the time spent using the resources.

3. Consumers pay only for what they use. Such a model for consumption facilitates a *conservative* approach to resource provisioning that helps reduce the inefficiency of server farms and can have significant impacts in the quest for green computing.

These three characteristics have allowed Cloud Computing to become a very powerful and popular tool among users who require access to computational resources and data storage without the fixed costs involved in purchasing, installing, and maintaining a private cloud.

1.2 Motivation

Amazon EC2 offers three instance purchasing options, (1) Spot instances, (2) On-Demand instances, and (3) Reserved instances, all described in detail in the next chapter. Each purchasing option is coupled with different pricing and volatility characteristics. Furthermore Amazon EC2 offers many different instance types in many different geographical locations (known as regions), and in many availability zones for even finer-grain placement. Each availability zone and instance type combination defines a Spot market in which users may bid against a time-varying market price for Spot instances, inexpensive but volatile virtual machines that are subject to early-termination by Amazon. On the other hand, Reserved and On-Demand instances are statically-priced and highly reliable virtual machines that are leased and terminated at the discretion of the

[1]For example, in March, 2012, Amazon EC2 was estimated to be powered by approximately 500,000 servers (Wired [2012]).

user only, with Reserved instances requiring an initial fixed cost, the size of which translates to lower hourly prices.

Faced with the diversity and volatility of resourced within IaaS providers such as Amazon EC2, differing questions emerge including:

1. How can we lease and allocate Spot instances in order to *prevent* early termination, while exploiting potential cost savings from low market prices?
2. How can we provide fault-tolerance to *manage* early termination once Spot instances have been leased?
3. Due to their fluctuating market price, what is the best way to estimate the costs of using Spot instances?
4. How best can we exploit the diversity among different availability zones, instance types, and their respective Spot markets, in order to further reduce incurred costs and provide more fault prevention?
5. How can we combine different purchasing options to subsume the best qualities of each?
6. How can we schedule deadline-constrained jobs or user requests for instances on a cluster comprised of these diverse options?

The two frameworks presented in this thesis aim to provide answers to these questions.

1.3 Contributions

The frameworks presented in this thesis focus on the efficient and elastic acquisition and allocation of Spot and On-Demand instances in order to satisfy different resource and system requirements. To accomplish this, two frameworks are introduced with the goal of accomplishing one of the following two objectives:

1. To reliably provide users, who require some minimum instance type for a desired amount of time, uninterrupted access to an instance that satisfies his/her requirements. The search for a candidate instance is conducted with the goal of maximizing the profit of the framework.

2. To cost-effectively schedule and execute deadline-constrained jobs on a dynamically provisioned cluster of Spot and On-Demand instances while meeting reliability and availability constraints on those instances.

However, to effectively develop both of these frameworks, due to the lack of *a priori* knowledge associated with using Spot instance, a means to estimate the total cost of running a Spot instance for some desired period of time is required. Thus, in addition, five methods designed to estimate this cost are presented and evaluated before the two frameworks are constructed. Each of the above frameworks will be outlined below.

1.3.1 Profit-Maximizing Resource Provisioning for Instance Requests (RAMP)

RAMP,[2] a Reliability-Aware Profit-Maximizing Resource Provisioner, accepts requests from users for desired instance types and availability times (specified in full and partial hours). For each request, the user is charged some portion of the On-Demand price for their corresponding requested instance type. To fulfill these requests, RAMP seeks to compare the expected profit of using different types of Spot and On-Demand instances in different availability zones. Furthermore, to provide compensation for volatility, in the event of early-termination of a Spot instance, the user is refunded their full payment plus a penalty. RAMP differs from others, such as the Dynamic and Stochastic Resource Rental Planners (DRRP and SRRP) (Zhao et al. [2012]), BROKER (Voorsluys and Buyya [2012]), and that presented by Chen et al. [2011], by introducing and utilizing:

- **A *reliability-aware* instance acquisition strategy** consisting of a trace-based reliability function and a bidding strategy for leasing specific Spot instances of a specific type and availability zone.

- **A *profit-driven* resource allocation strategy** that seeks to acquire new instances, or reassign idle instances, in order to fulfill user requests for some minimum instance type.

To examine the effectiveness of our approach, and in order to capture the market-price dynamics over an extended time period, simulations were run using Amazon's

[2]Our research on RAMP is currently submitted to the IEEE Transactions on Parallel and Distributed Systems (TPDS) journal.

Spot price history over four months from the period between July and November 2012, and with 20,000 requests taken from traces for the ANL Intrepid supercomputer. Results show that our approach can:

- Achieve early-termination rates as low as 2.2% of all requests, compared to approximately 13% when simply bidding the market price for each Spot instance.
- Completely offset its total cost when paying around only 17.5% of the equivalent On-Demand price for each requested instance.
- Achieve profits equal to 100% of the total baseline cost while charging the user just 35% of the On-Demand price, regardless of the penalty paid in the event of early-termination, compared to 60% when using market-price bids.

1.3.2 Cost-Minimizing Resource Provisioning for Deadline-Constrained Jobs (RAMC-DC)

RAMC-DC [3] is a resource provisioning and job scheduling framework that exploits performance and cost diversity in the cloud in order to minimize the total cost while scheduling deadline-constrained jobs on Spot and On-Demand instances. The approach presented in this thesis is unique, in comparison to, among others, those studies listed above, in that it compares price dynamics from different types of instance purchasing options, instance types, and availability zones, as well as different methods to estimate the cost of execution on a spot instance. Furthermore, RAMC-DC incorporates a two-tier bidding strategy designed to incorporate:

- **The *interchangeability* of short jobs on an instance.** For shorter estimated execution times, the need to acquire instances specifically for that job becomes less cost-effective than a strategy that acquires instances with the goal of being interchangeable among other short jobs.
- **The *probability of completion* of long jobs on an instance.** Due to the volatility of spot instances, the instance acquisition strategy used by RAMC-DC when leasing long jobs must ensure that the instance can successfully complete that job

[3]Our research on RAMC-DC has been published in the 2013 IEEE International Conference on Cloud Computing (CLOUD) (see Leslie et al. [2013]).

with some level of probability. To do this, RAMC-DC calculates the reliability function of the instance, as discussed above.

To fulfill jobs, RAMC-DC utilizes a resource allocation strategy that manages a dynamically-sized cluster of Spot and On-Demand instances. The resource allocation strategy used by RAMC-DC evaluates:

- **The *cost effectiveness* of each possible instance allocation.** RAMC-DC seeks to approximate the cost of running a job on a Spot or On-Demand instance, and locates the instance that minimizes this cost while meeting interchangeability and/or reliability constraints.

An evaluation of RAMC-DC was performed using the same set of jobs and Spot prices as above, but with the incorporation of estimated and true execution times for these jobs. In addition, Downey's speedup model as used an exemplar means to determine moldability in jobs. Results from simulations run using Spot price traces from the same period as the evaluation of RAMP show that RAMC-DC can:

- Save between 80% and 87% of the equivalent total cost when using only On-Demand instances.
- Maintain early-terminations in as low as 0.18% of all jobs. Adjusting the input parameters allows RAMC-DC to trade total cost for lower early-termination rates.
- Achieve deadline breaches in as low as 0.47% of all jobs, with the incorporation of periodic checkpointing maintaining deadline breaches in less than 1.1% of all jobs for all input evaluation lower bound levels greater than 0.

These results indicate that RAMC-DC, when incorporating Spot and On-Demand instances, can significantly reduce total costs while maintaining very low volatility. Additionally, even if no moldability is assumed, costs and volatility are still extremely low, and the number of deadline breaches is actually minimum when jobs are assumed to be rigid (i.e., no speedup is encountered on larger instances).

1.4 Materials

The frameworks presented in this thesis were implemented using C++ on Mac OSX. Amazon's Spot market price history were obtained and collated using Amazon's AWS

API in Java for three month segments. These price traces are also available online through the AWS Management Console (Amazon [2013a]), and through websites such as that provided by the University of Western Sydney (University of Western Sydney [2013]). Different On-Demand instance types and prices for those instance types were taken from Amazon EC2's website (Amazon [2013d]). Workload traces used for requests and jobs were downloaded from the Parallel Workloads Archive (par [2013]) and translated from the SWF (Simple Workload Format). Random number generation was performed using the C++ Standards Committee Technical Report 1 (std::tr1) extensions. All experiments were run on a machine with a four-core 2.3 GHz Intel Core i7 processor and 8 GB of RAM, and on a machine with 4 $\times$ 8-core 2 GHz Intel Xeon E5-2650 processors and 32 GB of RAM.

1.5 Thesis Outline

The remainder of this thesis is organized as follows.

- **Chapter 2:** An overview of Amazon EC2 is given with a focus on the performance and cost diversity that is the focus of our research. Related literature is then presented and critiqued, gaps in current research are identified, and and major differences between previous work and that presented in this thesis are outlined. Assumptions that are made in our research are then outlined.

- **Chapter 3:** Multiple methods to estimate the cost of running a Spot instance for a desired period of time are outlined and evaluated, and the motivation behind them is provided in the context of different periods of Spot prices.

- **Chapter 4:** The reliability-aware profit maximizing resource provisioner, RAMP, is presented. The problem formulation is given, and RAMP's instance evaluation, bidding, and profit maximization strategies are introduced and developed. RAMP is evaluated using Amazon's Spot price traces and artificially generated requests derived from traces from the parallel workloads archive.

- **Chapter 5:** The reliability and availability aware cost minimizing resource provisioner, RAMC-DC, is presented. The problem formulation is presented, and the strategies incorporated by RAMP are extended to account for the scheduling of deadline-constrained jobs on instances, and the necessary interchangeability of instances among short jobs. Furthermore, an exemplar of moldability (the variation in execution time on different instances) among jobs is introduced in an effort to broaden the search space. RAMC-DC is evaluated using Amazon's Spot price traces and an extension of the set of jobs used in the evaluation of RAMP. The effects of broadening the the search for suitable instance types by incorporating moldability is also analyzed.

- **Chapter 6:** The conclusions of the research presented within this thesis are presented, and the possible directions of future work are given.

Chapter 2

Background and Assumptions

2.1 Performance and Cost Diversity in Amazon EC2

The focus in the thesis will be on IaaS providers, specifically Amazon Elastic Compute Cloud (EC2): a web service designed to provide users with a reliable, inexpensive, and elastic platform from which to lease computational capacity. Amazon EC2 is a part of Amazon Web Services (AWS), Amazon's Cloud Computing platform, and has been out of beta since late 2008.

To rent computational capacity from EC2, a user must select an Amazon Machine Image (AMI) to be booted onto a virtual machine (VM), that functions as a private virtual server. This VM is known as an *instance*, and can be rented using one of three purchasing options as described in the following subsection. Amazon uses the Xen virtualization hypervisor for virtualization (Xen [2013]), and AMIs come with Linux, Solaris, Windows, and other operating systems, and are each priced differently.

2.1.1 Purchasing Options

To rent an EC2 instance, a user must choose between one of the three purchasing options discussed in Figure 2.1.

1. *On-Demand instances* have fixed hourly costs depending on the instance type, and may be started, stopped, and terminated at will by the user.

2. *Reserved instances* require that the user pays an initial cost for either a 1 or 3 year plan, with either *Light*, *Medium*, or *Heavy* utilization. Longer plans and

higher utilization translates to higher initial costs but lower hourly costs. Amazon recently introduced the Reserved Instance Marketplace, wherein users may sell off their unused Reserved Instances (Amazon [2013b]). The break-even points for Reserved vs. On-Demand instances are given in PlanForCloud [2013].

3. *Spot instances* have fluctuating hourly market prices and require the user to place a bid indicating a maximum price they are willing to pay for the instance. Spot instances are subject to early-termination by Amazon when the market price rises above the bid.

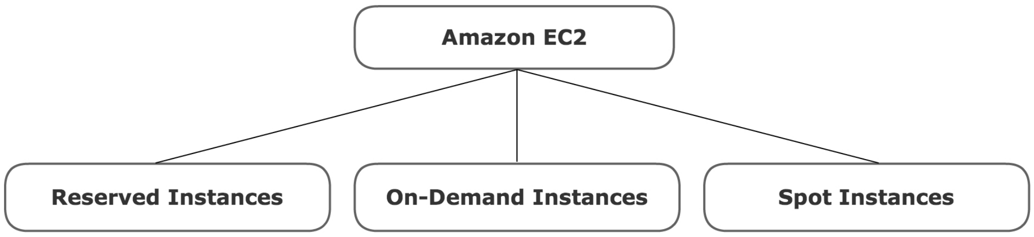
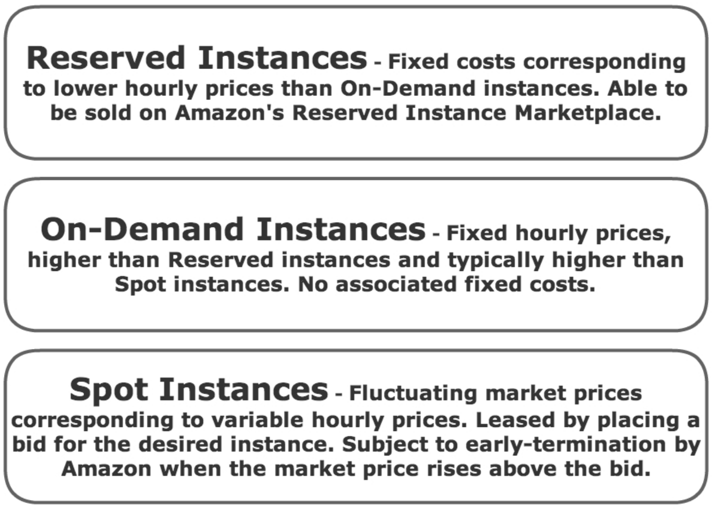


Figure 2.1: Amazon's instance purchasing options



Reserved Instances - Fixed costs corresponding to lower hourly prices than On-Demand instances. Able to be sold on Amazon's Reserved Instance Marketplace.

On-Demand Instances - Fixed hourly prices, higher than Reserved instances and typically higher than Spot instances. No associated fixed costs.

Spot Instances - Fluctuating market prices corresponding to variable hourly prices. Leased by placing a bid for the desired instance. Subject to early-termination by Amazon when the market price rises above the bid.

Figure 2.2: A description of Amazon's instance purchasing options

Each instance purchasing option is generally suited for specific applications. For example, in the context of service providers, Reserved instances are generally used to handle static load and On-Demand and Spot instances are generally used to handle elastic load. When executing tasks, Spot instances are generally used for applications that have some sort of built-in fault-tolerance, such as checkpointing via VM state migration, etc., or redundant execution on multiple instances.

The effective exploitation of service diversity when utilizing these various purchasing options can be of great practical importance for the cost-performance ratio faced by service providers. In this study, we address the effectiveness of such exploitation for profit maximization and cost minimization, when faced with the diverse cost and reliability characteristics of Spot and On-Demand instances. The combination of these two purchasing options can provide a very cost-effective means to provision instances, provided an analysis of the market-price dynamics of Spot instances and appropriate fault-tolerance strategies are incorporated.

When seeking to requisition an instance from Amazon EC2, the choice between leasing a Spot instance or an On-Demand instance corresponds to a tradeoff between cost and reliability. For example, as witnessed in the recent Pinterest case, supplementing Spot instances with On-Demand instances can help reduce both the total cost of an application, as well as the associated early-termination rates (High Scalability [2012]). Although bidding strategies for Spot instances can help achieve user-defined reliability constraints up to a certain level, such an approach may not be cost-effective if the market price for a Spot instance is higher than that of an equivalent On-Demand instance.

Spot instances allow Amazon to rent out unused EC2 capacity to users. Although Spot instances are generally priced far lower than On-Demand instances, they also have an associated inherent volatility. With suitable reliability evaluation and/or fault tolerance, however, Spot instances have been shown to provide a cost-effective alternative for many applications, especially when supplemented with On-Demand instances.

2.1.2 Instance Types and Availability

When users decide to lease an instance, they must choose a specific *instance type* and geographical location (known as an *availability zone*), within a certain *region*. Each instance type provides different amounts of memory, processing power, storage, etc., ranging from the smallest instance with 1 EC2 Compute Unit (the equivalent of a 1-1.2 GHz 2007 Opteron or Xeon Processor) and 1.7GB of memory, to cluster computing instances with 88 EC2 Compute Units and 60.5GB of memory, and to high storage and

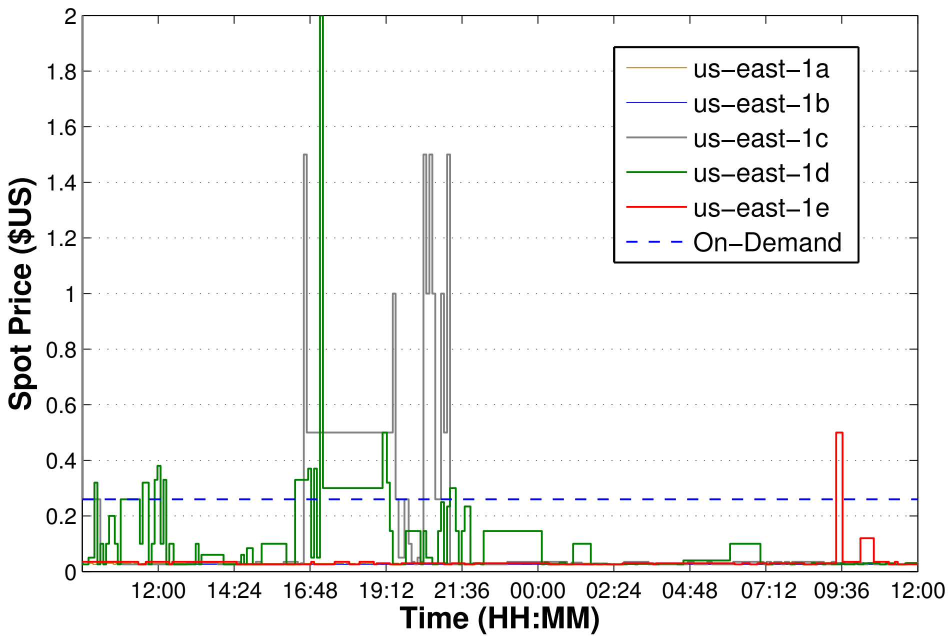


Figure 2.3: Spot prices for instance type *m1.large*, running a Linux OS, with all availability zones in region *us-east-1*. The equivalent On-Demand price of this instance type is $0.26/hour.

high I/O instances (Amazon [2013d]). A complete description of the instance types used in this thesis is given in Table 2.1. At present, Amazon allows users to lease instance from 8 different regions encompassing North and South America, Asia Pacific, and the European Union. Originally, users were only able to lease instances from specific regions. However, to increase fault tolerance, Amazon divided each region into different numbers of independent availability zones in which users could lease instances, and thus be able to spread their infrastructure across multiple data centers to eliminate a single point of failure. In addition to specifying the instance type and availability zone, if users wish to lease a Spot instance, they must also provide a bid, and if users wish to lease a Reserved instance, they must decide how much to pay upfront.

2.1.3 Storage

Many instance types come with the option for either local disk storage, with storage on a temporary volume linked to the instance, or Elastic Block Storage, which provides persistent storage that is independent of the instance and can be migrated among different instances. In addition to these storage options, AWS also offers a Simple Storage

Service (S3) that is accessible to EC2 instances and which may be used to persistently store data. S3 storage is billed on a monthly basis, and data transfer rates between an EC2 instance and S3 are free within the same region, and \$0.01/GB from different regions.

2.1.4 Characteristics of Spot Instances

Spot instances, introduced by Amazon in late 2009, are stochastically available virtual machines (VMs) with time-varying market prices determined by the provider (Amazon [2013c]). These instances typically are idle resources in a datacenter. Providers like Amazon lease out the VMs to increase the efficiency of servers and, owing to the typically lower costs of spot instances, capture different segments of the market than the statically-priced on-demand and reserved instances. Although spot instances generally are priced far lower than their statically-priced counterparts, they are coupled with the inherent volatility associated with bidding against a fluctuating market price, which is generally viewed as being random (Ben-Yehuda et al. [2011]).

Spot instances have proven to be a cost-effective resource for applications that are: able to cope with increased amounts of volatility; experience spikes in demand that exceed current capacity; or require low fixed costs. For example, "with the click of a button," CycleComputing built a 50,000 core cluster using Amazon's spot instances for \$650 per hour (Cycles [2012]).

2.1.4.1 Bidding

Each Spot instance type has different availability zones in which it is available, and each availability zone has an associated time-varying market price for that instance type. To lease a Spot instance, a user places a request for a specific instance type in a specific availability zone, and provides a bid for that instance. Not all instance types are available as Spot instances in all availability zones, however; Table 2.2 lists the availability zones within the region *us-east-1* where each Spot instance type is available.

After placing the request, if the user's bid is above the current market price for that instance type in that availability zone, the user gains access to the instance and, at the start of each hour block, pays the most recent market price. If the bid is below the market price, the instance becomes available to the user if and when the market price falls below this bid. Bids equal to the market price have been stated by Amazon to

Table 2.1: EC2 Instance Type Comparison (Linux OS)

Instance Type	On-Demand Price	EC2 Compute Units	Memory Size
m1.small	$0.065/hr	1.0	1.70 GB
m1.medium	$0.130/hr	2.0	3.75 GB
m1.large	$0.260/hr	4.0	7.50 GB
m1.xlarge	$0.520/hr	8.0	15.00 GB
m2.xlarge	$0.450/hr	6.5	17.10 GB
m2.2xlarge	$0.900/hr	13.0	34.20 GB
m2.4xlarge	$1.800/hr	26.0	68.40 GB
m3.xlarge	$0.580/hr	13.0	15.00 GB
m3.2xlarge	$1.160/hr	26.0	30.00 GB
c1.medium	$0.165/hr	5.0	1.70 GB
c1.xlarge	$0.660/hr	20.0	7.00 GB
cc1.4xlarge	$1.300/hr	33.5	23.00 GB
cc2.8xlarge	$2.400/hr	88.0	60.50 GB

not necessarily result in acquisition or termination, depending, among other factors, on the number of requests at that time (Amazon [2013c]). An example of market prices for a Spot instance of type *m1.large* in all availability zones in region *us-east-1*, with equivalent On-Demand price of $0.26/hr (see Table 2.1), is given in Figure 2.3.[1] Market prices for Spot instances are generally characterized by periods of stable market prices, interspersed with short price spikes.

2.1.4.2 Termination

A characteristic unique to Spot instances is the possibility of forced termination of the instance. If the market price for a Spot instance rises above the bid chosen by the user, the instance is automatically terminated by Amazon and the user is granted a refund for the last hour, resulting in a free partial hour of use. When Amazon terminates an instance, it does so by running shutdown scripts such as those in *etc/rc0.d* for Linux instances, and any unsaved progress for running tasks is lost. The refund for the last hour of use when a Spot instance is early-terminated by Amazon makes it beneficial to only terminate a Spot instance just before the hour block is over. Waiting for the full hour block increases the possibility of early-termination by Amazon and thus the possibility of a free partial hour of computation.

[1]Note that availability zones are specific to each user (e.g., one user's *us-east-1a* may be another's *us-east-1c*).

Table 2.2: An example of Spot Instance availability.

Instance Type	**Availability Zone**				
	us-east-1a	us-east-1b	us-east-1c	us-east-1d	us-east-1e
m1.small	✓	✓	✓	✓	✓
m1.medium	✓		✓	✓	✓
m1.large	✓	✓	✓	✓	✓
m1.xlarge	✓	✓	✓	✓	✓
m2.xlarge	✓		✓	✓	✓
m2.2xlarge	✓		✓	✓	✓
m2.4xlarge	✓		✓	✓	✓
m3.xlarge			✓	✓	✓
m3.2xlarge			✓	✓	✓
c1.medium	✓	✓	✓	✓	✓
c1.xlarge	✓	✓	✓	✓	✓
cc1.4xlarge	✓		✓	✓	
cc2.8xlarge			✓	✓	✓

Although forced termination of Spot instances comes without warning, research by Liu [2011] has suggested that it is possible to modify the shutdown scripts on a Spot instance to allow up to two extra minutes before forced termination. However, in this work we will assume that once the market price rises above the bid, the instance is terminated. Additionally, although Amazon has stated that bids exactly equal to the market price may or may not immediately be granted access (or be terminated), depending on the number of requests at that time, in this work we will assume that bids equal to the market price operate the same as those above, so that instances are leased if the bid equals the market price and terminated if the market prices rises above the bid. As we demonstrate within our frameworks, such an assumption can be easily eliminated with only a minor alteration of the bidding strategies presented for each framework.

2.1.4.3 Market Diversity

It is possible to exploit Amazon's various instance types and availability zones to achieve lower volatility, through the reduction of early-termination rates, and lower total costs. For example, Figure 2.4 illustrates the lifecycle of several Spot instances of type *m1.large* in region *us-east-1*, which are assigned to run tasks at the same time. Here, (1) and (2)

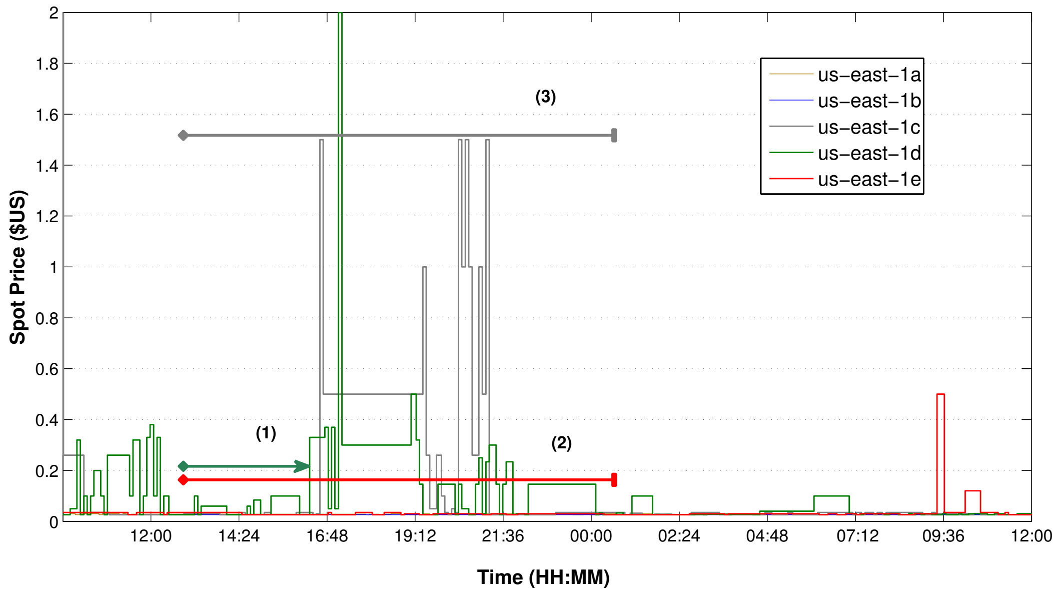


Figure 2.4: Spot prices over a 24 hour period for instance type *m1.large*, running a Linux OS, with all availability zones in region *us-east-1a*. (1), (2), and (3) represent instances, begun at the same time, with different bids and availability zones.

represent instances with bid $b = \$0.20/hr$, leased in *us-east-1d* and *us-east-1e*, respectively, and (3) represents an instance leased in *us-east-1c* with bid $b = \$1.51/hr$. Instance (1) is terminated before completion when the market price rises above $\$0.20/hr$, and, although both instance (2) and instance (3) successfully complete the task, instance (2) will do so while incurring significantly lower cost than (3) due to the lower and more stable market prices in zone *us-east-1e* during this time period.

In addition to the market diversity among availability zones presented in Figure 2.4, it is also possible to increase the size of the search space, and thus the possibility of locating more reliable, lower cost markets, by searching among other suitable instance types. For requests that require a minimum instance type to be available for a given time, searching among instance types greater, in some criteria, than that requested may offer a more reliable alternative. On the other hand, as examined in RAMC-DC, if the type of instance influences the execution time of the job, larger and smaller instances may be examined via the tradeoff of higher (lower) hourly costs vs lower (higher) execution time provided by each instance type.

2.1.5 Spot vs. On-Demand

Spot instances differ in many respects from On-Demand instances, especially in regards to pricing, availability, and volatility. Because of this difference, Spot and On-Demand instances may be used to in different circumstances and to meet different needs.

2.1.5.1 Cost vs. Volatility

When seeking to requisition an instance from Amazon EC2, the choice between leasing a Spot instance or an On-Demand instance corresponds to a tradeoff between cost and reliability. Although evaluation, bidding, and checkpointing strategies for Spot instances can help achieve user-defined reliability and availability constraints up to a certain level, several unavoidable problems remain. For example, the current market price for the Spot instance may be higher than the equivalent On-Demand instance, or the corresponding total cost for a Spot instance with a high bid may potentially be far higher than simply leasing an On-Demand instance (known as overbidding). In addition, Spot market price spikes have been shown to be random (Ben-Yehuda et al. [2011]), and thus can rise far higher than previously witnessed in the market price history.

A partial solution to the problem of imperfect fault-tolerance and fault-reduction when using Spot instances lies in the incorporation of On-Demand instances as well. Supplementing Spot instances with On-Demand instances can help reduce both the total cost of an application, as well as the associated early-termination rates (and thus the inherent volatility) when using only Spot instances. As discussed previously, an example of such supplementation in practice can be seen in the content sharing service Pinterest. Pinterest uses a 50-50 mixture of Spot and On-Demand instances to handle elastic load, with On-Demand instances replacing Spot instances during Spot price spikes (High Scalability [2012]). While doing this, and automatically shutting down unneeded instances, Pinterest has been able to reduce costs from \$54/hour to \$20/hour, thus achieving savings of 68% of their hourly cost.

2.1.5.2 Acquisition Times

An interesting difference between Spot instances and On-Demand instances lies in instance acquisition time: the time between when the user submits the request and when the instance is available to the user. Research by Mao et al. has shown that instance acquisition time for On-Demand instances is generally significantly lower than that of

Spot instances (Mao and Humphrey [2012]). Indeed, according to Mao et al., Spot instances generally take around 400 seconds before there are available to the user after the initial request is made, while On-Demand instances take around 100 seconds. Mao et al. also demonstrate that acquisition times vary greatly among providers as well, with Microsoft Azure far surpassing Amazon EC2 in average instance acquisition time. In addition, when leasing Spot instances, bidding the market price has been stated (as described above) to not necessarily guarantee instance acquisition.

Due to the large difference in acquisition times between instance purchasing options, such times will generally need to be taken into account when instances are frequently leased and released, and may provide another factor in the choice of whether to use an On-Demand or Spot instance for a particular application. The impact of these acquisition time differences may be decreased by employing predictive load forecasting methods; however, such predictive measures may also constrain the dynamicity of the approach.

2.1.6 Global Expansion and Price Reduction

Over the course of the research presented in this thesis, the number of regions in which instances have been offered, as well as the number of availability zones within each region, have been consistently expanded. Furthermore, prices for On-Demand instances and Spot instances have been regularly reduced as Amazon's datacenters expand and broader economies of scale are leveraged. Indeed, at the time of writing, prices for M1, M2, M3, and C1 On-Demand instances have been further reduced by an average of 10-20%, making them even better suited for supplementation (see table 2.3). The number of regions and availability zones in which M3 instance types are offered has also been expanded to encompass additional locations across the globe (recently including Sydney), and the data transfer prices between regions has been reduced, while data transfer prices between availability zones have been eliminated entirely.

New types of instances are also being offered over time. Examples of recent additions include the High I/O Quadruple Extra Large Instance, designed to provide very high instance storage I/O performance, and the High Storage instances, designed for applications that require instances with large amounts of disk and memory storage and high sequential I/O performance (Amazon [2013d]). In addition, Amazon now offers the option to launch instances with Elastic-Block-Store (EBS) optimization.

Instance Type	**On-Demand Price Change**
m1.small	-7.7%
m1.medium	-7.7%
m1.large	-7.7%
m1.xlarge	-7.7%
m2.xlarge	-8.9%
m2.2xlarge	-8.9%
m2.4xlarge	-8.9%
m3.xlarge	-13.8%
m3.2xlarge	-13.8%
c1.medium	-12.1%
c1.xlarge	-12.1%
cc1.4xlarge	0%
cc2.8xlarge	0%

Table 2.3: EC2 Price Reduction Comparison (Linux OS) after February 1$^{\text{st}}$, 2013.

2.2 Literature Review

Over the past several years, cloud computing has emerged as a means to acquire vast On-Demand resources. Many studies have shown the feasibility of using infrastructure-as-a-service cloud providers, such as Amazon (Amazon [2013d]), for applications including High-Performance Computing (HPC) (e.g. Evangelinos and Hill [2008], Walker [2008], Palankar et al. [2008], Youseff et al. [2008], Jackson et al. [2010]). A new way to acquire cloud computing resources is by bidding on virtual machines (VMs) known as Spot instances. As discussed earlier, Spot instances are VMs that are obtained by bidding on unused capacity in data centers, and are terminated when the user's bid falls above a market price determined by the cloud provider. Recent research has shown that Spot instances, like those available from Amazon (Amazon [2013c]), can increase further the effectiveness of acquired instances by decreasing the monetary cost of particular computing applications, especially when utilizing available Spot price history in conjunction with various fault-tolerant and price-predictive schemes.

Previous research on Spot instances covers several key aspects, including:

1. Statistical analysis of Spot prices & bidding strategies

2. Fault-tolerance strategies when using Spot instances

3. Scheduling & resource allocation when using Spot instances

In addition, profit-maximization when incorporating resources leased from IaaS providers has also seen much research, although little has been done with the incorporation of Spot instances.

Although many of the studies discussed in this chapter span multiple aspects, each will be grouped in terms of its relation to the research presented in this thesis.

2.2.1 Statistical Analysis of Spot Prices & Bidding Strategies

Although the stochastic nature of Spot instances implies almost all research on them includes some statistical analysis of their price and time dynamics, this section presents research work which focuses solely on analyzing the statistical characteristics of Spot instances.

Javadi et al. [2011] used one year of Spot price traces to model the time dynamics of the Spot price for all instance types and regions, using a mixture of Gaussian distributions with 3 components. The model was calibrated using the price traces, owing to a shift in Amazon's pricing strategy in mid-July 2010, and validated using CloudSim (Calheiros et al. [2011]) and workload traces from the LCG Grid (Iosup et al. [2008]). A high bid price (e.g. that of an On-Demand instance) was assumed for all simulations. Their results showed that the model predicted the mean total price of running tasks on Spot instances with a small maximum relative error of less than 3%, and with the instance type m1.small presenting the largest error.

Although the authors managed to predict accurately the time dynamics of Spot prices, they used a static, high bid price. Recent price traces show that bidding the On-Demand price often generates high costs, and the model proposed might miss many characteristics of the time dynamics corresponding to lower bids (Amazon Web Services).

Ben-Yehuda et al. [2011] analyzed Amazon's Spot price traces for all instance types and regions, using both Linux and Microsoft Windows operating systems, to determine how prices are set and to create a model that sets prices in the same manner. The availability of a declared price, D, was defined as

$$availability^{F}_{b\to e}(D) = \frac{T^{F}_{b\to e}(D)}{T_e - T_b},$$

where F is a Spot price trace file, T_b and T_e are the beginning and end of a time interval within F, and $T^{F}_{b\to e}(D)$ is the time between T_b and T_e during which the Spot price was lower than or equal to D. This availability was analyzed because it represents the probability of an instance being launched at a random time in the interval $[T_b, T_e]$. For each instance type and region, the authors claim that normalized graphs of price vs. availability share the same characteristic shape, with linearly increasing prices in a band between an artificial dynamic reserve price and a market-drive ceiling price. Further, they were able to match an AR(1) pricing algorithm, $\Delta_i = -a_1\Delta_{i-1} + \varepsilon(\sigma)$, with $a_1 = 0.7$ and $\sigma = 0.39(F-C)$, where F and C are the floor and ceiling prices, for all instances except m1.small, which had $a_1 = 0.5$ and $\sigma = 0.5(F-C)$.

The claim that Spot prices are artificially generated is significant as it contradicts other research papers that assume Amazon's Spot prices are entirely market-driven. Specifically, it implies that several research findings, such as those of Zhang et al. [2011] and Chen et al. [2011] below, are based on false assumptions. The findings also imply that sampling from the Spot price history is necessary to discover optimal bid prices for jobs. In addition, the authors identify three contiguous epochs in Amazon's price traces, characterized by different reserve and ceiling prices, inter-price times, etc. These epochs are important when using Spot price history to train models as the traces may cross epochs, making identification of transitions between epochs important.

Another example of research which mistakenly relies on the market-driven assumption is that by Shang et al. [2010], who present a knowledge-based continuous double auction framework for cloud resources. The authors focus on the Spot market as a one-sided auction market with a futures market in the form of On-Demand instances, and present a model that aims to determine a mutually agreed upon price between the cloud provider and the cloud user. The assumptions made in Shang et al. [2010] are made questionable by the results in Ben-Yehuda et al. [2011] and the experiments conducted are not fully explained.

Several papers examining Spot instances have specified trace-based bidding strategies when using them. Andrzejak et al. [2010] present bidding strategies using Amazon's Spot price traces to place bids based on the execution time of an instance for jobs requiring 1,000 minutes of execution that were designed to satisfy SLA constraints. Additionally, Song et al. [2012] and Zafer et al. [2012] developed optimal bidding strategies in Spot markets both from a client's and a broker's perspective. From the client's perspective, Zafer et al. designed a dynamic bidding policy (DBA) to minimize the total cost of a parallel or serial job within a deadline. Simulations run using real and synthetically generated Spot prices, and a serial job with 100 hours of computation time, show that DBA outperforms both random bidding and average-price bidding. From the broker's perspective, Song et al. develop a profit aware dynamic bidding algorithm (PADB) that maximizes the time average profit of the broker.

The bidding and evaluation strategies in this paper differ from those developed in Zafer et al. [2012], Andrzejak et al. [2010], and others, by allowing a trade-off between deadline breaches and total cost, adjusting the evaluation strategy depending on the execution time of a job, comparing the price dynamics across multiple availability zones, and incorporating the cost to suspend and resume an instance. Furthermore, when handling jobs with low execution times (e.g. under an hour), it becomes prudent to acquire instances with bids independent of the initial job executed on the instance so as to allow other jobs to fill up idle hour blocks.

2.2.2 Fault Tolerance

One of the main drawbacks of Spot instances is their inherent volatility. Because of this volatility, fault tolerant mechanisms are essential in many applications that utilize Spot instances. Although much of the research on Spot instances employs fault tolerance techniques, this section examines research which focuses on proposing and analyzing such techniques.

Checkpointing has often been proposed as a possible means to achieve fault tolerance when using Spot instances, and in failure-prone environments in general. Yi, Andrzejak, and Kondo (Andrzejak et al. [2010], Yi et al. [2010], Yi et al. [2011]) have authored several studies of fault-tolerance when using Amazon's Spot instances In Andrzejak et al. [2010], Spot price traces were used to create a model that attempts to minimize monetary costs while satisfying a Service Level Agreement (SLA). Users submit jobs and specify optional parameters, such as deadline and budgetary constraints, and the model then determines optimal bid prices and instance types while implementing an hourly checkpointing strategy. Simulations were run using exemplary workloads from the BOINC Catalog (BOINC [2011]) and Grid Workload Archive (Iosup et al. [2008]). Results from these simulations showed that higher bid prices, slightly lower confidence values, and higher budgets all helped decrease execution times. Additionally, the hourly checkpointing strategy is efficient for higher bid prices and reasonable confidence levels, and using larger instances can help reduce costs by up to 60%.

In Yi et al. [2010] and Yi et al. [2011], several adaptive checkpointing schemes are implemented to reduce the impact of out-of-bid situations, and migration is used to help improve task completion times. A fixed bid price is assumed, and four checkpointing schemes were compared: hourly checkpointing (HOUR); rising edge-driven checkpointing (EDGE); basic adaptive checkpointing (A); and current-price based adaptive checkpointing (C). In addition, three migration heuristics; lowest price, lowest failure rate, and highest failure rate, also are examined.

Experimental results in Yi et al. [2011] were obtained by running 100 simulations, assuming a total work of 1,000 minutes for each job, a known checkpointing cost, and variable price history windows. The results from these simulations showed that current-price based adaptive checkpointing, with a trace window of 112 days and lowest price migration, performed best in task completion and monetary cost. Additionally, work migration helped reduce the overall execution time by a factor of 2.5, although it slightly increased the cost of execution. These works use traces from a period of time in which Spot prices fluctuated much more frequently; current prices may not require very frequent checkpointing and simply adjusting the bid can prove to have better results.

Liu [2011] illustrates how the modification of shutdown scripts on VMs, such as /etc/rc0.d in Linux, can issue a signal to a MapReduce process so that states can be saved when necessary. Tests on real Spot instances showed that Amazon's hypervisor waits up to a maximum of two minutes for the shutdown scripts to finish before terminating the instance. Liu states that this is sufficient time for his implementation to save necessary states, with the shutdown window providing enough time to flush at least ”tens of megabytes of data” (p. 4).

Bougeret et al. [2011] analyze checkpointing strategies in an environment that suffers from processor failures, proving that periodic checkpointing times are optimal in order to minimize the makespan of either a parallel or sequential job with Exponential distributions of failure inter-arrival times. The authors claim that simulations run using their dynamic-programming algorithm with Exponential distributions, Weibull distributions, and real-world logs of failure inter-arrival times prove that such checkpointing strategies outperform all other solutions. Although the failure rates analyzed by Bougeret et al. are far lower than Spot instances have historically experienced (the real world logs used saw an average of 1 failure per day), in light of the more stable Spot prices offered by Amazon, the solutions provided in this paper may prove valuable for future work.

In the approach presented in this thesis, several rudimentary checkpointing strategies are used, but the proposed framework relies mainly on the prevention of early terminations achieved by being able to search across instance types and availability zones, and through specifying confidence levels when calculating bids and evaluating instances. In many cases, checkpointing allows only slight differences in total cost and deadline breaches per combination of parameters in the model while the largest difference lies in varying the parameters themselves.

2.2.3 Scheduling & Resource Allocation

A large portion of the research on Spot instances has focused on the problem of optimal resource allocation, both from the provider's side and from the client's side, with the aim of maximizing profit and decreasing failure rates and task completion time. To develop such strategies, statistically analyzing Spot prices and providing some fault tolerance scheme has been proven necessary to avoid the time and monetary costs incurred from out-of-bid situations, as evidenced below.

Zhang et al. [2011] address the problem of maximizing total revenue for a cloud provider (specifically Amazon), by dynamically allocating resources to each Spot market, given that demand for instance types fluctuates over time. Contrary to the findings presented in Ben-Yehuda et al. [2011], they assumed Spot price traces were indicative of client request volumes and bids. Their solution consists of dynamically scheduling and consolidating resources, based on demand predicted from Spot price traces using a simple autoregressive model. Two pricing schemes are analyzed: a fixed pricing scheme, where the price of a VM type does not vary with supply and demand; and a uniform pricing scheme, with runtime-adjustable prices. The first pricing scheme is modeled as a multiple knapsack problem (MKP), and the second is modeled using in the same way as the first pricing scheme, but with prices determined by an estimated demand curve.

The authors ran experiments were using CloudSim Calheiros et al. [2011], emulating a 1000-machine data center with 8 of the instance types available in Amazon EC2. Results showed that the dynamic allocation policy performed better than the static allocation policy, increasing average revenue by around 17%, with more significant gains resulting from dynamic demand patterns.

Mattess, Vecchiola, and Buyya (Mattess et al. [2010]) create resource provisioning policies that combined Spot instances and On-Demand instances to extend temporarily a local cluster. Among the policies examined are algorithms that only request On-Demand instances, request Spot instances based on whether the Spot price is above or below the maximum bid, or only use Spot instances. Using Amazon's Spot price traces, as well as traces consisting of Bag-Of-Task applications from the Grid Workload Archive, the authors found that using only On-Demand instances can reduce the Total Breach Time (the sum of the time tasks have spent in the queue beyond a Maximum Queue Time) and the Average Queue Time (the average of the time spent in a queue) by 2 and 3 orders of magnitude, respectively, with a total cost of only \$10,000 over the course of two years. Furthermore, incorporating Spot prices with a high bid price can further reduce the cost by half compared to using only On-Demand instances. The authors also observe, similarly to Ben-Yehuda et al. [2011], that there exists a band within which Spot prices are usually contained. Increasing bid prices to be above this band reduced mean costs by at least 20%.

Voorsluys, Garg, and Buyya (Voorsluys et al. [2011]) propose a system architecture that creates clusters of Spot instances, a resource allocation strategy that runs workloads on this cluster, and mechanisms to provide runtime estimates. The proposed resource provisioning and scheduling policy, BROKER, allocates jobs to VMs based on runtime estimates and the availability of idle instances. Experimental results, obtained from simulations on the CloudSim framework using embarrassingly parallel tasks in a workload from the LHC Grid at CERN, show a cost decrease of up to 60% compared to On-Demand instances, with slightly over-estimated runtimes providing the best results.

Voorsluys and Buyya [2012] also propose a resource provisioning policy, extending their work in Voorsluys et al. [2011], that encompasses two novel fault tolerance techniques aimed at decreasing the volatility of a heterogeneous cluster composed of Spot instances, including migration of VM states across availability zones. Estimates of the execution time were made in a similar fashion to those in this work. Voorsluys et al. also introduce an urgency estimation factor that represents the maximum time a job can wait in order to maximize the likelihood of meeting a deadline. Results suggest that VM migration, coupled with an aggressive urgency estimation factor and low bidding strategy, yield the lowest cost, but a moderate urgency estimation factor resulted in the lowest deadline violations.

Rahman [2011] propose an approach to decrease the cost of Spot instances using financial option theory. In their research, On-Demand instances are utilized as options, and several online policies are evaluated using only Spot instances, both Spot and On-Demand instances, and only On-Demand instances. Experiments were performed with modified traces from Amazon and the authors note that only a small variation in the prices existed at that time. However, their experimental results show that utilizing Spot instances decreases prices.

Rahman [2011] also makes mention of the drawbacks of the included policies; namely, that their binomial model assumes the original data follows a log-normal distribution, which may not be correct for Amazon, and they rely on the assumption that cloud providers follow a binomial option valuation method.

Zhao et al. [2012] develop deterministic and stochastic resource rental planning models (DRRP and SRRP) to minimize costs when running elastic computations on Spot instances. They show that the stochastic model is far superior, given the pricing uncertainty of Spot instances. The authors note that price approximation through prediction is insufficient, as discussed previously, and therefore attempt to reduce uncertainty by empirically estimating the probability distribution of the actual prices, determining the expected cost for different types of instances, and calculating the likelihood of an out-of-bid event.

Zhao et al. also consider three instance types and evaluate the performance of both SRRP and DRRP. Using a metric that determines the overpay amount (the increase in cost relative the the ideal case), DRRP is shown to perform well when using On-Demand instances but lags behind SRRP when used on Spot instances. DRRP is shown to decrease the overpay amount by up to 50% compared to using no planning strategy, and SRRP reduces that amount by nearly 50% again.

Taifi et al. [2011] describe a toolkit named SpotMPI that is used to run MPI applications on Spot instances by providing optimal checkpointing intervals and restarting of applications after out-of-bid situations through calculations of the density of out-of-bid failures from price history. Experimental results indicate that the communication overhead between Spot instances becomes highly detrimental as the number of instances is increased and, although higher bids provide better performance, they are coupled with a decrease in cost-effectiveness.

Chohan et al. [2010] and Liu [2011] authored research analyzing the efficacy of using Spot instances for MapReduce workflows. In Chohan et al. [2010], the authors model Spot instance lifetimes using Markov Chains and Amazon's Spot price history, with the edges of the Markov Chain representing the probability of price changes per hour. The n-step probability was then calculated as

$$P(i,b,n) = \sum_{j \notin B} M_{ij} P(j,b,n-1),$$

with starting market price, i, bid price, b, set of prices over b, B, and probability matrix of a price point from i to j, M_{ij}. The authors then calculated the expected lifetime of a Spot instance as

$$E(l) = \sum_{n=1}^{\tau} nP(i,b,n),$$

where τ is the max runtime. Simulations run using Spot instances as accelerators in MapReduce jobs indicated that Spot instances can help speed up applications, but the cost of losing a Spot instance (via termination by Amazon) can slow the application down do the point where using only On-Demand instances is more efficient. The authors recommend a fault-tolerant mechanism to deal with such a loss.

Liu [2011], as mentioned above, describes a new MapReduce implementation, Spot Cloud MapReduce, an extension of Cloud MapReduce (Liu and Orban [2011]), which is designed to deal with the fault tolerance inherent in Spot instances. One difference between Cloud MapReduce and Spot Cloud MapReduce is the modification of shutdown scripts on VMs as described earlier. Liu runs simulations using Amazon's Spot price traces and showed that Spot Cloud MapReduce costs are significantly less than Hadoop, as bids for Hadoop must be higher to prevent termination of the master node; but that completion times are dependent upon the time spent waiting for prices to drop below the user's bid.

Liu presents several significant findings. First, is that Amazon's hypervisor waits for up to two minutes for shutdown scripts to execute before terminating the instance. This can be used to advantage when designing checkpointing schemes, but it is vulnerable to policy changes by Amazon. The second finding, that Spot instances are more cost efficient than Hadoop when using a static, low bid also is significant because waiting times can be significantly reduced using optimal-bidding strategies, implying that the effectiveness of Spot instances relative to Hadoop can be increased further.

Mazzucco and Dumas [2011] propose a bidding scheme and server allocation policies to optimize revenue earned by a Software-as-a-Service (SaaS) provider leasing infrastructure (specifically, Spot instances) from an Infrastructure-as-a-Service (IaaS) provider, while satisfying performance and availability guarantees. The authors' revenue maximization scheme relies on a price prediction model and a server allocation and admission control policy. The server allocation policy determines the number of servers to rent based on traffic estimates and using a hill climbing heuristic. The admission control policy rejects or accepts requests based on a threshold value based on the number of servers. Experimental results determined that revenue is far greater (by almost 100%) when using Spot instances instead of On-Demand instances.

Autocorrelation is used to confirm that there is almost no correlation between different Spot prices, in line with the results given in Ben-Yehuda et al. [2011]. Therefore, the authors use a normal approximation to model the distribution of prices, with prices determined by linear regression if the autocorrelation function is above 0.4, and the quantile function of the normal distribution otherwise. To circumvent out-of-bid situations (where the Spot price is higher than the bid), the bid price is increased by 40% at each interval.

2.2.4 Profit Maximization

There have been many studies aimed at maximizing profit when using IaaS providers, including several focusing on the exploitation of inherent heterogeneity. Farley et al. [2012] seek to increase the payoff from leasing instances in Amazon EC2 by exploiting heterogeneity through hardware variations in the same instance type. Tsakalozos et al. [2011] introduce an approach that attempts to maximize per-user profit for a cloud provider, and quality-of-service received for a user. To do so, Tsakalozos et al. seek to specify a time-varying amount of of virtual machines a user should lease from a cloud provider, given a fixed budget, in order to achieve a desired response time for the user's applications.

Popovici and Wilkes [2005] develop scheduling policies designed to independently maximize the per-job profit of a service provider that rents uncertain resources with some associated price, in a similar fashion to Cloud providers, while also handling the problem of the dynamic acquisition of resources. Lee et al. Lee et al. [2010] address the problem of service request scheduling in cloud computing systems by presenting two sets of profit-driven service request scheduling algorithms, as well as a new pricing model that incorporates processor-sharing.

Chen et al. [2011] develop a new utility model to measure customer satisfaction in the cloud and the benefits gained from providing and utilizing heterogeneous resources, to determine the tradeoffs between profit and satisfaction, and to introduce two scheduling algorithms utilizing these tradeoffs to bid for Spot instances. The utility of a customer is quantified as

$$U(p,t) = U_0 - \alpha p - \beta t$$

where p is the service price, t is the response time, and U_0 is the maximum utility. The scheduling algorithms rely on maximizing profit while moving along an indifference curve, representing a level of customer satisfaction.

The two scheduling algorithms introduced by Chen et al. [2011] are: FirstFit-profit, where the service provider maximize profit while meeting a minimum level of customer satisfaction; and FirstFit-satisfaction, where the service provider maximizes the satisfaction while meeting a minimum profit level. The authors use three types of Spot instances from Amazon and use the price history as market clearing prices for auctions of VM instances, an assumption that is placed into question by the work in Ben-Yehuda et al. [2011]. Results showed that FirstFit-profit has a higher unit profit and lower instance number than approaches which utilize only a single types of instance. Furthermore, as the marginal rate of substitution between response time and service price, α/β, decreases, the number of instances with a larger number of cores increases, as shorter response times are sought, and unit profit decreases. The authors conclude that service profit and customer satisfaction are negatively correlated. The *FirstFit-profit* approach is similar in intent to the framework, RAMP, in that both RAMP and *FirstFit-profit* attempt to maximize profit while meeting a targeted level of instance reliability for customers.

Toosi et al. [2011] present resource provisioning policies that are aimed at increasing utilization and profit of Spot instances for an IaaS cloud provider. The proposed policies are: Non-Federated Totally In-house (NFTI), where, if feasible, providers terminate Spot instances with lower bids to accommodate On-Demand requests; Federation-Aware Outsourcing Oriented (FAOO) where requests are outsourced to other cloud providers if possible, otherwise Spot instances are terminated to accommodate the On-Demand request; and Federation-Aware Profit Oriented (FAPO), where the the decision to outsource On-Demand instances or terminate Spot instances is made based on the expected profit from each. Experimental results in Toosi et al. [2011], using the CloudSim framework and a workload generating model, show that FAPO has higher profit and less utilization (the ratio of the number of hours of VMs used by requests and the maximum number of hours of VM) than FAOO, with larger differences observable with higher loads, while both FAPO and FAOO have higher utilization than NFTI.

2.3 Assumptions

Using the works presented in this chapter, the following assumptions are made within this thesis regarding the nature of instances and Spot-market prices within Amazon EC2.

1. **Uniformity Within Instance Types:** Although results by Farley et al. [2012] indicate that some variation is encountered within the same instance type, as described in Section 2.2.4 above, in this thesis uniformity is assumed among different instance types and across availability zones. Thus, a job executing on an instance of type *i* will perform identically on a different instance of the same type, *i*.

2. **Randomness of Spot-Market Prices:** In concurrence with the research presented in Ben-Yehuda et al. [2011], Mazzucco and Dumas [2011], and Zafer et al. [2012], Spot market prices are not assumed to follow any distribution, and non-parametric methods are used to estimate market-price dynamics such as reliability, availability, and the probability of early-termination.

3. **Reliability of On-Demand Instances:** To provide a contrast to the volatility of Spot instances, On-Demand instances are assumed to have 100% availability over time, and thus an On-Demand instance will only ever be relinquished at the framework's request.

4. **No Instance Acquisition Time:** As mentioned in Chapter 1, due to the infrequency with which instances are leased, and the significant variation among providers, instance types, and purchasing options, we assume instances have no associated acquisition times (i.e., the time from the instance request to instance availability). In the case of requests without deadline constraints (as used in RAMP), such an assumption will have a negligible impact on the results.

Chapter 3

Cost Estimation

Each framework presented within this thesis requires a means to perform a cost-performance comparison of different instances when allocating resources. Thus, each framework is faced with the problem of approximating the cost of maintaining instance availability for some time period. Such an estimation facilitates a comparison of the potential cost-performance ratio of each instance and plays a crucial role in the evaluation of each instance-request or instance-job assignment.

3.1 Instance Models

Within this thesis, an instance, leased either as Spot or On-Demand, will be represented by the variable v. An instance v refers to either an *unleased* instance or an already *leased* instance as defined below.

- **Unleased Instances** are representations of new potential instances that have not yet been leased from Amazon. Unleased instances are examined when identifying potential resources to add to the existing pool and, when leased, are updated with a pointer to the newly leased EC2 instance. Cost efficiency, reliability, availability, etc., for unleased instances are examined when making the decision to lease a new instance.

- **Leased Instances** are representations of instances that have already been leased, and include a pointer to the existing EC2 instance. This pointer gives data critical to instance evaluation, such as the remaining time left in the hour block of the instance, the request/job currently executing on the instance, etc.

If v is unleased, it may be represented by the triple $\langle i,z,b \rangle$, where i is the instance type, z is the availability zone, and b is the bid (if the instance is leased as a Spot instance). The instance type i is an element of the set of all types, I, as described in Table 2.1, and z is an element of Z, the set of availability zones. Within this thesis, $Z = \{$*us-east-1a*, *us-east-1b*, *us-east-1c*, *us-east-1d*, *us-east-1e*$\}$, the set of availability zones in the region *us-east-1*.

Not all instance types are available as Spot instances in all zones: if v is a Spot instance, $z \in Z_i$, where Z_i is the set of all zones in which a Spot instance of type i is available. v is an On-Demand instance if and only if $b = \emptyset$; otherwise, v is a Spot instance and $b \in \mathbb{R}_+$. When an instance is leased, a pointer to the EC2 instance is added to v to retrieve data such as the status of the instance and the time remaining in the hour. Note that each Spot market is uniquely identified by the type-zone pair (i,z).

3.2 Analysis of Total Costs

Amazon's Spot price history has experienced many diverse pricing periods. In the past, Spot markets were generally characterized by highly fluctuating market prices with reasonably high price spikes, whereas over time there was a transition to markets with relatively stable market prices interspersed with short periods of very high prices. This change has necessarily reflected a shift in mechanisms designed to calculate the total cost of leasing a Spot instance for some period of time.

Approaches to approximating the cost of maintaing the lease of a Spot instance can generally be grouped into three categories: trace-based estimations that rely on random sampling; trace-based estimations that average the entire history; and current-market-price estimations. Trace-based estimations utilize Amazon's Spot price history to compute the total cost, whereas current-market-price estimations utilize only the current market price for that instance. As the average number of price-shifts within a given time period increases, cost estimations that take previous market prices into account will necessarily perform better than others, although such methods come with the cost of higher time-complexity and may be subject to recent changes in pricing policies. On the other hand, during periods in which market prices are relatively stable and the average number of price-shifts during a given time period drops closer to 0, Occam's Razor prevails and cost estimation techniques that incorporate only the current market price can give much better estimations of the cost and with much lower time-complexity.

The size of the bid at which the Spot instance was leased can also heavily influence the accuracy of these estimations. Higher bids allow for much more price-fluctuation without the instance being terminated, and thus both the estimation of the total cost for a Spot instance placed with such a bid, as well the true cost of this instance, can vary dramatically. For large bids, estimations that rely on random-sampling may be prone to missing market price spikes and thus may potentially underestimate the total cost. On the other hand, time-weighted average-price estimations may place too much weight on bid spikes during times of stable market prices.

3.3 Cost Estimation Methods

For each instance type and availability zone combination, (i,z), such that $i \in I$ and $z \in Z_i$, the algorithm has access to Amazon's spot price history for some past span of time: $H_{i,z} = \{(p_1,d_1),\ldots,(p_k = p_{mkt},d_k)\}$, where p_m is the price at time d_m, and p_k is the current market price. The following five methods to approximate the total cost of running a new Spot instance, $v = \langle i,z,b \rangle$, for $t \in \mathbb{R}_+$ hours are presented below and evaluated in the next section. Implementations in C++ are given in Section A.1.

1. *Market Price*: The estimated cost is calculated as the market price multiplied by the ceiling of the desired computation time:

$$\widehat{C}_{mkt}(t,v) = p_{mkt} \cdot \lceil t \rceil, \tag{3.1}$$

   where p_{mkt} is the current market price as discussed above.

2. *Average Price*: The estimated cost is determined using an average per-hour price calculated as the weighted sum of all previous market prices less than the bid over some window of the Spot price history, with each weight equal to the fraction of the time spent at each market price compared to the total time spent under the bid:

$$\widehat{C}_{avg}(t,v) = \frac{\sum\limits_{p_m \le b,\, m<k} p_i \cdot (d_{m+1} - d_m)}{\sum\limits_{p_m \le b,\, m<k} (d_{m+1} - d_m)} \cdot \lceil t \rceil. \tag{3.2}$$

3. *Monte Carlo*: The estimated cost is calculated using a nonparametric Monte Carlo estimate:
$$\widehat{C}_{mc}(t,v) = \frac{1}{|X_C|} \sum_{x \in X_C} C_x(t,v), \tag{3.3}$$
where X_C is a set of dates sampled uniformly over a past window of the Spot price history and $C_x(t,v)$ is the true cost of running the job at x if the job completes successfully, and is otherwise equal to $\widehat{C}_{mkt}(t,v)$.

4. *Market-Average*: If the runtime of the job is less than some parameter α, the estimated cost is determined using the *Market Price* estimate of the cost. Otherwise, the estimated cost is calculated as:
$$\widehat{C}_{ma\_\alpha}(t,v) = \widehat{C}_{mkt}(\lceil \alpha \rceil, v) + \widehat{C}_{avg}(t - \lceil \alpha \rceil, v). \tag{3.4}$$
Therefore, the estimated cost is the sum of the *Market Price* method for the first $\lceil \alpha \rceil$ hours and the *Average Price* method for the remaining time.

5. *Market-Monte Carlo*: As in 4) but the *Monte Carlo* estimate is used for the remaining time:
$$\widehat{C}_{mmc\_\alpha}(t,v) = \widehat{C}_{mkt}(\lceil \alpha \rceil, v) + \widehat{C}_{mc}(t - \lceil \alpha \rceil, v). \tag{3.5}$$

Estimated costs for on-demand instances are calculated as $\widehat{C}_{OD}(t,v) = \lceil t \rceil \cdot ODPrice_i$, where $ODPrice_i$ is the on-demand price for instance type i, as given in Table 2.1. Furthermore, in the case of VM state migration across availability zones, Amazon's rate of \$0.01/GB will be added to the total cost.

Note that these estimations are strictly for new instances. If an instance has been leased, $\widehat{C}(t,v)$ must be calculated as the total cost accounting for the fact that the remaining hour block on v has already been paid for. Thus, $\widehat{C}(t,v)$ approximates the cost using the time $t-h$, where h is the predicted time that will remain in the instance's hour block.

The above estimation methods were chosen to compare the incorporation of the market price, to determine the appropriate weight that should be given to this market price and to evaluate the difference in accuracy between a random sampling and an averaging approach. Increasing the weight of the market price in an estimation will generally lead to more accurate results if the inter-price time (the time between market price changes) is very high. When the inter-price time is high, the probability of a change in market price during execution is low, and so the market price first encountered is likely to be the only one encountered. Similarly, estimations incorporating the time-weighted average price will generally give a better estimation than those that rely on random sampling when price spikes are not infrequent, and thus an average price estimation will be less likely to incorporate outlier data. Random sampling, on the other hand, has a higher potential to avoid such outliers.

3.4 Evaluation of Estimations

To determine which cost estimation method achieves the lowest relative error, simulations are run using 20,000 randomly generated requests with desired availability time, t, uniformly sampled between 1 and 12 hours (desired availability times above 1 hour guarantee that the true cost is nonzero), and with desired instance and availability zone also uniformly sampled. We assume each job is allocated a new Spot instance of the requested type, and in the requested availability zone.

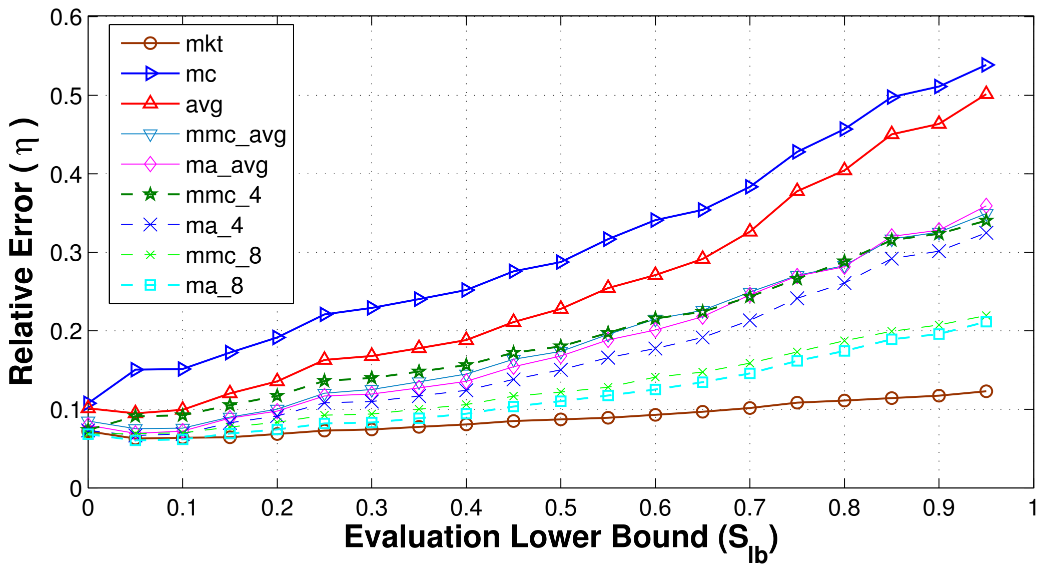


Figure 3.1: Percent Relative Error for Various Cost Estimation Methods. Traces Are From February-June, 2012.

Figures 3.1, 3.2, 3.3 illustrate the relative error, η, of each cost estimation method for successful jobs (that were not terminated early) using the minimum bid, greater than

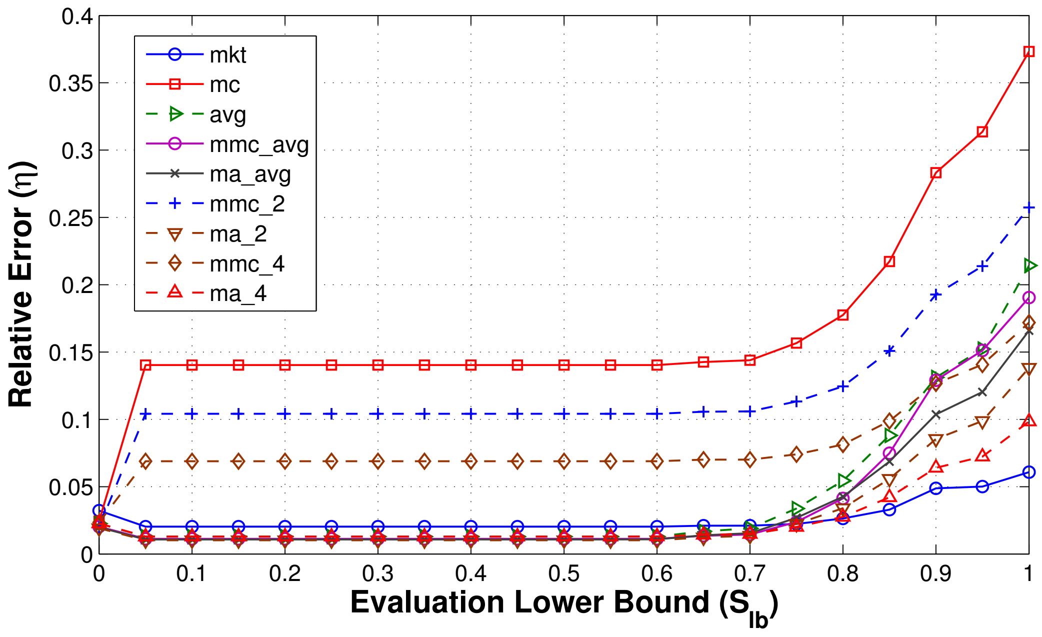


Figure 3.2: Percent Relative Error for Various Cost Estimation Methods. Traces Are From June-November, 2012.

or equal to the market price, such that the market price for that instance has lower than the bid for at least S_{lb} of the time over the past two months. Here, η is calculated as:

$$\eta = \frac{|C(t,v) - \widehat{C}(t,v)|}{C(t,v)}, \tag{3.6}$$

and the variable *avg* represents the average inter-price time of market prices under the bid (see Listings A.6, A.7). In the first figure, Spot price traces from the period between February and June, 2012, were used. In the second, Spot price traces from taken from the period between June and November, 2012. The period reflected in the first image generally had less frequent market price fluctuations than that used in the second.

When prices fluctuate less frequently, as seen in Figures 3.1, for $S_{lb} > 0.1$, *Market Price* estimation achieved the highest accuracy, with the distance between the next-closest estimate widening to as much as 0.09 as S_{lb} is increased. For $S_{lb} \leq 0.1$, Market-Average estimates yield the highest accuracy, and both Market-Average and Market-Monte Carlo perform slightly better than Market Price estimation with a relative increase in accuracy of 5.3% for mma_8 compared to *mkt*, from 0.072 to 0.068. As α is increased, Market-Average and Market-Monte Carlo both become more accurate due to the convergence to the Market-Price estimate.

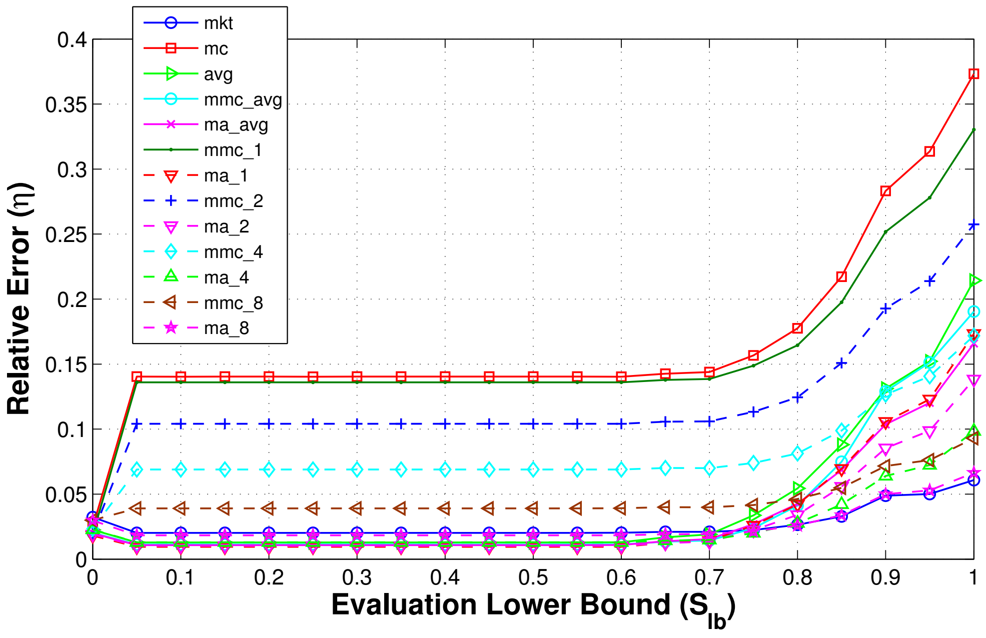


Figure 3.3: Percent Relative Error for More Cost Estimation Methods. Traces Are From June-November, 2012.

On the other hand, when prices fluctuate more often, for values of S_{lb} less than 0.7, Average, Market-Average and Market-Monte Carlo estimates of the cost perform the best, with Market-Average slightly more accurate, achieving relative errors of only around 0.015-0.025 each. A Market-Average estimation with $\alpha = 4$ hours achieves the highest accuracy for these values of S_{lb}, with lower values of α achieving higher relative errors. For higher values of S_{lb} (greater than 0.7), the simplest estimate, Market Price, performs the best, with other estimates quickly becoming more and more inaccurate as S_{lb} increases. As with older traces, Market-Average estimations outperform Market-Monte Carlo methods, especially at lower values of α.

Figure 3.3 illustrates the relative error for α = 1, 2, 4, 8, *avg*, and 0 in in the case of *mc* and *avg* estimations. For both *ma* and *mmc* estimations, and for $S_{lb} \leq 0.7$, $\alpha = 4$ hours attains the highest accuracy, with lower and higher values of alpha achieving lower accuracy. In addition, as α increases to 4 hours, the relative error of both methods drops in response. For $S_{lb} > 0.7$, however, the relative error drops as α is increased, and attained a minimum when $\alpha = \infty$ and thus the *mkt* method is employed.

The increasing disparity between cost estimates as S_{lb} increases reflects the fact that other cost estimate methods rely on the instance's potential bid. As S_{lb} is increased, the bid will also monotonically increase, allowing for a wider range of past market prices to be taken into account when calculating the average prices or the average costs. Since Spot prices exhibit periods of little fluctuation punctuated by large price spikes, using data from periods of different market prices in the estimation will be less indicative of the actual cost. When prices fluctuate quite often, for lower values of S_{lb}, the range of bids which satisfy the lower bound is constricted (when $S_{lb} = 0$, the bid will always be equal to the market price) and thus cost estimation methods utilizing past Spot prices will have a more accurate estimate. Depending on the frequency of price fluctuation, the value of S_{lb} at which trace-based estimations become more accurate will either increase or decrease if prices fluctuate more or less often, respectively. As expected, cost estimates that do not take into account the current market price tend to perform more poorly than others in each group.

Chapter 4

RAMP

In this chapter, RAMP's approach to instance acquisition is presented, including the evaluation of instance reliability and the location of bids designed to satisfy reliability constraints. Following this, the strategy for locating the instance which maximizes the expected profit for a given request is presented, including a simple means to approximate the total cost of fulfilling a request on a specific Spot instance. Simulations designed to evaluate RAMP's performance are then presented and results are discussed.

4.1 Overview

In this section, the model used for user-submitted requests is presented, and the problem faced by RAMP is formalized.

4.1.1 Request Models

To request an instance, a user submits a request $r = (i_r, t_r)$. Here, i_r is the minimum instance type required, chosen from the set, I, given in Table 2.1, and t_r is the desired number of full or partial hours, $t_r \in \mathbb{R}_+$ (specified at second granularity), for which the user requires access to the instance.

When submitting a request, the user is charged some fraction, $p \in [0,1]$, of the On-Demand price for the instance i_r. Rather than a per-hour price, for each request the user will instead pay:

$$R_r = t_r \cdot p \cdot ODprice(i_r), \tag{4.1}$$

where $ODprice(i)$ is the hourly price for the equivalent On-Demand instance of type i. Thus, R_r represents the *revenue* acquired by RAMP for request r. Such a pricing strategy allows the user very fine time-period granularity (on the order of per-second in this paper) when requesting instances. To help compensate for the volatile nature of Spot instances, in the event of early-termination of an allocated Spot instance the user is refunded the initial payment for their request plus a *penalty fraction*, $e \in [0,1]$, of this payment. Therefore, if the user is allocated a Spot instance which is subsequently early-terminated, the user is refunded $R_r + e \cdot R_r$. Other possible penalties are discussed in Section 4.3.1 but are not evaluated in this work.

4.1.2 Problem Formulation

RAMP seeks to fulfill requests from users specifying desired availability times for minimum instance types by leasing either Spot or On-Demand instances from Amazon EC2. The value i_r is used to construct I_r, the set of instance types *greater than or equal to* that requested. In the context of RAMP, instance type i_1 is *greater than or equal to* instance type i_2 if both the memory and the number of EC2 Compute Units of i_1 are greater than or equal to that of i_2. These two criteria were chosen for the sake of simplicity; other factors such as I/O performance, storage space, and Elastic Block Store (EBS) optimization may be compared instead of, or in addition to, these.

Therefore, given the minimum instance type required by the user, i_r, the set:

$$I_r = \{i \in I \mid i \geq i_r\} \tag{4.2}$$

represents the set of instance types RAMP will search when allocating an instance to the user. For example, if the user submits a request with $i_r =$ *c1.xlarge*, the set of instance types RAMP will search is $I_r = \{$*c1.xlarge*, *m3.2xlarge*, *m2.4xlarge*, *cc1.4xlarge*, *cc2.8xlarge*$\}$. Adding or removing constraints for an instance type to be greater than or equal to another instance type will, respectively, decrease or increase the size of I_r. Increasing the size of I_r will provide more options for the resource provisioner to search among, and therefore offers a higher chance of locating instances that offer lower market prices while still meeting reliability constraints.

Let $C(r,v)$ be the *true cost* of running an allocated instance v for either the full t_r hours, or until possible early-termination if the instance is a Spot instance. Here, v

can refer to either an already-leased instance or an unleased instance. Each instance v contains the instance's type i, bid (if a Spot instance) b, availability zone z, and, if already leased, both a pointer to the instance and the time remaining in the hour block $h \in (0,1)$. The availability zone z is chosen from the set of all possible availability zones, Z. At the time of this writing, there are five availability zones in region *us-east-1*, illustrated in Figures 2.3 and 2.4. In general, $C(r,v)$ will not be known *a priori* when using Spot instances and must instead be approximated at the time of evaluation.

The problem: The *true profit* acquired by RAMP for an instance is dependent upon outcome, $o(r,v)$, of using instance v to to fulfill request r. Thus, given the success (completion) or failure (termination) of the instance during the request, the profit is defined as:

$$\pi(r,v) = \begin{cases} \pi_s(r,v) & \text{if } o(r,v)\text{=}\textit{success} \\ \pi_f(r,v) & \text{if } o(r,v)\text{=}\textit{failure} \end{cases} \tag{4.3}$$

where

$$\pi_s(r,v) = R_r - C(r,v) \tag{4.4}$$

and

$$\pi_f(r,v) = -C(r,v) - e \cdot R_r \tag{4.5}$$

Therefore, RAMP faces the problem of locating, for each request, r, the instance, v^*, such that the profit of using v^* to satisfy r is maximal:

$$v^* = \underset{v \in V}{\arg\max}\ \pi(r,v), \tag{4.6}$$

where V is the set of instances such that the empirical probability of *success* is greater than or equal to some lower bound, S_{lb}.

Thus, to be considered as a possible candidate for allocation, an instance must be able to achieve *success* with probability greater than S_{lb}. However, given the imperfect *a priori* knowledge associated with using Spot instances, including the true probability of *success* and $failure$ occurring, comparisons of the profitability of each request-instance assignment will instead be made through a comparison of the *expected profit* of each possible allocation. The lifecycle of a request is given in Figure 4.1.

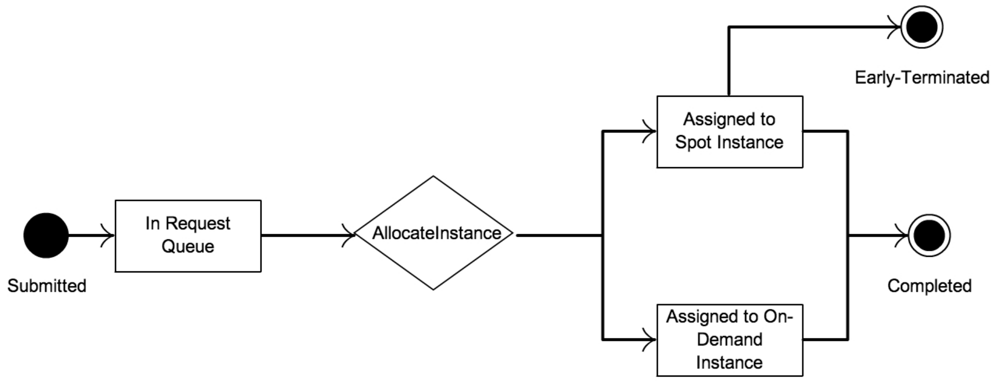


Figure 4.1: The lifecycle of a request.

4.2 Reliable Instance Acquisition

Estimating the reliability of a Spot instance, given the instance type, availability zone, and the bid at which the Spot instance is requested, can offer critical information about the susceptibility of the instance to early-termination during a specified time period. Additionally, such reliability estimates are necessary when determining the bid with which to request a Spot instance, while satisfying an upper bound on the instance's probability of termination. In this section, we present a method for calculating instance reliability, and a bidding strategy for leasing Spot instances that seeks to locate the minimum bid at which a Spot instance satisfies a lower bound on the reliability function.

4.2.1 Instance Reliability

To effectively utilize Spot instances, a means to estimate the probability of an instance being available for the duration of the user's request is required. To do so, RAMP estimates the probability of *success* by empirically determining the *reliability function* (tail distribution) of the instance availability time for the corresponding instance, $S(r,\nu) = S(r,\langle i,z,b\rangle)$ for $\nu = \langle i,z,b\rangle$. Therefore, this reliability function will be determined as $S(r,\nu) = P(T(\nu) \geq t_r)$, where $T(\nu)$ is a random variable representing the availability time of an instance of type i with bid b, leased in zone z, and $P(T(\nu) \geq t_r)$ is the empirical probability of *success* when using this instance.

Let X be a set of dates sampled uniformly from some window of the Spot price

Table 4.1: Frequently Used Variables (RAMP)

Variable	Description
t_r	The requested availability time.
i_r	The requested instance type.
p	The fraction of the On-Demand price charged.
e	The fraction of the payment given as penalty in the event of early-termination.
S_{lb}	Lower bound for the *reliability function*.
R_r	The revenue from the request.
C_r	The cost incurred by RAMP for the request.
I	The set of instance types (cf. Table 2.1).
Z	The set of availability zones.
i	Some instance type ($i \in I$).
z	Some availability zone ($z \in Z$).
b	The bid for a Spot instance.
v	An instance, either Spot or On-Demand, leased or unleased.

history for that instance. $S(r,v)$ can then be estimated using the nonparametric Kaplan-Meier Estimator:

$$S(r,v) = P(T(v) \geq t_r) = \prod_{t_x(v) \leq t_r} \frac{n_x(v) - 1}{n_x(v)}. \tag{4.7}$$

Here, $t_x(v)$ is the true time the instance was available (i.e. the step length) at time $x \in X$, and $n_x(v) = |\{y \in X \setminus \{x\} : t_y(v) \geq t_x(v)\}|$ is the number of samples with availability time greater than $t_x(v)$. If v is an On-Demand instance, we will assume that the reliability is equal to 1 for all requests.[1]

Although random sampling has the potential to miss certain events such as market price spikes, both random sampling and the Kaplan-Meier Estimator are used to estimate the reliability function due to previous research such as (Ben-Yehuda et al. [2011]) and (Mazzucco and Dumas [2011]). As discussed in Chapter 2.3, these studies suggest that Spot Market prices do not follow any particular distribution, but are instead randomly generated between a hidden market-determined upper bound, and a provider-determined lower bound.

[1]An implementation of $S(r,v)$ is given in B.1.

4.2.2 Bidding Strategy

Given the reliability, $S(r, v)$, of using an instance, v, to satisfy a request, r, RAMP leases a Spot instance by first locating a bid designed to guarantee that the instance is available to the user for t_r hours with empirical probability greater than or equal to some specified lower bound, S_{lb}. For a particular instance type and availability zone, i and z, we determine this bid as:

$$b(r,i,z) = \min_{b \geq p_{mkt}(i,z)} \{b \mid S(r, \langle i,z,b \rangle) \geq S_{lb}\}, \tag{4.8}$$

where $p_{mkt}(i,z)$ is the current Spot market price of instance type i in zone z. Thus, RAMP chooses the minimum bid that both satisfies the reliability function lower bound, and is greater than or equal to the current market price. When $S_{lb} = 0$, the bidding strategy presented here will simply find the lowest bid greater than or equal to the current market price. Thus, such a strategy will always choose the current market price for that instance type and availability zone as the bid. Note that, if we wish to avoid the situation, discussed in Section 2.1.4, where an instance may or may not start if the bid is exactly equal to the current market price, the inequality in the above equation may be made strict so that RAMP requires $b > p_{mkt}(i,z)$.

The empirical probability of successful completion is a monotonically increasing function of the bid price (decreasing the bid should never increase $S(r, \langle i,z,b \rangle)$). Hence, finding $b(r,i,z)$ can be implemented as a hill-climbing search between the current market price and the maximum market price seen over the Spot price history window.

The bidding strategy discussed in this section is used both to satisfy lower bounds on the probability of successful completion, and to reduce the possibility of maintaining job execution through one of the large price spikes frequently seen in the Spot price history. Although more complex than a bidding strategy that simply bids the greatest market price seen so far, the strategy presented here prevents such overbidding and can thus be critical for maintaining low costs. For example, if the market price for the chosen instance type and availability zone increased to \$1,000.00, the bidding strategy used here would, in general, provide a bid low enough to ensure that execution would terminate once the price spiked this high. If, on the other hand, a bidding strategy choosing the maximum bid within the Spot price history is used, it is possible that execution would continue through this price spike and thus result in high costs. In fact, Amazon has previously mentioned that overbidding for Spot instances is actually a cause of many Spot market price spikes (Amazon [2012]).

4.3 Profit Maximization

To simplify our approach, we assume that RAMP can meet all user demand when using Amazon EC2, and thus we will disregard the problem of scarcity of resources. Therefore, for each request, r, the problem in Equation 4.6 can be revised so that RAMP needs only to find the instance, v^*, which maximizes the *expected profit* of that request, independent of supply constraints:

$$v^* = \underset{v \in Idle \cup New}{\arg\max} \; E(\pi(r,v)), \tag{4.9}$$

where

$$Idle = \{v \in \text{idle instances} \mid S(r,v) \geq S_{lb} \wedge i \in I_r\}, \tag{4.10}$$

and

$$New = \{\langle i,z,b(r,i,z)\rangle, \langle i,z,\emptyset\rangle \mid (i,z) \in I_r \times Z\}. \tag{4.11}$$

The sets $Idle$ and New include both Spot and On-Demand instances (differentiated by whether $b = \emptyset$, as mentioned above), and New represents a set of potential instances which can be leased and allocated to the request. If the profit-maximizing instance, v^*, has no pointer to an existing leased instance, RAMP leases the instance matching v^*'s specifications before assigning the instance to the request.

4.3.1 Profit Estimation

Let $\widehat{C_s}(r,v)$ be the *estimated cost*, calculated at the time of evaluation, of successfully servicing the request with instance v, and let $\widehat{C_f}(r,v)$ be the estimated cost in the case of failure. Furthermore, in the case of *success* or *failure*, assume $\widehat{C_s}(r,v)$ and $\widehat{C_f}(r,v)$ differ absolutely from the true cost, $C(r,v)$, by some, generally small, amount $\varepsilon_s, \varepsilon_f \geq 0$. Then, given the true profit from the two possible outcomes discussed in Section 4.1, as well as the empirical probability of successful completion, $S(r,v)$, the *expected profit* of an assignment of instance ν to request r can be calculated as:

$$\begin{aligned}
E(\pi(r,\nu)) &= S(r,\nu)\cdot\pi_s(r,\nu) + (1-S(r,\nu))\cdot\pi_f(r,\nu) \\
&= S(r,\nu)\cdot(R_r - \widehat{C}_s(r,\nu)+\varepsilon_s) \\
&\quad -(1-S(r,\nu))\cdot(\widehat{C}_f(r,\nu)+\varepsilon_f + e\cdot R_r) \\
&\approx S(r,\nu)\cdot(R_r-\widehat{C}_s(r,\nu)) \\
&\quad -(1-S(r,\nu))\cdot(\widehat{C}_f(r,\nu)+e\cdot R_r)
\end{aligned} \tag{4.12}$$

The method to find the maximum profit instance is presented in Algorithm 1 (as stated above, we will assume $S(r,\nu)=1$ for On-Demand instances). Thus, for approximations that are close to the actual cost (i.e., for small values of ε), the maximization problem presented in Equation 4.9 can be represented as:

$$\nu^* = \underset{\nu\in Idle\cup New}{\arg\max}\; E(\pi(r,\nu)). \tag{4.13}$$

Note that profit in the case of termination can take many forms, depending on the application. An example not used in this paper may, in the case of *failure*, let $\pi(r,\nu,failure)$ be a linear function of the time, t, for which the instance was actually available to the user, e.g.:

$$\pi_f(r,\nu,t) = -C(r,\nu) - e\cdot(1-t/t_r)\cdot R_r \tag{4.14}$$

Alternatively, a priority-based payment strategy may be introduced to allow users to pay more for higher values of both the reliability lower bound, S_{lb}, of their instances, and the penalties paid to them in the event of early-termination. In the latter case, such a penalty may help to further balance out the high costs associated with high reliability levels, and provide lower costs to users with applications that are able to handle higher volatility.

4.3.2 Cost Approximation

In the case of *success*, we estimate the true cost simply as the current market price, $p_{mkt}(\nu)$, multiplied by the ceiling of the request length, t_r, minus any already-paid-for

Algorithm 1: MaxProfitInstance - Finding the instance that maximizes the expected profit

```
   Data: Request $r = (t_r, i_r)$
   Result: $v^*$, the instance that maximizes the expected profit
1  begin
2      $I_r \leftarrow \{i \in I \mid i \geq i_r\}$
3      $Idle \leftarrow \{v \in \text{idle instances} \mid S(r,v) \geq S_{lb} \wedge i \in I_r\}$
4      $New \leftarrow \{\langle i,z,b(r,i,z)\rangle, \langle i,z,\emptyset\rangle \mid (i,z) \in I_r \times Z\}$
5      $v^* \leftarrow \emptyset$, $\pi^* \leftarrow -\infty$
6      for $v \in Idle \cup New$ do
7          $\pi \leftarrow E(\pi(r,v))$
8          if $\pi > \pi^*$ then
9              $v^* \leftarrow v$, $\pi^* \leftarrow \pi$
10         end
11     end
12     return $v^*$
13 end
```

Algorithm 2: AllocateInstance - Allocating an instance to a request

```
   Data: Request $r = (t_r, i_r)$
   Result: $\langle r, v^*\rangle$, the request-instance allocation pair
1  begin
2      $v^* \leftarrow$ MaxProfitInstance($r$)
3      if $v^*$ is unleased then
4          if $v^*$ has no bid then
5              Lease an On-Demand instance with type $i$ in zone $z$, for $i,z \in v^*$
6          else
7              Lease a Spot instance with bid $b$, type $i$, and in zone $z$, for $i,z,b \in v^*$
8          end
9          Add a pointer to the corresponding EC2 instance to $v^*$
10     end
11     return $\langle r, v^*\rangle$
12 end
```

partial hour, h (as defined in Section 4.1.2):

$$\widehat{C_s}(r,v) = p_{mkt}(v) \cdot \lceil t_r - h \rceil. \tag{4.15}$$

In the case of *failure*, we construct X, a set of 10,000 random samples over the past 60 days of the Spot price history, and determine the subset $X_f \subseteq X$ such that if r was fulfilled using instance ν at time $x \in X_f$, the instance would have been terminated early by Amazon. Then we calculate the Monte-Carlo estimate of the average runtime in the event of failure, minus the remaining partial hour, and multiply this by the current market price. Since Amazon does not charge for the last partial hour before termination, we take the maximum of the floor of this value and 0 (no negative costs):

$$\widehat{C}_f(r,\nu) = p_{mkt}(\nu) \cdot \frac{\sum_{x \in X_f} max\{\lfloor t_x(\nu) - h \rfloor, 0\}}{|X_f|}. \tag{4.16}$$

Here, $t_x(\nu)$ is the true time the instance ν was available for (in hours), starting from time x.[2]

Figure 4.2 illustrates the percent relative error when using these methods to estimate the cost of running a Spot instance for the desired amount of time, and for different values of S_{lb}. Here, δ is calculated as:

$$\delta = \frac{|C(r,\nu) - \widehat{C}(r,\nu)|}{C(r,\nu)}, \tag{4.17}$$

where

$$\widehat{C}(r,\nu) = \begin{cases} \widehat{C}_s(r,\nu) & \text{if } o(r,\nu)\text{=}success \\ \widehat{C}_f(r,\nu) & \text{if } o(r,\nu)\text{=}failure \end{cases} \tag{4.18}$$

This evaluation was made using a set of 20,000 requests over a two month period, and encompasses only those requests with non-zero $C(r,\nu)$. For each request, the desired availability time is uniformly distributed between 1 and 12 hours, and the requested instance type is uniformly sampled from those in Table 2.1.

As evidenced by the figure, the percent error is very small when using these estimations. In fact, these methods attain a maximum error of only 6.3% of the actual cost when $S_{lb} = 1.0$, the reliability level where bids are at their highest (allowing for the most variation in the market price during the request). In addition, our estimation technique maintains an error of between 2.2% and 2.5% for $S_{lb} \in (0, 0.75]$. If the instance is an On-Demand instance, $p_{mkt}(\nu)$ is replaced by $ODprice(i)$ for $i \in \nu$, where $ODprice(i)$ is the On-Demand hourly price for an instance of type i, as discussed in Section 4.1 and depicted in Table 2.1. In the case of possible inter-region data transfers, the total cost can be adjusted using Amazon's rate of \$0.01/GB.

[2]An implementation of $\widehat{C}_f(r,\nu)$ is given in B.2.

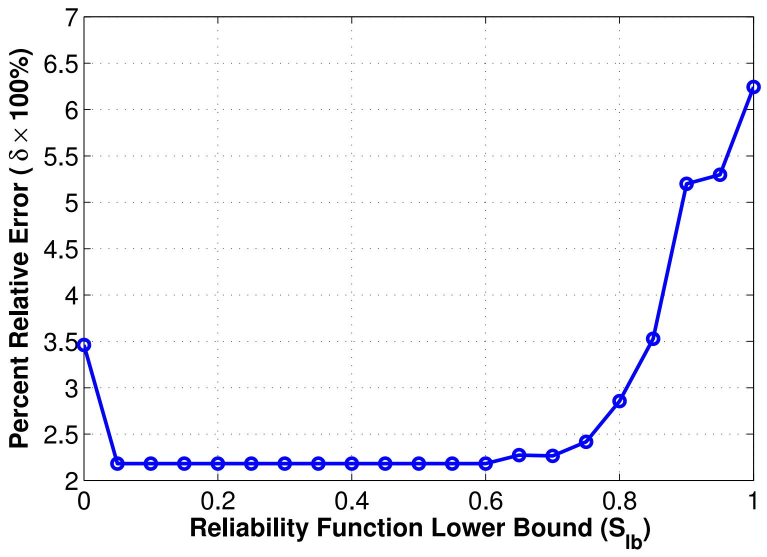


Figure 4.2: Percent relative error of the cost using Market Price cost estimation with the requests described in Section 4.3.2.

A drawback of using only the current market price to estimate the true cost lies in the fact that such an estimation does not incorporate any knowledge of previous market prices. Although, due to their highly-fluctuating nature, past Spot prices would have necessitated the use of, for example, a Monte Carlo estimate of past costs from an equivalent request, recent Spot price traces tend be characterized by more stable market prices and thus diminish the requirement for such an estimate.

4.3.3 Request Fulfillment

The complete procedure for finding and allocating the maximum profit instance for a user's request is presented in Algorithm 2. The maximum profit instance, v^*, either an idle or to-be-leased instance, is found using the method described in Algorithm 1 (line 2). If v^* contains no pointer to an existing instance, it is not yet leased. Thus, RAMP either leases a Spot instance of type i in zone z, if the the maximum profit instance has bid $b \neq \emptyset$ (line 7), or RAMP leases a new On-Demand instance of type i in zone z, if v^* contains no bid (i.e., $b = \emptyset$) (lines 4-5). The instance is then assigned to the request and made available to the user until either the instance is terminated, or the user's requested availability time has expired.

4.4 Evaluation Overview

In this section we first describe the experimental setup and evaluate the efficacy of RAMP as follows.

- We measure and analyze the achieved early-terminations rates versus the reliability function lower bound.
- We compare total profits when: charging the user different fractions of the On-Demand price; varying the penalty associated with early-termination; and varying the reliability function lower bound.
- We evaluate the differences in our approach against a version of RAMP that simply bids the market price for each instance, and determine that the any variation from such a strategy can significantly increase profit and successful completion rates.

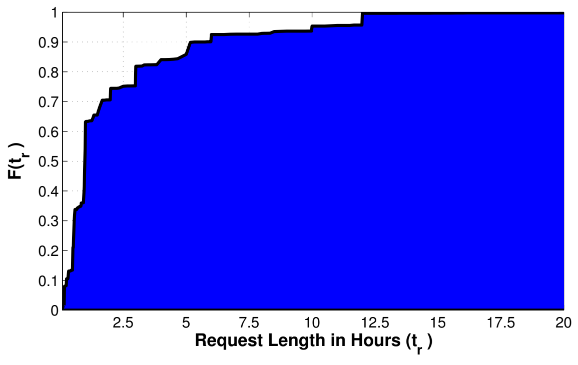
In our evaluation we assume demand is constant, regardless of the amount charged to the user. Preliminary results assuming linear demand curves are given in Appendix B.1.

4.5 Experimental Setup

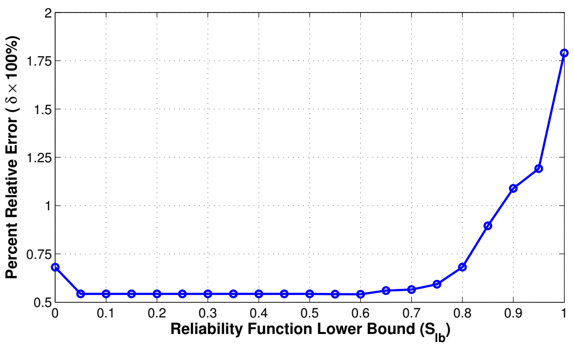
Access to the Spot market price history for all instance types and zones is available through Amazon's API for the past 90 days, and from websites such as (University of Western Sydney [2013]) for longer periods of time. Simulations were run using a set of 20,000 requests that are (1) distributed over the period between September 12 and November 12, 2012, (2) have arrival times and requested runtimes taken from traces from the ANL Intrepid supercomputer (Feitelson), and (3) have requested instance types sampled uniformly from those given in Table 2.1, all running a Linux operating system.

Figure 4.3a illustrates the CDF of the request lengths for those requests with $t_r < 20$ hours. Traces from ANL Intrepid were chosen for the following reasons:

- The dispersal of requests over several months facilitates an observation of Spot price characteristics over an extended period of time. Such a broad window helps



(a)



(b)

Figure 4.3: (a) CDF of the request lengths, t_r, for all r such that $t_r \leq 20$ hours. (b) Percent relative error of the cost using Market Price cost estimation.

to eliminate any period-specific patterns that may occur during shorter windows of Spot price traces.

- The proximity of execution time estimates to the true values. Having close estimates allows for a more informative analysis of the strategies presented in this paper, rather than the accuracy of the runtime estimations. These traces provide realistic user estimates and also allow us to estimate real-world arrival times.

- The range of request lengths (from several minutes to several days) allows for a comprehensive evaluation of RAMP's methods to estimate the empirical probability of success, as well as the expected profit. Moreover, a large range of t_r provides an informative window into how RAMP handles both long and short requests. Figure 4.3a illustrates the CDF of the request lengths for those requests with $t_r < 20$ hours.

.

Spot prices were taken from the period from July 15, 2012 to November 15, 2012, in the region *us-east-1*, for all instance types listed in Table 2.1 and their corresponding availability zones. We assume that each request initially requires access to data residing in the zone *us-east-1a*, of size equal to 1GB, and which must be transferred, in negligible time, to the chosen zone at Amazon's rate of \$0.01/GB before the instance is available. Thus, instances not started within *us-east-1a* incur an extra \$0.01. When calculating the *reliability function* for a Spot instance, the set of uniformly sampled dates has size $|X| = 10,000$ and spans a window of the past 60 days of Spot price history. Starting in June, 2012, Spot market prices have fallen to as low as 1/10 of the equivalent On-Demand price, from their historical levels of between 1/3 and 1/2.

Figure 4.3b illustrates the percent relative error when using the market price to estimate costs, further showing that such an estimation can achieve very accurate results. Indeed, the percent relative error from the true cost attains a maximum of only 1.8% when $S_{lb} = 1$, and is around 0.5-0.75% for all $S_{lb} \leq 0.8$.

When comparing and evaluating profits with different values of S_{lb}, e, and p, the metric that will be used is:

$$\eta(S_{lb}, e, p) = \frac{\Pi(S_{lb}, e, p)}{-\Pi_0}, \tag{4.19}$$

where $\Pi(S_{lb}, e, p)$ is the *total profit* accrued by RAMP for the given values of S_{lb}, e, and p (i.e., $\Pi = \sum_r \pi_r$), and $\Pi_0 = \Pi(0,0,0)$ is the *baseline total profit*: the total profit when all three parameters are zero. Here, Π_0 is always negative, and $-\Pi_0$ therefore represents the *baseline total cost* (excluding penalties) of RAMP with $S_{lb} = 0$, an approach which simply finds the instance with the highest expected profit generated by simply bidding the market price for that instance. In our experiments, $\Pi_0 = -\$6,003$.

4.6 Results

Experimental results will be evaluated via the resulting changes in early-termination rates, τ, and the measure of total profit, η, when altering the fraction of the On-Demand price the user is charged, p, the reliability level lower bound required for instances, S_{lb}, and the fraction of the original payment awarded to the user in the event of early-termination, e.

4.6.1 Early-termination rates

The instance reliability evaluation and bidding strategy, discussed in Section 4.2, can keep successful completion rates above S_{lb} (i.e., $1 - \tau \geq S_{lb}$) over the entire simulation period, provided $S_{lb} \leq 0.971$. Figure 4.4 depicts the actual early-termination rate for various values of the reliability function lower bound, when charging the user 25% of the equivalent per-second On-Demand price, and refunding the user an additional 25%

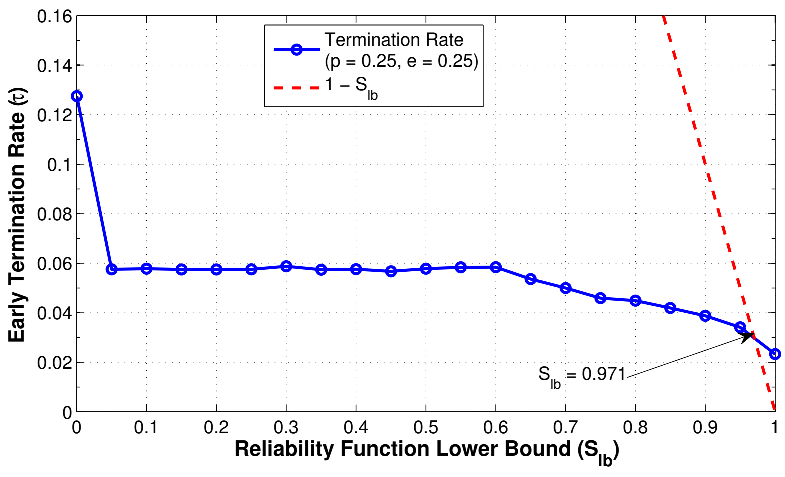


Figure 4.4: Early Termination Rates (τ) vs. the *reliability function* lower bound (S_{lb}), when $e = p = 0.25$.

of their payment in the event of early-termination. Early-termination rates range from a maximum of approximately 13%, when $S_{lb} = 0$, to 2.2%, when $S_{lb} = 1$. Due to the ability to exploit different instance types and availability zones to locate more reliable instances, this early termination rate is often far lower than $1 - S_{lb}$. However, once S_{lb} is increased above 0.971, early-termination rates are higher than the desired level. This leveling-off, coupled with the non-zero minimum, of early-termination rates is due to the frequent occurrence of market price spikes, as discussed earlier, which either may not be observed during the random sampling, or are higher than those previously encountered.

4.6.2 Varying the amount charged

Regardless of the value of S_{lb} and e, positive profits can be achieved while charging the user only a small fraction of the On-Demand price for the equivalent instance. Figure 4.5 illustrates RAMP's total profit for different values of p, given different values of the reliability function lower bound, and with $e = 0.25$. Total profit in the figure attains a maximum when $S_{lb} = 1$ and a minimum for $S_{lb} = 0.75$. Furthermore, letting $S_{lb} = 1.0$ attains the maximum level of profit for all $p > 0$, with total profit ranging from $\eta = -1.0$, a net loss equal to the total baseline cost, to $\eta = 1.8$. For most values of S_{lb}, RAMP generates positive profits while the user pays greater than or equal to approximately 17.5-20% of the equivalent On-Demand price of the requested instance.

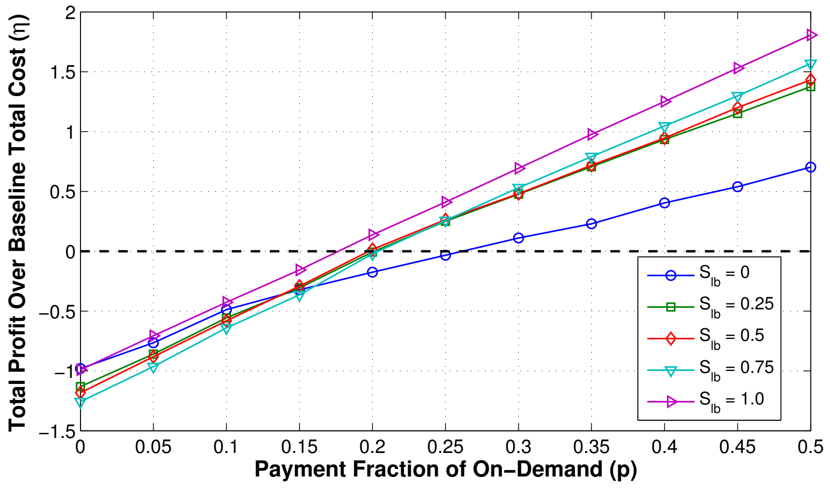


Figure 4.5: Measure of total profit for different values of p given specific values of S_{lb} and with $e = 0.25$

Although the actual value of p needed to break even depends on the values of e and S_{lb}, to completely offset the total cost, RAMP requires p to fall between 0.15 and 0.25. In addition, RAMP can achieve 100% of the total baseline cost as profit when charging the users just 35% of the corresponding On-Demand price, and can achieve 200% of the total baseline cost as profit when charging the user approximately 53% of the On-Demand price. For the equivalent strategy using market-price bids, the same level of profit requires RAMP to charge the user as much as 60% and 80% of the On-Demand price, respectively, depending on the value of the penalty fraction.

4.6.3 Varying instance reliability

Increasing S_{lb} can either increase or decrease the profit, depending on the values of p and e. Figure 4.6 shows the total profit for different values of the lower bound on instance reliability, given various values of the penalty payment fraction and while letting $p = 0.25$. As seen in this figure, when charging the user 1/4 of the equivalent On-Demand price, profits generally increase as S_{lb} increases, with the chosen value of e heavily influencing the total profit for lower values of S_{lb}. For example, when $S_{lb} = 0$ (a market-price bidding strategy), letting $e = 0$ results in a total profit over total baseline cost of $\eta = 0.1$, whereas when $e = 1.0$, $\eta = -0.52$ (a net loss for RAMP equal to approximately half of the total baseline cost). When $S_{lb} = 1$, however, profit is maximized independent of e. For this value of S_{lb}, the error plays a much less important

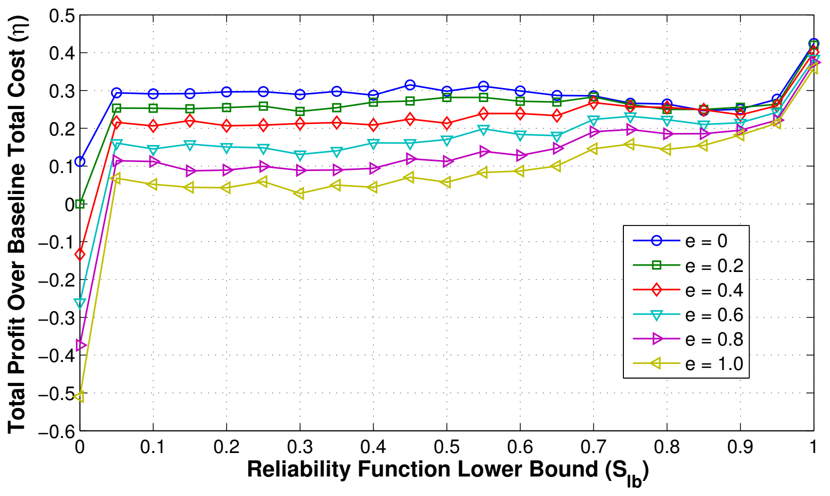


Figure 4.6: Measure of total profit for different values of S_{lb}, given for various values of e and with $p = 0.25$

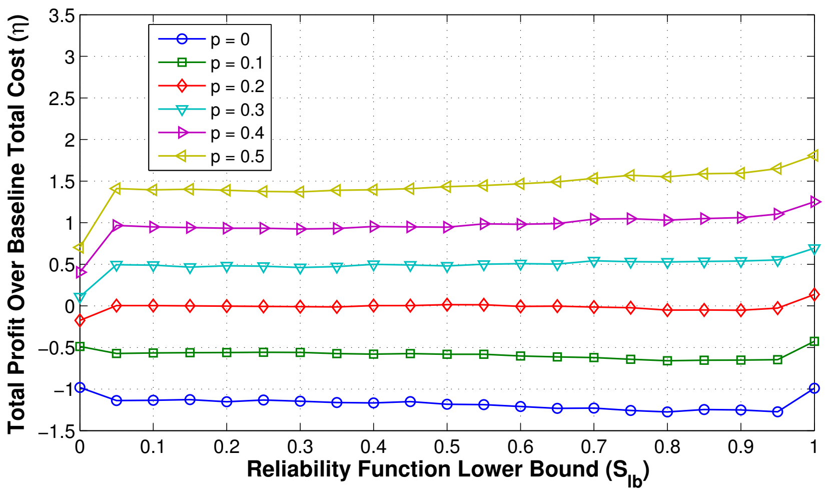


Figure 4.7: Measure of total profit for different values S_{lb} given specific values of p and with $e = 0.25$

role due to the decrease in early-terminations accompanying higher reliability values; letting $e = 0$ produces $\eta = 0.43$, while letting $e = 1.0$ produces $\eta = 0.36$.For $S_{lb} < 1.0$, profit is maximized at $\eta = 0.33$ when $S_{lb} = 0.45$.

For lower values of p and $S_{lb} < 1$, letting $S_{lb} = 0$ can yield lower losses and higher profit. As demonstrated in Figure 4.5, for $p > 0.125$ and $S_{lb} < 1$, profit is generally highest when $S_{lb} = 0$, with total profit surpassing that of $S_{lb} = 1.0$ when $p = 0$. For $p < 0.125$, however, higher values of S_{lb} maintain the highest profits, with $S_{lb} = 1$ surpassing all others. Figure 4.8 also illustrates the fact that $S_{lb} = 1$ achieves the highest

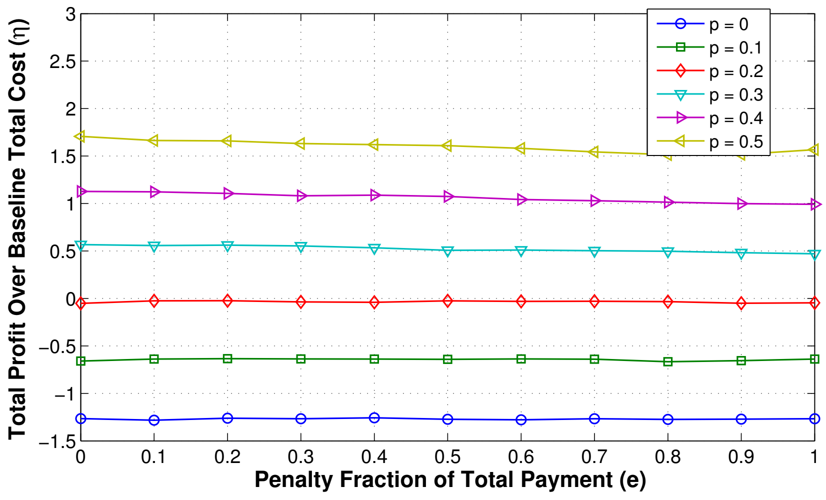


Figure 4.8: Measure of total profit for different values e given specific values of p, and with $S_{lb} = 0.95$

profit, as well as the fact that decreasing S_{lb} from 0.05 to 0 sees a sharp drop in profit, regardless of the value of e, with a change in η ($\Delta\eta$) of up to 0.6.

As demonstrated in Figure 4.7, total profit increases at an approximately constant rate as p increases, and these profits are relatively stable for each value of p. Total profits for each given value of p increase or decrease depending on whether p is, respectively, greater than, or less than, approximately 0.2, the point at which total profits are relatively stable and equal to 0. This opposite effect is due to the fact that higher values of S_{lb} incur larger costs due to the higher bids on Spot instances and the more frequent use of On-Demand instances. Thus, if p cannot sufficiently cover these costs, increasing S_{lb} will serve only to decrease the profit accrued. As discussed above, when $S_{lb} = 1$, total profit always increases.

4.6.4 Varying the penalty

Figure 4.8 illustrates the total profit for different values of the penalty payment fraction, e, given various values of p, and with $S_{lb} = 0.95$. For higher values of p, the penalty payment fraction significantly influences the profit, with an increase in e from 0 to 1 resulting in a change of $\Delta\eta \approx 0.25$ when $p = 0.5$. For lower values of p this drop becomes far less pronounced, and is almost nonexistent for $p \leq 0$. Varying the penalty may also have a pronounced effect depending on the value of S_{lb}. Figure 4.6 demonstrates that, for moderate-to-low values of S_{lb}, due to the larger number of early terminations, increasing e can significantly decrease the total profit. For example, when $S_{lb} = 0$, increasing e from 0 to 1 will drop η from a gain of approximately 11% of the baseline total cost to a loss of approximately 52% of the baseline total cost. Furthermore, Figure 4.6 also demonstrates that, as e increases, local maxima for the profit are shifted further to the right, and merge with the global maximum at $S_{lb} = 1$. Moreover, for higher values of e, profits rise more rapidly when S_{lb} is increased.

4.7 Discussion

The evaluation performed in the preceding section yields four main results regarding the profitability and volatility of RAMP, when using various combinations of S_{lb}, p, and e, as discussed below.

The instance-reliability evaluation and bidding strategy discussed in Section 4.2 can keep successful completion rates above S_{lb} over the entire simulation period, provided $S_{lb} \leq 0.971$. The approach presented in this paper can significantly eliminate much of the volatility inherent in Spot instances . Using the reliability evaluation and bidding strategy presented in this paper, RAMP is able to lower the early-termination rate from an unsustainable level of 13% when using a market-price bidding strategy (i.e., $S_{lb} = 0$), to a much more reasonable and manageable level. In addition, RAMP can keep the early termination rate below that requested (i.e., $\tau \leq 1 - S_{lb}$) for all but very high values of S_{lb}. However, regardless of the value of S_{lb}, there will still be a minimum early-termination rate and this minimum may need to be considered when offering SLA guarantees of the termination rate. Although it is possible to make the early-termination rate approach 0 through the use of very high bids, as discussed in Sections 2.1.4 and 4.2.2, such an attempt will necessarily be coupled with far higher costs.

To help avoid such blind-bidding strategies, and to prevent overbidding, other methods to lower these termination rates can include a higher number of samples and/or a larger window of the price history in which to sample. Additionally, a relaxation of the constraints necessary for an instance to be greater than or equal to that requested will help lower early-termination rates, and possibly further decrease costs, by increasing the size of the search space during instance acquisition and allocation.

Regardless of the values of S_{lb} and e, positive profits can be achieved while charging the user only a small fraction of the On-Demand price for the equivalent requested *instance.* For all reliability levels and penalty amounts, requiring that the user pays around 1/5 of the price of the equivalent On-Demand instance they have requested results in positive profit, with many values of S_{lb} and e generating positive profit for values of p as low as 0.15. In the context of a private system operator, these results imply that total costs can be completely offset when paying just 20% of the equivalent per-second On-Demand cost per instance, for each request. Moreover, when $S_{lb} = 1$, this is also coupled with an early-termination rate of only 2.2% and, when $e > 0.15$, letting $S_{lb} = 1$ yields a total profit of 100% of the baseline total cost when charging users just 35% of the equivalent On-Demand cost, assuming constant demand. In addition, the total profit generated by RAMP will rise linearly as p increases, and 200% of the baseline total cost can be generated as profit when charging the user just 53% of the equivalent On-Demand price.

Increasing the penalty tends to increase the required reliability level at which total profit is maximized. Although moderate and high values of the penalty fraction generally obtain near-maximum profit when $S_{lb} \approx 1$, decreasing the penalty fraction to values less than 0.3 tends to create local maxima closer to $S_{lb} = 0$. In fact, for values of e close to 0, it may actually be, in some cases, more profitable for RAMP to allow some instances to be terminated, and pay the penalty, rather than incur the high market prices that may result otherwise. This trade-off between paying penalties and incurring high market prices is more pronounced as e, and thus the amount paid in penalty in the event of early-termination, decreases. RAMP's response to this trade-off is reflected in a requirement of lower values of S_{lb} in order for RAMP to achieve the reasonably high levels of profit. Such a result implies that a per-request reliability level may be a better choice than a global level.

There is, respectively, a significant decrease in failure rates and a significant increase in profit when S_{lb} is increased to any amount above 0. Such significant changes occur because, when $S_{lb} = 0$, RAMP simply finds the instance type and availability zone which will generate the maximum profit when bidding the market price. Although the ability to search among availability zones and instances can still provide an instance with high reliability, any increase in the market price will result in the termination of the instance and the subsequent refund of the user's money plus a penalty. However, when $S_{lb} = \varepsilon$, for small $\varepsilon > 0$, the bidding strategy finds the lowest bid with positive probability of completion. In many cases, this lowest bid will still correspond to a reasonable high reliability value due to the ability to search among different instance types and availability zones. Thus, in all cases, bidding the market price for a Spot instance is *always* inferior to placing a bid with the maximum of either the market price or any greater bid with nonzero probability of completion. In fact, for higher penalties, finding the instance with the highest expected profit from bids that resulted in 100% reliability in the past will yield the highest profits.

Profit is generally maximized when $S_{lb} = 1$. Choosing the maximum reliability for instances will almost always generate the highest level of profit for RAMP, independent of the payment or error. The stability of Spot market prices helps to reduce the risk associated with choosing *reasonably* high bids, which in the past may have made the user susceptible to very high market prices (see Section 4.2.2).

Chapter 5

RAMC-DC

This chapter presents an overview of RAMC-DC, a Reliability and Availability Aware Cost-Minimizing Resource Provisioner for deadline constrained jobs. In this chapter, RAMC-DC's approach to modeling job execution on Spot and On-Demand instances will be discussed. Following this, a two-tier instance evaluation strategy will be presented, followed by a bidding strategy for leasing new Spot or On-Demand instances, as well as an outline of the algorithm that searches for the optimal instance on which to run a job. Afterwards, the approach to job-scheduling on instances will be outlined. Results from simulations used to evaluate RAMC-DC will then be outlined and discussed.

5.1 Overview

In this section, the model used for user-submitted jobs is presented, and the problem faced by RAMC-DC is formalized.

5.1.1 Job Models

Users submit a job, j, which is placed in an FIFO queue, J. Each job is independent and includes a desired instance type, i_j, an estimated execution time, $\widehat{t}$ (given in full or partial hours), on instance type i_j, and a deadline, D. In the event of termination, j also contains a reference to the last zone j was executed in, z_j, and i_j is updated with j's last instance type. The zone z_j is initially equal to $\emptyset$, and is updated upon j's execution on some instance.

5.1.2 Problem Formulation

RAMC-DC searches among spot and on-demand purchasing options for the most cost efficient instance on which to run a deadline-constrained job. Specifically, the framework searches across different instance types and geographical zones and uses job execution time, estimated cost, and instance reliability as selection criteria when allocating instances to jobs. In addition, RAMC-DC aims to address issues associated with spot instances including: determining when to use spot instances; which spot instances to use; when to use other, less volatile purchasing options; and finding a bid that represents a desired trade-off between the probability of early termination and the cost of running the job, while still being able to reuse the instance for other jobs.

Users submit jobs, as described in the preceding subsection, which are placed in a queue, J. The algorithm incorporates tunable parameters $S_{lb} \in [0,1]$ and $t_{split} \in \mathbb{R}_{+}$, with the first parameter specifying a minimum **confidence level**, similar to that used in RAMP, for instance reliability and availability evaluation, and the second representing a **job execution time splitting parameter**, in full or partial hours, used to separate long jobs from short jobs and thereby determining the bidding strategy used for that job. S_{lb} is generally used to specify how much resistance to early termination is required for spot instances; higher values of S_{lb} generally increase the bid and thus incur higher costs but lower early termination rates.

To evaluate the effects of moldability (the speedup or slowdown encountered on different instance types) in our search, we use Downey's speedup model as an exemplar for estimating job execution time on different instances. Downey's speedup model requires only the knowledge of the processing power of different instance types to calculate the speedup, and thus we will evaluate speedup using the number of EC2 Compute Units in an instance, n_i. When leasing and evaluating Spot instances, RAMC-DC employs a two-tier instance evaluation and bidding strategy. This strategy uses the given execution time, $\widehat{t}$, to classify the job as *long* or *short*. If the job is *short*, to allow for reusability of the instance after job completion, RAMC-DC finds a set of possible instances, and their corresponding bids if Spot instances, that have observed a minimum level of availability of that instance in the past. On the other hand, for *long* jobs, RAMC-DC locates the set of instances and corresponding bids that guarantee a minimum probability of successful completion of the job on that instance. The choice of which particular instance within each set is used to execute the job is made using by a calculation of whether the job can be executed before the deadline, as well as the estimated cost of the job on that instance. As with RAMP, this cost will be approximated at the time of evaluation. In addition, jobs run on spot instances will incorporate one of three basic checkpointing strategies: none, hourly, and rising-market-price.

The problem: For each job in the queue, RAMC-DC wishes to locate an instance-job assignment that minimizes the cost, $C(j,v)$, of running the job on that instance while meeting either reliability or availability constraints, depending on whether a job is classified as *short* or *long* (i.e., if $\widehat{t} \leq t_{split}$). Thus, given a job j, RAMC-DC seeks to find the instance v^* such that:

$$v^* = \arg\min_{v \in V} C(j,v), \tag{5.1}$$

where the domain V is a search space comprised of a set of leased and unleased instance representations such that (if possible):

$$\forall v \in V, j \text{ can be executed on } v \text{ before } D \text{ and} \begin{cases} Availability(v) \geq S_{lb} & \text{if } \widehat{t} \leq t_{split} \\ Reliability(j,v) \geq S_{lb} & \text{if } \widehat{t} > t_{split}. \end{cases}$$

Here, *success* requires that the job successfully execute on the allocated instance.

In the event of early-termination, RAMC-DC will insert the job at the front of the queue and try again. If the job was checkpointed before termination, RAMC-DC will readjust the estimated execution time to incorporate the remaining time and the time to migrate the saved instance state into the search for a new instance. The lifecycle of a job is presented in Figure 5.1.

5.2 Modeling Job Execution

In order to efficiently locate instances on which to run the job, we first need to effectively model the execution of a job on some instance. This includes estimating the job's execution time and total cost given the instance type, availability zone, and whether the instance is a Spot instance or an on-demand instance.

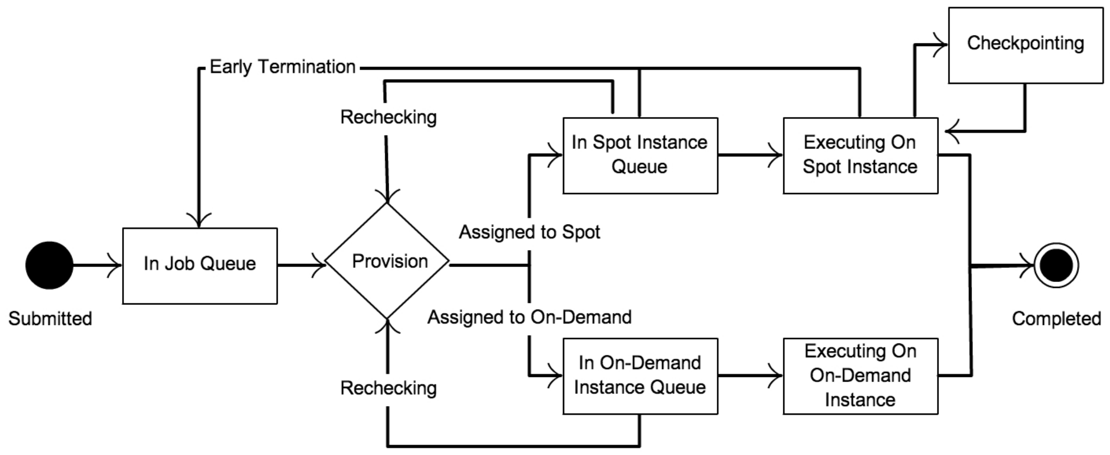


Figure 5.1: Lifecycle of a Job.

5.2.1 Estimating Job Execution Time

To compare cost and performance dynamics across the instance offerings in Amazon EC2, we will utilize a means to model job execution time that takes into account factors such as the number of EC2 compute units provided by an instance, and the extent to which the job can make use of these compute units. Using such a means allows us to compare price and performance characteristics among different instances. Thus, to determine the execution time of a job on some instance, we assume each job is independent and known to be one of the following.

- *Moldable*: For a job to be moldable implies that some speedup or slowdown is observed when running the job on larger or smaller instances (with respect to computational power), respectively. Speedup will be determined similarly to the approach presented in Voorsluys and Buyya [2012], using Downey's speedup model (Downey [1997]).[1] Downey's model requires two additional parameters, A, and σ, which measure the *average parallelism* and the *coefficient of variance in parallelism*, respectively, and measures the increase in execution time for a job running on n processors compared to a job running on 1 processor. Given A and

[1]Additional results using Amdahl's law are presented in Appendix C.1.

σ, the speedup of a job using n processors is:

$$SU(n) = \begin{cases} \frac{An}{A+\sigma(n-1)/2} & (\sigma \le 1) \wedge (1 \le n \le A) \\ \frac{An}{\sigma(A-1/2)+n(1-\sigma/2)} & (\sigma \le 1) \wedge (A \le n \le 2A-1) \\ A & (\sigma \le 1) \wedge (n \ge 2A-1) \\ \frac{nA(\sigma+1)}{\sigma(n+A-1)+A} & (\sigma \ge 1) \wedge (1 \le n \le A+A\sigma-\sigma) \\ A & n \ge A+A\sigma-\sigma \end{cases} \tag{5.2}$$

Therefore, we calculate the estimated execution time on instance i as:

$$\widehat{t_i} = \widehat{t} \cdot SU(n_i)/SU(n), \tag{5.3}$$

where n_i is the number of EC2 compute units in i. Values of A and σ are calculated using the model of Cirne and Berman (Cirne and Berman [2001]), and are assumed to be known at submission.[2] Rudimentary evaluations using Amdahl's law were also performed, and may be seen in the Appendix.

- *Rigid*: For a job to be rigid implies that no speedup is encountered on larger instances, and that the job will not execute on smaller instances. Thus, for an instance with n_i EC2 compute units we have:

$$\widehat{t_i} = \begin{cases} \infty & n_i < n \\ \widehat{t} & \text{otherwise} \end{cases} \tag{5.4}$$

Thus, rigidity requires that only instance types with $n_i \ge n$ be used to execute the job, similar to the approach presented for locating the set of *greater* instance types used by RAMP.

5.2.2 Incorporating Resource Volatility

As resource reliability/volatility particularly with Spot instances needs to be explicitly dealt with we extend the estimation model above incorporating checkpointing times. For some availability zone, z, the execution time in z is modified to include the estimated

[2]The model of Cirne and Berman was constructed using questionnaires distributed to supercomputer users. A is modeled using a joint uniform-log distribution, and σ is modeled using a normal distribution.

Table 5.1: Frequently Used Variables (RAMC-DC)

Variable	Description
t_{split}	The parameter dividing long jobs into short jobs.
S_{lb}	The parameter specifying the confidence level.
j	A job submitted by a user.
n	The number of requested EC2 Compute Units.
$\widehat{t}$	The estimated execution time of a job.
D	The job's deadline.
i	The instance type.
z	The availability zone for an instance.
b	The bid for an instance.
t_i^{susp}	The suspend time of an instance.
$t_{z_j \to z}^{res}$	The resume time of an instance to zone z.
$\widehat{t_i}$	The speedup adjusted execution time.
$\widehat{t_{i,z}}$	Execution time adjusted for checkpointing and resume time.

checkpointing times of the job, if run on a Spot instance, and the time to resume the job's previous instance from a suspended state if the job's previous instance was checkpointed during execution:

$$\widehat{t_{i,z}} = \widehat{t_i} + \widehat{t_i^{chkpt}} + t_{z_j \to z}^{res}. \tag{5.5}$$

Here, the total checkpointing time, $\widehat{t_i^{chkpt}}$, as well as the resume time, $t_{z_j \to z}^{res}$, of an instance in zone z from some checkpoint in j's last zone z_j, are determined as in Voorsluys and Buyya [2012] using research by Sotomayor et al. [2008], where the suspend and resume rates of a virtual machine state in the same availability zone are $s = 63.67 MB/s$ and $r = 81.27 MB/s$, and the resume rate from a different availability zone is set to $r/2$. Thus, the time per checkpoint is determined as the time to save the instance's memory to a global file system (e.g., Amazon S3), and is given as $t_i^{susp} = m_i/s$ where m_i is the memory size of instance type i. Similarly, the time to resume a checkpointed instance state is calculated as:

$$t_{z_j \to z}^{res} = \begin{cases} m_i/r & \text{if } z == z_j \\ m_i/(r/2) & \text{otherwise} \end{cases} \tag{5.6}$$

where m is the memory size of the job's last instance. When resuming instance states on On-Demand instances, we let $t_{OD}^{res} = t_{z_j \to z_j}^{res} = m_i/r$, since we assume z will always be the same availability zone in which the job was last executed (i.e., z_j) when a job is re-executed as an On-Demand instance as On-Demand instances are available in every zone Spot instances are, and pricing characteristics for On-Demand instances are static across zones.

The following basic checkpointing methods, and their associated total checkpointing time, are compared when running jobs on a Spot instance:

1. *None*: No checkpoints are taken. Therefore, the estimated checkpointing time is $\widehat{t_i^{chkpt}} = 0$ and, upon forced termination, all completed computation is lost, forcing the job to be restarted from scratch.

2. *Hourly*: A checkpoint is taken at the end of each hour block. Estimated checkpointing time is therefore calculated as $\widehat{t_i^{chkpt}} = \lfloor \widehat{t_i} \rfloor \cdot t_i^{susp}$ and, upon forced termination, execution resumes from the end of the last hour before termination. As the execution time is given in hours, $\lfloor \widehat{t_i} \rfloor$ specifies the number of full hours the job is estimated to require on instance type i.

3. *Rising Market Price*: A checkpoint is taken each time the market price for that instance rises. Thus, the estimated number of checkpoints is taken as the average number of price increases for a $\widehat{t_i}$ period over the past 60 days and the estimated checkpointing time is calculated as $\widehat{t_i^{chkpt}} = avg\_incr \cdot t_i^{susp}$.

When leasing an instance, there generally is an acquisition time of order several minutes before the instance is available. Recent research by Mao and Humphrey [2012] suggests that acquisition times for EC2 instances can vary greatly by provider, purchasing option, region, availability zone, instance type, and operating system. As instances generally will be leased infrequently, jobs lengths are usually many times the acquisition time, and to focus on the price, reliability, and performance dynamics of Spot and on-demand instances, acquisition times will not be examined in this paper.

5.2.3 Estimating the Cost of Job Execution

Given the fluctuating prices of Spot instances, an accurate method to estimate the total cost of running a job on each such instance must be found. For each instance type and availability zone combination, (i,z), such that $i \in i$ and $z \in Z_i$, the algorithm has access to Amazon's Spot price history for some past span of time: $H_{i,z} = \{(p_1,d_1),\ldots,(p_k = p_{mkt},d_k)\}$, where p_i is the price at time d_i, and p_k is the current market price. To determine the best way to estimate this total cost, we may potentially used any of the cost-estimation methods presented in Chapter 3, but with the calculation of the cost of the speedup adjusted runtime plus the checkpointing and VM-state resume overheads (if applicable). For simplicity, we will assume $\widehat{C}(j,v)$, the estimated cost of execution j on instance v is calculated as $\widehat{C}(\widehat{t_{i,z}}, \langle i,z,b\rangle)$; e.g.,

$$\widehat{C}_{mkt}(j,v) = \widehat{C}_{mkt}(\widehat{t_{i,z}}, \langle i,z,b\rangle) = p_{mkt} \cdot \lceil \widehat{t_{i,z}} \rceil, \tag{5.7}$$

and

$$\widehat{C}_{ma\_\alpha}(j,v) = \widehat{C}_{ma\_\alpha}(\widehat{t_{i,z}}, \langle i,z,b\rangle) = \widehat{C}_{mkt}(\lceil\alpha\rceil, \langle i,z,b\rangle) + \widehat{C}_{avg}(\widehat{t_{i,z}} - \lceil\alpha\rceil, \langle i,z,b\rangle). \tag{5.8}$$

Similarly, estimated costs for On-Demand instances will be calculated as:

$$\widehat{C}_{OD}(j,v) = \widehat{C}_{OD}(\widehat{t_{i,z}}, i) = \lceil \widehat{t_{i,z}} \rceil \cdot ODPrice_i, \tag{5.9}$$

where $ODPrice_i$ is the On-Demand price for instance i.

In addition, as discussed in Chapter 3, If v has been leased, $\widehat{C}(j,v)$ is calculated as the total cost accounting for the fact that the remaining hour block has already been paid for. Thus, $\widehat{C}(j,v)$ approximates the cost using the estimated execution time $\widehat{t_{i,z}} - RemHour(v)$, where $RemHour(v)$ is the predicted time that will remain in v's hour block when j is expected to start.

5.3 Dynamic Resource Provisioning

The overall resource provisioning process RAMC-DC employs is performed by (1) evaluating instance suitability based on j's execution time and t_{split}, (2) finding the most cost-effective instance among already leased and unleased instances that satisfy the evaluation lower bound, S_{lb}, as well as the deadline, D, (3) leasing a new Spot or On-Demand instance if required (i.e., the optimal instance is unleased), and (4) assigning j to the resulting instance.

5.3.1 Two-Tier Instance Evaluation

In addition to estimating the execution time and cost of running a job on an instance, when leasing an instance we must also determine the specific instance type to acquire, the availability zone in which to run it, and, if the instance is a Spot instance, the bid at which the instance will be requested. The determination of these values is facilitated via a two-tier instance evaluation strategy that involves the calculation of the reliability or availability of an instance ν, depending on whether job is classified as *short* or *long*. Within this study, this classification is done through a comparison of the estimated execution time, $\widehat{t}$, and the splitting parameter, t_{split}. If $\widehat{t} \leq t_{split}$, the job is classified as *short*. Otherwise, the job is classified as *long*. Therefore, the instance evaluation function is given as:

$$S(j,\nu,t_{split}) = \begin{cases} Availability(\nu) & \text{if } \widehat{t} \leq t_{split} \\ Reliability(j,\nu) & \text{if } \widehat{t} > t_{split}. \end{cases} \tag{5.10}$$

The two functions, *Reliability* and *Availability*, represent the empirical job-specific *reliability* and overall *availability* of an instance, and are calculated for a job j and instance ν as follows.[3]

1) For $\widehat{t} \in j$, if $\widehat{t} > t_{split}$ then $S(j,\nu,t_{split})$ is calculated as the empirical probability of *success* if ν was used to execute j using a similar method to that proposed in Section 4.2.1. That is, $Reliability(j,\nu) = P(T_{i,z,b} \geq \widehat{t}_{i,z})$ where $T_{i,z,b}$ is a random variable representing the true length of time for which the Spot instance is available to the user when bidding b on instance type i in availability zone z, and $\widehat{t}_{i,z}$ is the estimated execution time on instance type i in zone z. As in Section 4.2.1, this probability can estimated using the nonparametric Kaplan-Meier Estimator:

$$Reliability(j,\nu) = \prod_{t_{i,z,b}(x) \leq \widehat{t}_{i,z}} \frac{n_{i,z,b}(x)-1}{n_{i,z,b}(x)}, \tag{5.11}$$

where X is a set of dates sampled uniformly over some window of the Spot price history, $t_{i,z,b}(x)$ is the true step length for an instance leased at time $x \in X$ with type i in zone z, and with bid b. Here, $n_{i,z,b}(x) = \left|\left\{y \in X \setminus \{x\} \mid t_{i,z,b}(y) \geq t_{i,z,b}(x)\right\}\right|$ is the number of dates $y \in X$ $(y \neq x)$ in which the instance with the previously-mentioned parameters and started at y was available for longer than $t_{i,z,b}(x)$, the true available time (time before termination) of an instance started at x. That is, the number of samples in which an instance $\langle i,z,b \rangle$ had its bid above the market price for longer

[3]An implementation of $Availability(\nu)$ for unleased ν is given in Listing C.2, and the two-tier strategy is given in Listing C.1. $Reliability(j,\nu)$ uses the same approach as Listing B.1.

than $t_{i,z,b}(x)$.

2) If $\widehat{t} \leq t_{split}$, $S(j, v, t_{split})$ is calculated as the portion of time within the Spot price window that an instance with v's specifications would have been above the market price. Hence, $S(j, v, t_{split})$ will be calculated as:

$$Availability(v) = \frac{\sum\limits_{p_m \leq b,\, m<k} (d_{m+1} - d_m)}{d_k - d_1}. \tag{5.12}$$

As described in Chapter 3, $(p_m, d_m) \in H_{i,z}$ for $m = 1, \ldots, k$, and $p_k = p_{mkt}$ is the current market price. $H_{i,z}$ is a window of the Spot price history over some period of time.

5.3.2 Bidding

The optimal bid for a Spot instance is calculated as the minimum bid that satisfies a lower bound on the instance evaluation function described above. Thus, if we wish to execute a job j on a Spot instance of type i in zone z, we may calculate this required bid in a similar fashion as that in the bidding strategy for RAMP as:

$$b(r, i, z) = \min_{b \geq p_{mkt}(i,z)} \{b \mid S(r, \langle i, z, b \rangle) \geq S_{lb}\}, \tag{5.13}$$

for p_{mkt} as defined above. Note that the only differences between the bidding strategy here and the one used in RAMP lies in the definition of $S(j, \langle i, z, b \rangle)$, and the inclusion of the t_{split} parameter.

If the job has estimated execution time greater than the splitting parameter t_{split}, the bidding strategy located the minimum bid such that the empirical probability of completion of j on an instance of type i in zone z is greater than or equal to the lower bound, S_{lb}. Such a strategy helps to provide job-specific bids that can limit the risk of early-termination for long jobs. On the other hand, if the execution time is less than t_{split}, the bidding strategy instead locates the minimum bid such that the instance has been available (i.e., the market price has been under the bid) for at least S_{lb} of the time over the Spot price window. This approach helps guarantee that instances are *interchangeable* among short jobs (no instance has a bid specifically tied to some job), thereby filling partially empty hour blocks.

Algorithm 3: Provision - Identifying the minimum cost job-instance assignment and provisioning resources.

```
Data: $J, S_{lb}, t_{split}$
1  begin
2      $SPOT \leftarrow \emptyset$, $OD \leftarrow \emptyset$
3      while $true$ do
4          $j \leftarrow$Pop($J$) // waits for J to be non-empty
5          $V \leftarrow \emptyset$, $v^* \leftarrow \emptyset$, $v_{new} \leftarrow \emptyset$, $breach \leftarrow false$
6          $I_j \leftarrow \{i \in I : \widehat{t_i} + t_{OD}^{res} \leq D\}$
7          if $I_j == \emptyset$ then
8              $I_j \leftarrow \{i \in I \mid n_i \geq n_j\}$, $breach \leftarrow true$ // deadline breach
9              $V \leftarrow \{v \in SPOT \cup OD \mid i \in I_j \wedge \texttt{ETUI}(v) == 0 \wedge S(j, v, t_{split}) \geq S_{lb}\}$
10         else
11             $V \leftarrow \{v \in SPOT \cup OD \mid i \in I_j \wedge \texttt{ETUI}(v) + \widehat{t_{i,z}} \leq D \wedge$
                   $S(j, v, t_{split}) \geq S_{lb}\}$
12         end
13         $v_{new} \leftarrow$MinNew($j, S_{lb}, t_{split}, I_j, breach$) // Algorithm 2
14         $v^* \leftarrow \arg\min_{v \in V \cup \{v_{new}\}} \widehat{C}(j, v)$
15         if $v^* == v_{new}$ then
16             Lease($v^*$) // lease $v^*$ from Amazon EC2
17             if $v^*$ is a Spot instance then
18                 Add($v^*, SPOT$)
19             else
20                 Add($v^*, OD$)
21             end
22         end
23         Assign($j, v^*$) // push $j$ to $v^*$'s FIFO queue
24     end
25 end
```

Due to the monotonically increasing nature of the relationship between the bid, b, and $S(j, v, t_{split})$, locating the minimum bid can be implemented as a hill-climbing search within the lowest and highest prices observed in the Spot price history.

5.3.3 Resource Provisioning

The process of resource provisioning and job assignment is described in Algorithm 3. Here, $ETUI(v)$ represents the estimated time until v is idle and equals the sum of the remaining estimated runtimes of each job assigned to v.

From the set of all instance types, I, described above, we determine the set of *feasible types*, $I_j \subset I$, that would satisfy the deadline with the corresponding On-Demand instances (Algorithm 3, line 6). If no feasible types exist, I_j is constructed as the set of instance types with n_i greater than or equal to that of the instance type the job was previously executed on. After I_j is constructed, the set of all *feasible instances*, V, is constructed as follows.

i) If there are no feasible types (as defined above), then for each $v \in V$, v is *idle* (i.e., has no jobs in its queue) and has greater than or equal to the number of EC2 compute units of the last instance j was executed on (lines 8-9). Narrowing the leased instance search space when jobs have surpassed their deadline to only those instances that are idle allows the framework to minimize the amount of time by which the deadline has been surpassed.

ii) Otherwise, for each $v \in V$, the sum of the estimated time until v is idle and the execution time of j on v must be less than or equal to D (line 11). Here, $EstTimeUntilIdle(v)$ equals the sum of the remaining estimated runtimes of each job assigned to v.

In both cases, each v must also satisfy the instance evaluation inequality, $S(j, v, t_{split}) \geq S_{lb}$.

The instance v_{new} in line 13, determined using Algorithm 4, represents the lowest cost instance that may *potentially* be leased if no lower cost already-leased instances are found.The optimal instance, v^*, is determined as the instance that minimizes the estimated cost of execution (line 14), subject to having either $reliability$ or $availability$ greater than S_{lb}. If v^* is not yet leased (i.e., $v^* == v_{new}$ as discussed above), an instance matching v^*'s description is leased as follows.

i) If $b \neq \emptyset$, a Spot instance of type i, in zone z, with bid b is leased (lines 17-18).

ii) Otherwise an On-Demand instance in zone z_j is leased, where z_j is either *us-east-1a* if j has not been previously attempted, or j's last availability zone otherwise.

After v^* is leased, it is added to the corresponding set of leased instances, *SPOT* or *OD*, and a pointer to this instance is added to v^*. The submitted job, j, is assigned to v^*'s FIFO queue to await execution. In this study, *SPOT* and *OD* are arrays of instance sets, with indices corresponding to an ordered enumeration of the

Note that instance v's queue need not be FIFO; instead, for example, v could keep a priority queue wherein jobs may be sorted by their relative proximity to their deadline (i.e., $\widehat{t_{i,z}}/D$), or their relative waiting time (i.e., the time since arrival). Such a queue will necessitate a change in the formulation of $EstTimeUntilIdle(v)$ to account for j's potential position in the priority queue, and all jobs after j's insertion position must be reevaluated for other, potentially better, instance alternatives due to the added delay from j's execution.

5.3.4 Identification of New Resources

The identification of v_{new} from line 13 of Algorithm 3 is outlined in Algorithm 4. The first **for** loop (line 3) iterates through the set of feasible instance types given by I_j, and the nested **for** loop (line 4) iterates through the corresponding availability zones in which a Spot instance of type i is available. For each (i,z) combination, if the estimated execution time on type i in zone z ($\widehat{t_{i,z}}$) satisfies the deadline, or if the job will surpass the deadline regardless, the estimated cost is compared to the current minimum. Although Amazon has since made such transfers free, if the job must be resumed from another availability zone, a data transfer cost is added at the rate of \$0.01/GB (Amazon [2013d]) and is calculated by $C_{resume}(j,z)$. Due to the static pricing characteristics of On-Demand instances among availability zones, potential On-Demand instances are evaluated for each instance type only (lines 14-18).

Due to the static pricing characteristics of On-Demand instances among availability zones, potential On-Demand instances are evaluated for each instance type only, with estimated costs for On-Demand instances calculated via $\widehat{C_{OD}}(j,v_{OD})$ as $\lceil\widehat{t_{i,z}}\rceil$ times the On-Demand price for type i (lines 16-22). As mentioned above, we assume $S(j,v_{OD},t_{split}) = 1$ for an On-Demand instance v_{OD}. Therefore, On-Demand instances will typically be used when other feasible Spot instance types are experiencing market price spikes, or bids guaranteeing high *availability* or *reliability* are estimated to have high associated costs.

Algorithm 4: MinNew - Identifying the minimum cost new potential instance satisfying $S(j,v,t_{split}) \geq S_{lb}$.

```
Data: $j, S_{lb}, t_{split}, I_j, breach$
Result: $v_{new}$ (an unleased instance)
begin
    $v_{new} \leftarrow \emptyset$, $c^* \leftarrow \infty$, $s^* \leftarrow 0$
    for $i \in I_j$ do
        for $z \in Z_i$ do
            if $\widehat{t_{i,z}} \leq D \vee breach$ then
                $v_{SPOT} \leftarrow \langle i,z,b(j,i,z,t_{split})\rangle$
                $s \leftarrow S(j, v_{SPOT}, t_{split})$
                $c \leftarrow \widehat{C}(j, v_{SPOT}) + C_{resume}(j,z)$
                if $(c < c^*) \vee (c == c^* \wedge s > s^*)$ then
                    $v_{new} \leftarrow v_{SPOT}$, $s^* \leftarrow s$, $c^* \leftarrow c$
                end
            end
        end
        $v_{OD} \leftarrow \langle i, z_j, \emptyset\rangle$ // potential new On-Demand
        $c \leftarrow \widehat{C}(j, v_{OD})$ // see Section 5.3.3
        if $c \leq c^*$ then
            $v_{new} \leftarrow v_{OD}$, $s^* \leftarrow 1$, $c^* \leftarrow c$
        end
    end
end
```

5.3.5 Job Scheduling and Resource Deprovisioning

If j has been assigned to an instance but has not been started before $D - \widehat{t_{i,z}}$ and the assigned instance is not idle, j is pushed to the front of J. Otherwise, prior to execution, the algorithm again searches for any lower cost instances on which to run the job and reassigns the job if a cheaper alternative is found. If no cheaper alternatives are found, i_j and z_j are updated and j is executed. In addition to the loss of Spot instances from early-termination, On-Demand and Spot instances are automatically released at the end of the current hour block if their assignment queues are empty.

In the event of forced termination of a Spot instance by Amazon, all jobs assigned to the instance are placed back into the job queue, J, in order of their initial submission time, with recalculated estimated runtime for the currently executing job, depending on when the job was last checkpointed, and with recalculated deadlines for all jobs on that instance. Although the queues evaluated in RAMC-DC are FIFO, it is possible to use priority queues based on the length of a job, or on the portion of time remaining before the deadline is breached (i.e., jobs may be ordered by the value of $D/\widehat{t_{i,z}}$ on an instance or $D/\widehat{t}$ in J). Such ordered queues may help to place an emphasis on assigning jobs that are close to their deadline over those that are not, and may consequently decrease the number of deadline breaches.

5.4 Evaluation Overview

In this section we first describe the experimental setup and evaluate the efficacy of RAMC-DC as follows.

- We compare and analyze the total costs (in comparison to the cost using only On-Demand instances) and deadline breach rates using different combinations of t_{split} and S_{lb}, and with different checkpointing strategies and job types (moldable and rigid).
- We measure the achieved early-terminations for different combinations of t_{split} and S_{lb} with no checkpointing strategy.
- We evaluate the average fraction of the deadline exceeded in each case.
- We evaluate the differences in our approach against a version of RAMC-DC that simply bids the market price for each instance, and determine that the any variation from such a strategy can significantly decrease total cost, early-termination rates, and deadline breaches.

5.5 Experimental Setup

Experiments to determine the effectiveness of our approach are divided into those relating to cost estimation, resource allocation and scheduling of moldable jobs, and resource allocation and scheduling of rigid jobs. We implement and evaluate our approach using Amazon's publicly available price history from the region *us-east-1* and search among all instance types, I, given in Table 2.1, with each running a Linux operating system. Each instance type $i \in I$ has an associated set of availability zones, $Z_i \subset \{us\text{-}east\text{-}1a, ..., us\text{-}east\text{-}1e\}$, in which a spot instance of that type is offered. Two sets of 20,000 jobs are constructed using arrival times, estimated execution times, and true execution times, taken from traces from the ANL Intrepid supercomputer, and with initial instances sampled uniformly from those in Table 2.1, as in the evaluation of RAMP (Chapter 5.4). Each set differs only in the assumption of rigidity or moldability. The values of A and σ used in our exemplar moldability analysis are taken from the model of Cirne and Berman, as discussed in Chapter 5.2. To demonstrate the efficacy of our approach over different pricing epochs, preliminary results from an earlier period of Spot prices are also given in Appendix C.2.

Experiments were run using both sets of jobs with various combinations of t_{split} and S_{lb}, and with each of the three checkpointing strategies. The total costs, deadline breach rates, and early termination rates were analyzed for each input parameter combination.

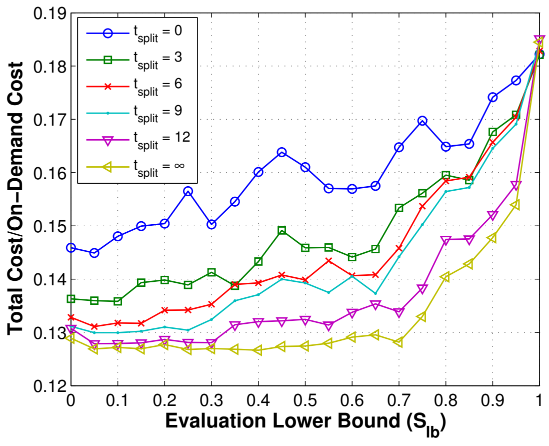
5.6 Results

The results from the experiments described above will be presented in the context of (1) Total Costs, (2) Deadline Breach Rates, (3) Early-Termination Rates, and (4) Average Deadline Exceeded.

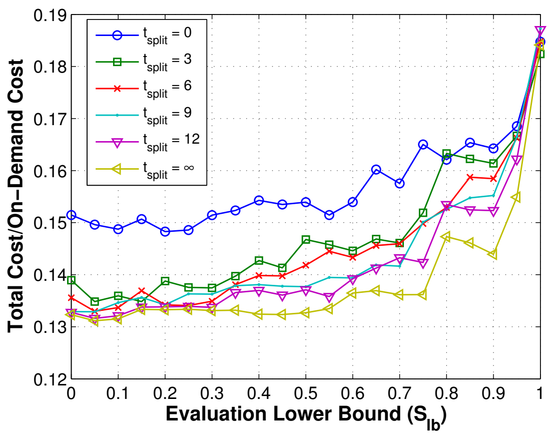
5.6.1 Total Costs

Total costs comparing all checkpointing strategies, with $t_{split} \in \{0, \infty\}$ for both sets of jobs are shown in Figure 5.4a and 5.4b. Total costs for various checkpointing strategies with moldable jobs are shown in Figure 5.2 and for rigid jobs in Figure 5.3. The total costs from using only On-Demand instances in our approach are equal to \$15,305 when using moldable jobs, and \$24,433 when using rigid jobs. The decision to include only the two values of t_{split} used in Figures 5.4a and 5.4b was made due to the fact that these values determine upper and lower bounds on the total cost of each checkpointing strategy for any value of t_{split}, regardless of the value of S_{lb}. Thus, these two figures specify the range of observable total costs given each checkpointing strategy. Varying t_{split} between these two values therefore effectively allows a tradeoff between cost and volatility given the value of S_{lb} and the checkpointing strategy used, with higher values of t_{split} decreasing cost but increasing volatility, and vice versa.

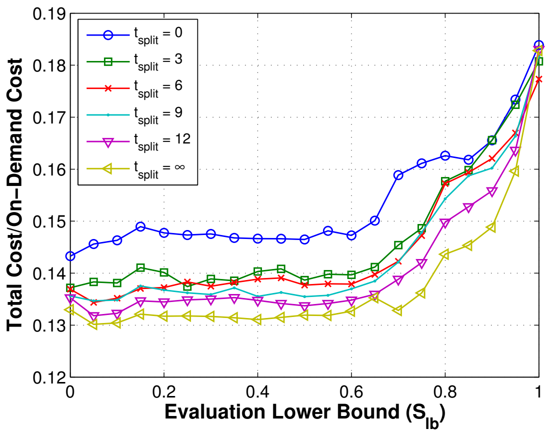
As seen in Figure 5.4, for all values of S_{lb}, t_{split}, and each checkpointing strategy, total costs when incorporating Spot instances are far lower than when using On-Demand



(a) Moldable Jobs, No Checkpointing



(b) Moldable Jobs, Hourly Checkpointing

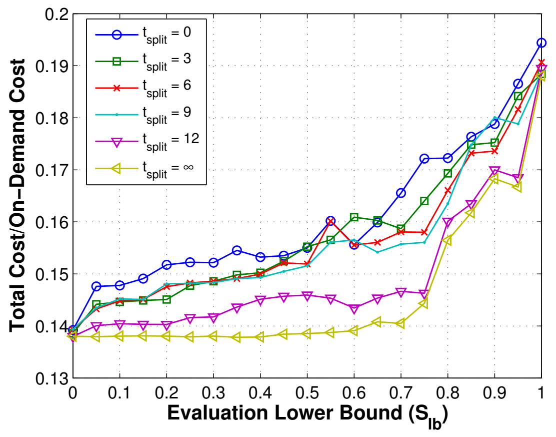


(c) Moldable Jobs, Rising-Price Checkpointing

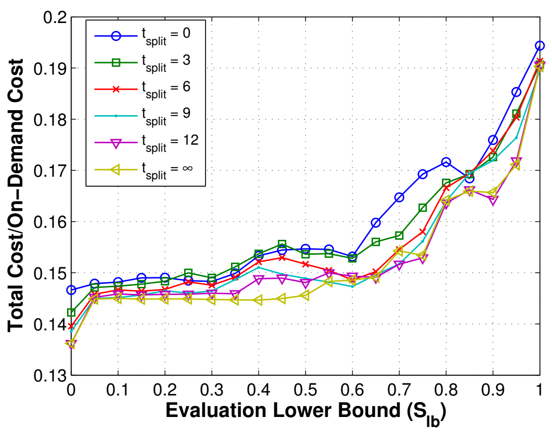
Figure 5.2: The total cost over On-Demand cost using various checkpointing strategies, values of t_{split}, and with moldable jobs.

instances, with costs rising from a minimum of approximately 12.75% to a maximum of 18.5% for moldable jobs, and from 13% to 19.5% for rigid jobs. For both sets of jobs, an hourly checkpointing strategy generally results in the highest total cost due to the extra execution time generated by checkpointing. Furthermore, incorporating no checkpointing strategy generally results in the lowest total cost in most cases, although a rising-market price strategy when using rigid jobs will achieve the lowest costs for $S_{lb} \geq 0.8$.

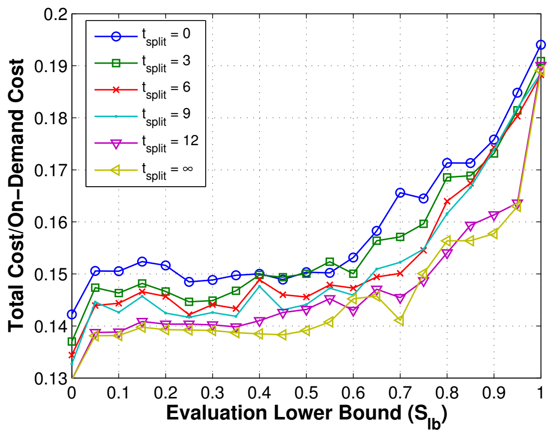
Placing bids as a function of the execution time of a job is inferior, in terms of total cost, when compared to placing bids based on the past overall availability of the instance using that bid, regardless of the length of the job. When all bids are placed as availability bids, total costs decrease by as much as 4% of the total On-Demand



(a) Rigid Jobs, No Checkpointing



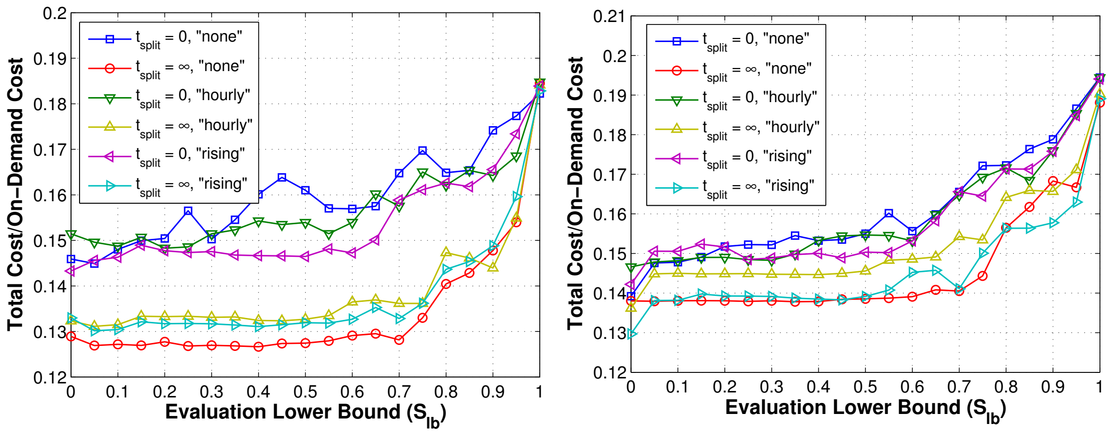
(b) Rigid Jobs, Hourly Checkpointing



(c) Rigid Jobs, Rising-Price Checkpointing

Figure 5.3: The total cost over On-Demand cost using various checkpointing strategies, values of t_{split}, and with rigid jobs.

cost compared to when using only S_{lb}-reliability bids. This decrease in cost is due the interchangeability of jobs and instances inherent in such a bidding strategy. This interchangeability alleviates the necessity of matching an execution time with a bid in order to help prevent early termination, as opposed to finding an overall availability bid. Thus, backfilling jobs and running them on idle instances can be better utilized with such bids due to the fact that instances are matched with jobs regardless of the job's execution time or the instance's bid, therefore decreasing the number of times a new instance is leased while an idle instance can satisfy the deadline.



(a) Moldable Jobs, All Checkpointing

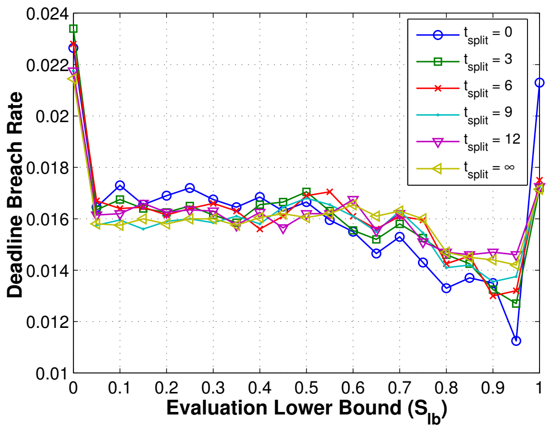
(b) Rigid Jobs, All Checkpointing

Figure 5.4: The total cost over On-Demand cost using various checkpointing strategies, job types, and values of t_{split}.

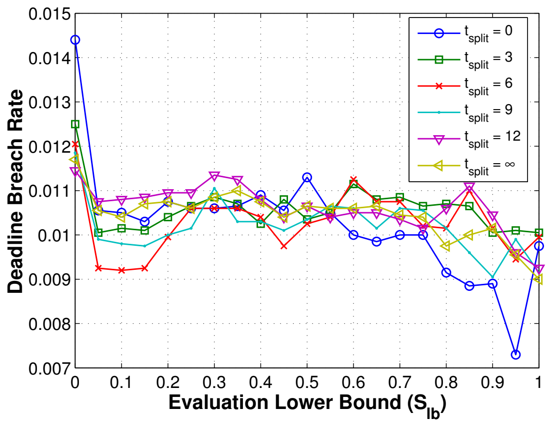
5.6.2 Deadline Breach Rates

When decreasing S_{lb}, for both moldable and rigid jobs, total costs decrease and early-terminations (and thus deadline-breaches) increase. Depending on the value of t_{split} and the checkpointing strategy used, however, it is still possible to maintain very low deadline breach rates as S_{lb} decreases. As seen in Figure 5.5b, using an hourly checkpointing strategy and letting $t_{split} = 0$, for example, allows our approach to still maintain low deadline breaches (as low as 1.1% of all moldable jobs and 0.7% of all rigid jobs) when $S_{lb} = 0.05$, while keeping the total cost equal to 13% and 14.5% of the On-Demand cost for moldable and rigid jobs. Alternatively, in the case of moldable jobs, a rising-market price strategy allows our approach to maintain steady deadline breach rates at around 1.6% of all jobs while incurring lower costs than an hourly checkpointing strategy (see Figures **??**b and 5.5c).

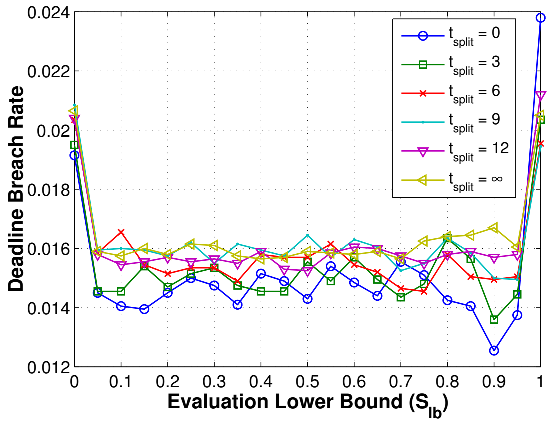
Deadline breaches generally occur very infrequently, with rates achieving a minimum rate of 0.74% when using an hourly checkpointing strategy with moldable jobs, and 0.46% when using no checkpointing strategy and rigid jobs (see Figures 5.5a and 5.6a). Furthermore, as evidenced in Figures 5.5 and 5.6, for all values of the input parameters, deadline breaches achieve a maximum rate of no more than 2.4% for moldable jobs and no more than 0.845% for rigid jobs. For both sets of jobs, hourly checkpointing tends to have the lowest number of deadline breaches. Furthermore, for both sets



(a) No Checkpointing



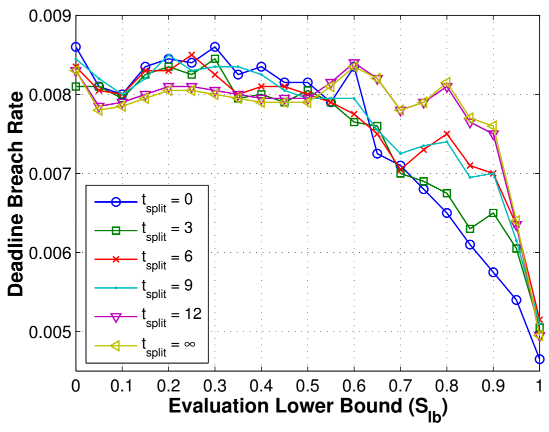
(b) Hourly Checkpointing



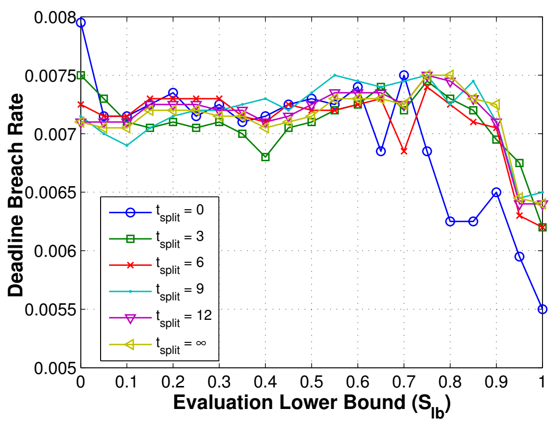
(c) Rising-Price Checkpointing

Figure 5.5: The deadline breach rate using different checkpointing strategies and values of t_{split} with moldable jobs.

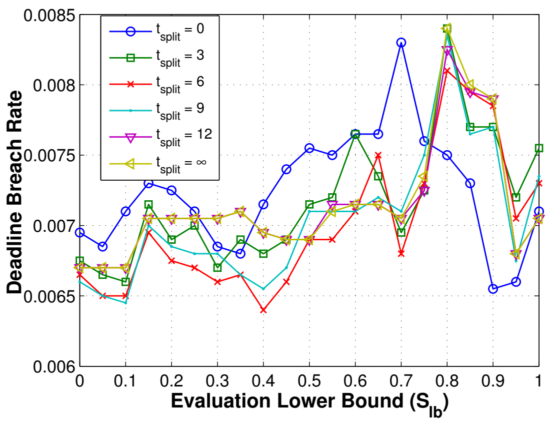
of jobs, increasing S_{lb} and decreasing t_{split} tends to decrease the number of early terminations (see Figure 5.7) and, consequently, the number of deadline breaches. However, when using moldable jobs, letting $S_{lb} = 1$ results in a sharp spike in deadline breaches for the *none* and *rising* checkpointing strategies. This spike is due to the fact that, in many cases, no such instance can be found satisfying this value of S_{lb} while maintaining a reasonable bid and market-price, and thus the job must wait for such an instance to become available. This additional waiting time increases the risk of a deadline breach due to the increased impact of an early-termination after the job is finally assigned to an instance. In addition, each termination has the potential to impact a larger number of jobs than other values of S_{lb}, due to the sparsity of instance types satisfying this constraint.



(a) No Checkpointing



(b) Hourly Checkpointing

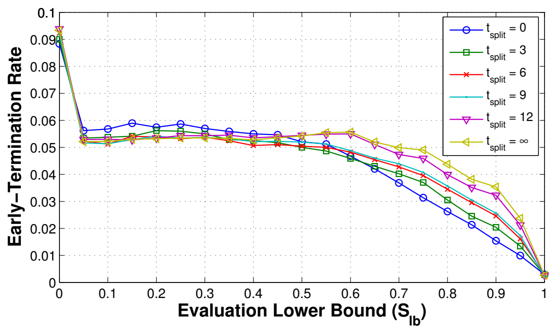


(c) Rising-Price Checkpointing

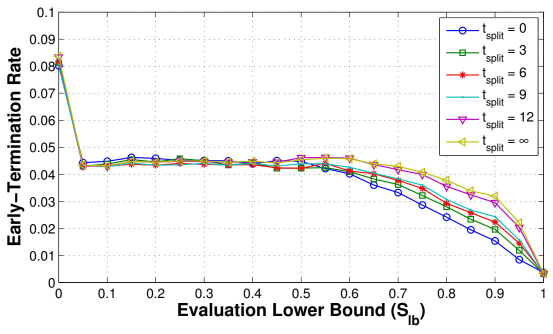
Figure 5.6: The deadline breach rate using different checkpointing strategies and values of t_{split} with rigid jobs.

5.6.3 Early-Termination Rates

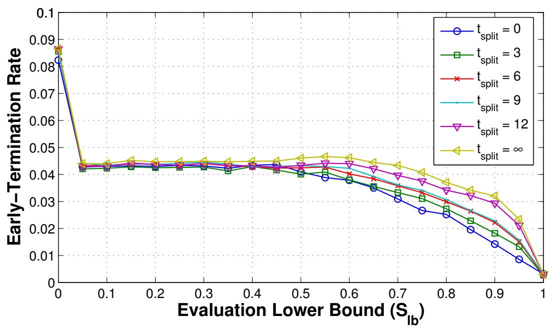
In addition to maintaining low deadline breaches, Figures 5.7 5.8 show that the approach presented in this paper can maintain early-termination rates as low as 0.18% of all jobs when $S_{lb} = 1$, regardless of moldability, checkpointing strategy, and value of t_{split}, while keeping these early-termination rates below 9.5% and 12.5% of moldable and rigid jobs, respectively, when $S_{lb} = 0$. Indeed, while achieving such low early-termination rates, our approach still incurs total costs under 19.5% of the On-Demand cost, regardless of job type. The variation in early-termination rates for different values of t_{split} is highest when S_{lb} is not equal to 0 or 1, and decrease as S_{lb} moves to these values. For lower values of S_{lb} (less than 0.5), all values of t_{split} achieve roughly similar early-termination



(a) No Checkpointing



(b) Hourly Checkpointing



(c) Rising-Price Checkpointing

Figure 5.7: The early-termination rate using different checkpointing strategies and values of t_{split} with moldable jobs.

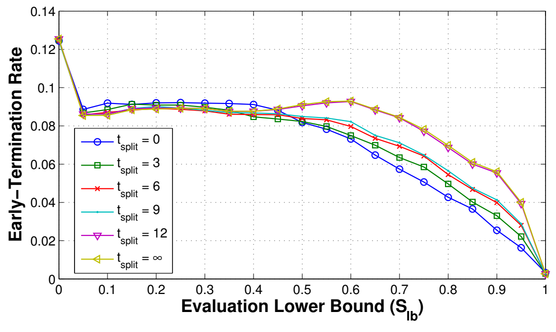
rates, with higher values of t_{split} incurring slightly lower rates. As expected, as S_{lb} increases, however, early termination rates are lowest when leasing all instances with S_{lb}-reliability bids, due to the shift in focus to the successful completion of each job, rather than the overall reliability of the instance.

5.6.4 Average Deadline Exceeded

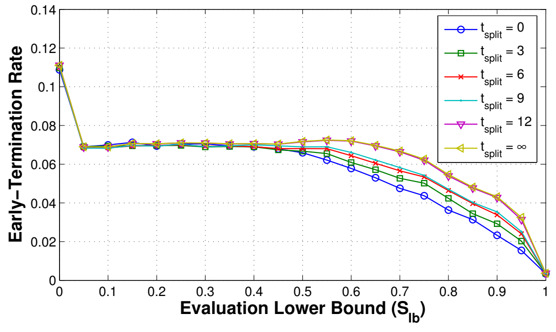
Figure 5.9 illustrates, for those jobs that breached their deadline, the average fraction of this deadline exceeded. That is, this figure illustrates:

$$\xi(S_{lb}, t_{split}, \rho) = \frac{t_j(S_{lb}, t_{split}, \rho)/D_j}{|\{j \mid t_j(S_{lb}, t_{split}, \rho) > D_j\}|}, \tag{5.14}$$

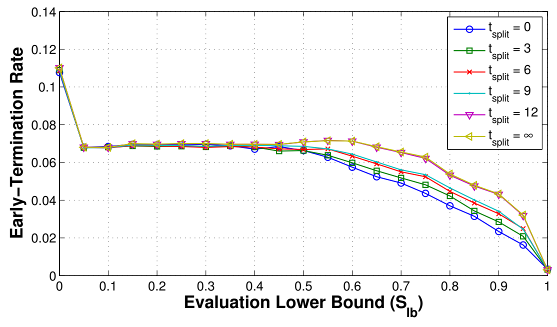
where $t_j(S_{lb}, t_{split})$ is the true (empirical) runtime of job j, using the provided values of S_{lb}, t_{split}, and using the checkpointing strategy $\rho \in \{none, hourly, rising\}$, and D_j is j's given deadline. Thus, $\xi(S_{lb}, t_{split}, \rho)$ reflects the average amount by which a deadline breach exceeds the deadline, for different combinations of S_{lb}, t_{split}, and ρ.



(a) No Checkpointing



(b) Hourly Checkpointing



(c) Rising-Price Checkpointing

Figure 5.8: The early-termination rate using different checkpointing strategies and values of t_{split} with rigid jobs.

For moldable jobs, as illustrated in Figure 5.9(a), when no checkpointing strategy is used, setting t_{split} to 0 has a significant impact in the fraction of deadline exceeded, reducing ξ by as much as 50% in some cases. On the other hand, when checkpointing strategies are used, $t_{split} = \infty$ experiences the lowest values of ξ.

5.7 Discussion

RAMC-DC can achieve cost savings of between 81% and 87% of that incurred when using only On-Demand instances. Regardless of the input parameters or checkpointing strategy used, RAMC-DC can achieve a significant reduction in total costs through strategies incorporating the acquisition and evaluation of Spot instances. Total cost savings range between 81.6% and 87.2% of the equivalent cost using only On-Demand instances, when using moldable jobs, and between 81.2% and 86.9% when using rigid jobs. Such high costs savings are also coupled with very low volatility when the appropriate bidding strategy (via the selection of t_{split} and S_{lb}) and checkpointing strategy

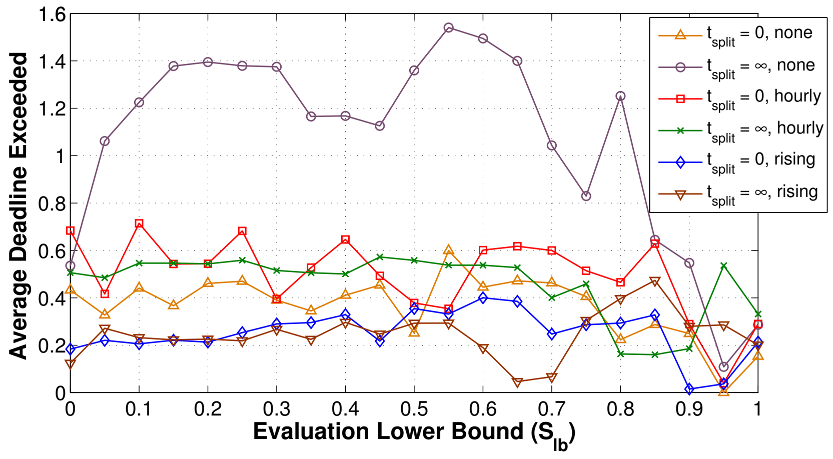


Figure 5.9: The fraction of the execution time by which the deadline was exceeded for early-terminated (moldable) jobs.

are used. These cost savings through RAMC-DC make the use of Spot instances a very cost-efficient alternative to using only On-Demand instances, and for applications that can handle small amounts of volatility, such high cost savings may easily be worthwhile. Furthermore, the highest cost savings are not necessarily coupled with the highest volatility. Letting $S_{lb} = 0.7$, $t_{split} = \infty$, and using an hourly checkpointing strategy sees very low costs coupled with very low deadline breach rates.

Altering the value of t_{split} allows users to effectively trade volatility for total cost. Increasing t_{split} allows RAMC-DC to decrease the total cost by trading higher early-termination rates for lower cost, when using moderate to high values of S_{lb}. Early-termination rates for lower values of S_{lb} are not significantly impacted by the value of t_{split} chosen, due to the fact that the market price is high enough to satisfy most lower values of S_{lb}, and is thus the bid that will be chosen. Deadline breaches for both sets of jobs and all checkpointing strategies tend to achieve both a minimum and maximum when $t_{split} = 0$, and therefore all evaluations are made using the $Reliability$ of the instance, rather than the $Availability$. Such evaluations match instances to specific job execution times, and therefore decrease the overall volatility. In doing so, however, the reusability of each instance among jobs is also decreased, resulting in idle hour blocks that are not taken advantage of.

By increasing S_{lb}, RAMC-DC can significantly decrease, or mitigate, the volatility associated with using Spot instances. Early-termination rates as low as 0.18% can be achieved by letting $S_{lb} = 1$. Although coupled with higher costs, letting $S_{lb} = 1$ results in the lowest early-termination rates and, in the case of rigid jobs, generally the lowest deadline breach rates as well. However, values of S_{lb} approximately equal to 0.7 result in low costs, low deadline breaches, and low early-termination rates, suggesting that high termination resistance may be an inferior goal to an appropriate checkpointing strategy, or to a bid designed to allow early termination when potential costs get too high. Letting $S_{lb} = 1$ results in RAMC-DC choosing a bid that has seen either full availability in the past (if $\widehat{t} \leq t_{split}$), or such that all random samples were executed successfully with that bid ($\widehat{t} > t_{split}$). In either case, such a bid will almost certainly be high enough that most cost fluctuations will not rise above it, and thus RAMC-DC will achieve higher total costs. However, since very high price spikes are relatively infrequent, lowering the value of S_{lb} will increase early-termination rates only slightly, but may have a very large impact on the total cost. Moreover, incorporating an appropriate checkpointing strategy may effectively eliminate any the impact of any volatility from such a decrease.

Periodic checkpointing results in the lowest number of deadline breaches for both rigid and moldable jobs. Deadline breaches when using an hourly checkpointing strategy are generally much lower than both other strategies, and achieve lower minimums as well. Although periodic checkpointing strategies come with higher associated total costs, for moderate values of S_{lb} deadline breach rates are roughly the same for all values of t_{split}. Such a result implies that total costs can be reduced by letting $t_{split} = 0$ and incorporating an hourly checkpointing strategy. The rising price checkpointing strategy did not see very favorable results using these Spot price traces. The reason for this is that prices tend not to fluctuate gradually, but rather experience high spikes that come suddenly and without any previous increase. The results presented here suggest that periodic checkpointing sufficiently mitigates volatility, and the specified period between checkpoints may be tuned depending on the application. Incorporating no checkpointing strategy, however, may still be a viable option if the goal is to provide full access to the instances to a user, or if the applications may not be checkpointed; when incorporating no checkpointing strategy, deadline breaches can still attain a minimum of 1.1% of all moldable jobs and 0.46% of all rigid jobs.

Increasing S_{lb} to any value above 0 *(a market-price bidding strategy) results in a dramatic decrease in early-termination rates and deadline breaches, as well as an increase in total cost.* Similar to the evaluation of RAMP, any shift from a market-price bidding strategy can significantly lower the volatility associated encountered by

RAMC-DC. Although the market-price bidding strategy associated with $S_{lb} = 0$ achieves the lowest costs for all input parameters, it is coupled with a significant increase in the number of jobs surpassing their deadline and the number of early-terminations seen by RAMC-DC. Increasing S_{lb} to any amount above 0 sees large drops in early-terminations (up to nearly 5% with moldable and rigid jobs), as well as in deadline breach rates (up to nearly 0.5% with moldable jobs and 0.1% with rigid jobs). In almost all cases, letting $S_{lb} = 0$ results in the lowest total costs, particularly when assuming all jobs are rigid. When letting $t_{split} = \infty$, however, total costs can actually fall with variation from 0. Assuming all jobs are moldable sees very stable total costs for $S_{lb} \leq 0.7$, but increasing S_{lb} from 0 to 0.05 sees a drop in this total cost of nearly 0.4%.

Chapter 6

Conclusion

The frameworks presented in this thesis take a step towards reliable cloud resource provisioning for the purpose of either profit maximization or cost minimization, given the goal of either request fulfillment or job execution. To fulfill these goals, we develop and evaluate RAMP and RAMC-DC, two resource provisioning frameworks that seek to exploit the large cost and performance diversity within Amazon EC2. Such diversity includes different instance purchasing options, instance types, availability zones, and, when utilizing Spot instances, bids, taking into account cost, reliability, and availability characteristics of these instances. Both RAMP and RAMC-DC seek to exploit the trade-off between service reliability and service cost, taking advantage of the diversity in public cloud services. In addition, both approaches seek to use Amazon's publicly available Spot price history to achieve reliability and/or availability constraints when using Spot instance, and utilize this history in intelligent instance evaluation and bidding strategies when leasing such instances.

Before the construction of either framework, the problem of cost approximation given fluctuating market prices is addressed and five approximation methods are introduced and evaluated using two sets of Spot price traces. Results from this evaluation show that approximating the cost using the current market price is the most accurate method when placing high bids. However, when the bids are close to the market price, trace-based approximations such as incorporating the time-weighted average market price under the bid will yield better accuracy, depending on the amount of fluctuation in the Spot market. Moreover, cost approximation by random sampling is inferior to an average price approximation and incorporating the market price is these trace-based approximations is superior to ignoring it.

RAMP incorporates two novel components: a reliability-aware resource acquisition strategy, and a profit-driven resource allocation algorithm. Together, these components enable RAMP to cost effectively provision resources complying with reliability constraints. Based on our experimental results, successful completion rates are shown to match or exceed almost all values of the desired reliability level, and early-termination rates as low as 2.2% can be achieved. Results also show that significant total profit can be achieved while charging the user only a small fraction of the On-Demand price, increasing the penalty necessitates an increase in reliability to maintain high profits, and increasing the reliability level generally increases total profit. In general, RAMP can completely offset its total cost while charging the user as little as between 15% and 20% of the equivalent cost of each user's requested On-Demand instance. Assuming constant demand, total profits equal to 100% of the baseline total cost can be acquired while charging the user just 35% of the corresponding On-Demand instance price, and total profits equal to 200% of the baseline total cost can be acquired while charging the user 53% of the equivalent On-Demand price.

RAMC-DC utilizes a novel two-tier instance evaluation function that gauges how suitable a Spot or On-Demand instance is to run a job. In addition, RAMC-DC also utilizes a corresponding trace-based instance acquisition strategy that utilizes this evaluation function when determining which instances to acquire. Experimental results show that RAMC-DC provides a cost effective and low-volatility means to execute both moldable and rigid jobs using instance options from Amazon EC2. Moreover, the tunable parameters , S_{lb} and t_{split}, allow a user to effectively trade total cost for volatility. Bids calculated independently of each job's execution time are shown to be more cost effective but also the more failure prone than bids calculated as a function of the job's execution time, and Market Price cost estimation is shown to be generally the most accurate among all methods presented. Additionally, checkpointing can decrease both the total cost and the number of deadline breaches per combination of parameters, with Rising Market Price checkpointing significantly decreasing both when using rigid jobs. RAMC-DC's two-tier instance evaluation function, combined with the ability to search among multiple instance types and availability zones, can achieve early-termination rates as low as 0.18% in a set of 20,000 moldable jobs, and as low as 0.55% in the same set of jobs with the assumption of rigidity. In addition, deadline breaches achieve a *maximum* of no more than 2.4% for moldable jobs and 0.845% for rigid jobs, with breaches in both cases as low as 0.73% and 0.55%, respectively, when using an hourly checkpointing strategy. When achieving such low termination and deadline breach rates, RAMC-DC is able to achieve savings between 80% and 87% of the total cost when using only On-Demand instances.

Future work in this area may involve experiments using actual instances from Amazon EC2. In addition, the effectiveness of a user-specified *priority* parameter designed to influence the amount the user pays, the corresponding reliability lower bound associated with the instance allocated to the user, and the size of the penalty awarded to the user upon early-termination, may be studied. The cost-efficiency of utilizing Reserved instances alongside Amazon's newly introduced Reserved Instance Marketplace Amazon [2013b] can be evaluated, and an evaluation of early-termination rates with different constraints for the construction of the instance type search set, I_r, may be undertaken. Future work may also examine the incorporation of jobs that run across multiple instances and zones, and examining penalties for deadline breaches in order to further influence the choice between Spot and On-demand instances.

Appendix A

Cost Estimation

A.1 Implementations

C++ implementations of all methods to estimate the cost of running a Spot instance for a desired time are given below.

Implementation A.1: Market Price estimation of the cost: $\widehat{C}_{mkt}(t,v)$

```cpp
/*Approximate the cost of running the Spot instance "inst" for "time" full or partial hours using only the current market price.*/
double C_mkt(SpotInstance* inst, double time, double cur_date, double alpha){
  unsigned int inst_idx = inst->get_instance();
  unsigned int zone_idx = inst->get_avail();
  double mkt_price = get_mkt_price(inst_idx, zone_idx, cur_date);
  return mkt_price*ceil(time);
}
```

Implementation A.2: Monte Carlo estimation of the cost: $\widehat{C}_{mc}(t,v)$

```cpp
/*Approximate the cost of running the Spot instance "inst" for "time" full or partial hours using a Monte Carlo estimate.*/
double C_monte(SpotInstance* inst, double time, double cur_date){
  unsigned int inst_idx = inst->get_instance();
  unsigned int zone_idx = inst->get_avail();
  double mkt_price = get_mkt_price(inst_idx, zone_idx, cur_date);
  pair<int, int> end_start = window_search(dates[inst_idx][zone_idx], cur_date, truncate_value+time);
  int end_idx = end_start.first;
  int start_idx = end_start.second;
```

```
if(end_idx == -1 || start_idx == -1 || dates[inst_idx][zone_idx][end_idx] - dates[inst_idx][zone_idx][start_idx] <= 0){
  return mkt_price*ceil(time);
}
double start = max(dates[inst_idx][zone_idx][end_idx] - truncate_value - time, dates[inst_idx][zone_idx][start_idx]);
double width = cur_date - start - time;
if(width <=0){
  return mkt_price*ceil(time);
}
//create NUM_DATES random dates
vector<double> rand_dates = random_array1(NUM_DATES, dates[inst_idx][zone_idx][start_idx], dates[inst_idx][zone_idx][end_idx]);
vector<double> cost(rand_dates.size(), prices[inst_idx][zone_idx][end_idx]*ceil(adj_comp_time));
//get the true cost for every date
for(unsigned int k = 0; k < rand_dates.size(); ++k){
  int split_idx = binary_search_vector(dates[inst_idx][zone_idx], rand_dates[k], start_idx, end_idx);
  if(split_idx == end_idx || split_idx == -1){
    continue;
  }
  double mkt_price = prices[inst_idx][zone_idx][split_idx];
  if(mkt_price > bid){
    continue;
  }
  vector<double>::iterator above_iter = find_if(prices[inst_idx][zone_idx].begin() + split_idx +1, prices[inst_idx][zone_idx].begin() + end_idx+1, greater_than(bid));
  double true_step;
  if(above_iter == prices[inst_idx][zone_idx].begin() + end_idx +1){
    true_step = adj_comp_time;
  }else{
    unsigned int above_idx = distance(prices[inst_idx][zone_idx].begin(), above_iter);
    true_step = dates[inst_idx][zone_idx][above_idx] - rand_dates[k];
  }
  if(true_step < adj_comp_time){
    cost[k] = prices[inst_idx][zone_idx][split_idx]*ceil(adj_comp_time);
    continue;
  }
  int interm_end_idx = binary_search_vector(dates[inst_idx][zone_idx], rand_dates[k] + adj_comp_time, split_idx, end_idx);
  if(interm_end_idx == -1){
    interm_end_idx = end_idx;
  }
  if(interm_end_idx == split_idx){
    if(true_step < ceil(adj_comp_time) && true_step >= adj_comp_time){
      cost[k] = mkt_price * floor(adj_comp_time);
    } else if(true_step >= ceil(adj_comp_time)){
      cost[k] = mkt_price * ceil(adj_comp_time);
      }else{
        cost[k] = mkt_price*floor(true_step);
      }
    }else{
      cost[k] = instance_cost(mkt_price, true_step, rand_dates[k], adj_comp_time, ↩
          inst_idx, zone_idx, split_idx, interm_end_idx);
    }
  }
  //return the mean of these samples
  return mean(cost);
}
```

Implementation A.3: Average Price estimation of the cost: $\widehat{C}_{avg}(t,\nu)$

```
/*Approximate the cost of running the Spot instance "inst" for "time" full or ↩
    partial hours using an Average Price estimate.*/
double C_avg(SpotInstance* inst, double time, double cur_date){
  unsigned int inst_idx = inst->get_instance();
  unsigned int zone_idx = inst->get_avail();
  double mkt_price = get_mkt_price(inst_idx, zone_idx, cur_date);
  pair<int, int> end_start = window_search(dates[inst_idx][zone_idx], cur_date, ↩
      truncate_value);
  int end_idx = end_start.first;
  int start_idx = end_start.second;
  if(end_idx == -1 || start_idx == -1 || end_idx - start_idx <= 0){
    return mkt_price*ceil(time);
  }
  double start = max(dates[inst_idx][zone_idx][end_idx] - truncate_value, dates[↩
      inst_idx][zone_idx][start_idx]);
  double width = cur_date - start;
  if(width <=0){
    return mkt_price*ceil(time);
  }
  double interm_prices[end_idx - start_idx];
  double total_time_under_bid = 0.0;
  //assign the prices and corresponding times
  for(int i = start_idx; i < end_idx; ++i){
    interm_prices[i-start_idx] = -1.0;
    times[i-start_idx] = -1.0;
    if(prices[inst_idx][zone_idx][i] <= inst->get_bid());
      interm_prices[i-start_idx] = prices[inst_idx][zone_idx][i];
      times[i-start_idx] = dates[inst_idx][zone_idx][i+1] - prices[inst_idx][↩
          zone_idx][i];
      total_time_under_bid += times[i-start_idx];
    }
  }
  double avg_hour_price = 0.0;
  //take the time-weighted average of the prices
  for(int i = 0; i < idx_diff; ++i){
    if(interm_prices[i] != -1.0){
      avg_hour_price += interm_prices[i]*(times[i]/total_time_under_bid);
    }
  }
  return avg_hour_price*ceil(time);
}
```

Implementation A.4: Market-Monte Carlo estimation of the cost: $\widehat{C}_{mmc\_\alpha}(t,\nu)$

```
/*Approximate the cost of running the Spot instance "inst" for "time" full or ↩
    partial hours using the market price and a Monte Carlo estimate.*/
double C_mmc_alpha(SpotInstance* inst, double time, double cur_date, double alpha){
    unsigned int inst_idx = inst->get_instance();
    unsigned int zone_idx = inst->get_avail();
    double mkt_price = get_mkt_price(inst_idx, zone_idx, cur_date);

    if(time <= alpha){
        return ceil(alpha)*mkt_price;
    }
    pair<int, int> end_start = window_search(dates[inst_idx][zone_idx], cur_date, ↩
        truncate_value+time);
    int end_idx = end_start.first;
    int start_idx = end_start.second;
    if(end_idx == -1 || start_idx == -1 || end_idx - start_idx <= 0){
        return mkt_price*ceil(time);
    }
    return C_monte(inst, ceil(time-ceil(alpha)), cur_date) + ceil(alpha)*mkt_price;
}
```

Implementation A.5: Market-Average estimation of the cost: $\widehat{C}_{ma\_\alpha}(t,\nu)$

```
/*Approximate the cost of running the Spot instance "inst" for "time" full or ↩
    partial hours using the market price and an Average Price estimate.*/
double C_ma_alpha(SpotInstance* inst, double time, double cur_date, double alpha){
    unsigned int inst_idx = inst->get_instance();
    unsigned int zone_idx = inst->get_avail();
    double mkt_price = get_mkt_price(inst_idx, zone_idx, cur_date);
    if(time <= alpha){
        return ceil(time)*mkt_price;
    }
    pair<int, int> end_start = window_search(dates[inst_idx][zone_idx], cur_date, ↩
        truncate_value);
    int end_idx = end_start.first;
    int start_idx = end_start.second;
    if(end_idx == -1 || start_idx == -1 || dates[inst_idx][zone_idx][end_idx] - ↩
        dates[inst_idx][zone_idx][start_idx] <= 0){
        return mkt_price*ceil(time);
    }
    double start = max(dates[inst_idx][zone_idx][end_idx] - truncate_value, dates[
        inst_idx][zone_idx][start_idx]);
    double width = cur_date - start;
    if(width <=0){
        return mkt_price*ceil(time);
    }
    double interm_prices[end_idx - start_idx];
    double total_time_under_bid = 0.0;
    double last_mkt_price = prices[inst_idx][zone_idx][start_idx];
    for(int i = start_idx; i < end_idx; ++i){
      interm_prices[i-start_idx] = -1.0;
  times[i-start_idx] = -1.0;
  if(prices[inst_idx][zone_idx][i] <= inst->get_bid()){
    interm_prices[i-start_idx] = prices[inst_idx][zone_idx][i];
    times[i-start_idx] = dates[inst_idx][zone_idx][i+1] - dates[inst_idx][zone_idx][
        i];
    total_time_under_bid += times[i-start_idx];
  }
    }
    double avg_hour_price = 0.0;
    for(int i = 0; i < idx_diff; ++i){
        if(interm_prices[i] != -1.0){
            avg_hour_price += interm_prices[i]*(times[i]/total_time_under_bid);
        }
    }
    return avg_hour_price*ceil(time - ceil(alpha)) + ceil(alpha)*mkt_price;
}
```

Implementation A.6: Market-Monte Carlo estimation of the cost using the average inter-price time: $\widehat{C}_{mmc\_avg}(t,v)$

```
/*Approximate the cost of running the Spot instance "inst" for "time" full or
    partial hours using the market price and a Monte Carlo estimate with the "avg"
    parameter.*/
double C_mmc_avg(SpotInstance* inst, double time, double cur_date){
    unsigned int inst_idx = inst->get_instance();
    unsigned int zone_idx = inst->get_avail();
    double mkt_price = get_mkt_price(inst_idx, zone_idx, cur_date);
    pair<int, int> end_start = window_search(dates[inst_idx][zone_idx], cur_date,
        truncate_value+time);
    int end_idx = end_start.first;
    int start_idx = end_start.second;
    if(end_idx == -1 || start_idx == -1 || end_idx - start_idx <= 0){
        return mkt_price*ceil(time);
    }
    double start = max(dates[inst_idx][zone_idx][end_idx] - truncate_value - time,
        dates[inst_idx][zone_idx][start_idx]);
    double width = cur_date - start - time;
    if(width <=0){
        return mkt_price*ceil(time);
    }
    //calculate the average inter-price time
    double avg_inter_price_time = 0.0;
    int num_changes = 0;
    double last_mkt_price = prices[inst_idx][zone_idx][start_idx];
    for(int i = start_idx; i < end_idx; ++i){
        if(prices[inst_idx][zone_idx][i] == last_mkt_price){
            avg_inter_price_time += dates[inst_idx][zone_idx][i+1] - dates[inst_idx↩
                ][zone_idx][i];
        }else{
            last_mkt_price = prices[inst_idx][zone_idx][i];
            num_changes++;
        }
    }
    //prevent division by zero
    if(num_changes == 0){
        avg_inter_price_time = numeric_limits<double>::max();
    }else{
        avg_inter_price_time /= num_changes;
    }
    if(ceil(time) <= ceil(avg_inter_price_time)){
        return mkt_price*ceil(time);
    }
    vector<double> rand_dates = random_array1(NUM_DATES, dates[inst_idx][zone_idx][↩
        start_idx], dates[inst_idx][zone_idx][end_idx]);
    return C_monte(inst->get_bid(), start_idx, end_idx, rand_dates, time- ceil(↩
        avg_inter_price_time), inst_idx, zone_idx) + mkt_price*ceil(↩
        avg_inter_price_time);
}
```

Implementation A.7: Market-Average estimation of the cost using the average inter-price time: $\widehat{C}_{ma\_avg}(t,\nu)$

```
/*Approximate the cost of running the Spot instance "inst" for "time" full or ↩
    partial hours using the market price and an Average Price estimate with the "avg↩
    " parameter.*/
double C_ma_avg(SpotInstance* inst, double time, double cur_date){
    unsigned int inst_idx = inst->get_instance();
    unsigned int zone_idx = inst->get_avail();
    double mkt_price = get_mkt_price(inst_idx, zone_idx, cur_date);
    pair<int, int> end_start = window_search(dates[inst_idx][zone_idx], cur_date, ↩
        truncate_value);
    int end_idx = end_start.first;
    int start_idx = end_start.second;
    if(end_idx == -1 || start_idx == -1 || dates[inst_idx][zone_idx][end_idx] - ↩
        dates[inst_idx][zone_idx][start_idx] <= 0){
        return mkt_price*ceil(time);
    }
    double start = max(dates[inst_idx][zone_idx][end_idx] - truncate_value, dates[inst_idx][zone_idx][start_idx]);
    double width = cur_date - start;
    if(width <=0){
        return mkt_price*ceil(time);
    }
    int idx_diff = end_idx - start_idx;
    double interm_prices[idx_diff];
    double times[idx_diff];
    for(int i = 0; i < idx_diff; ++i){
        interm_prices[i] = -1.0;
        times[i] = -1.0;
    }
    //calculate average inter-price time
    double total_time_under_bid = 0.0;
    double avg_inter_price_time = 0.0;
    int num_changes = 0;
    double last_mkt_price = prices[inst_idx][zone_idx][start_idx];
    for(int i = start_idx; i < end_idx; ++i){
        if(prices[inst_idx][zone_idx][i] == last_mkt_price){
            avg_inter_price_time += dates[inst_idx][zone_idx][i+1] - dates[inst_idx][zone_idx][i];
        }else{
            last_mkt_price = prices[inst_idx][zone_idx][i];
            num_changes++;
        }
        if(prices[inst_idx][zone_idx][i] <= inst->get_bid()){
            interm_prices[i-start_idx] = prices[inst_idx][zone_idx][i];
            times[i-start_idx] = dates[inst_idx][zone_idx][i+1] - dates[inst_idx][zone_idx][i];
            total_time_under_bid += times[i-start_idx];
        }
    }
    if(num_changes == 0){
        avg_inter_price_time = numeric_limits<double>::max();
    }else{
        avg_inter_price_time /= num_changes;
    }
    if(ceil(time) <= ceil(avg_inter_price_time)){
        return mkt_price * ceil(time);
    }
    double avg_hour_price = 0.0;
    for(int i = 0; i < idx_diff; ++i){
        if(interm_prices[i] != -1.0){
            avg_hour_price += interm_prices[i]*(times[i]/total_time_under_bid);
        }
    }
    return avg_hour_price*ceil(time - ceil(avg_inter_price_time)) + mkt_price*ceil(avg_inter_price_time);
}
```

Appendix B

RAMP

B.1 Total Profits Assuming Linear Demand

In addition to the assumption of constant demand in our evaluation of RAMP in Section 5.4, we may also assume different demand curves that may depend on a number of input parameters. Two examples of such demand curves are the following:

- **Linear Demand:** Linear demand curves for a single pricing parameter, p, are typically of the form $n_D = a - b \cdot p$, for constants a and b. For our experiments, in order to focus the results on reasonable pricing ranges, we would like to observe demand within an upper and lower bound, ub and lb, respectively. Therefore, we will use the demand function:

$$n_D = (ub - p) \cdot \frac{n}{ub - lb},$$

  for $p \in [lb, ub]$, with $lb, ub \in [0,1]$ and $lb \leq ub$. Thus, demand increases linearly from $n_D = 0$ when $p = ub$ to $n_D = n$ when $p = lb$.

- **Exponential Demand:** Exponential demand curves are typically of the form $n_D = a \cdot e^{-b \cdot p}$. As with Linear Demand, we wish to view results within an upper and lower bound and so we obtain the demand function: $n_D = n^{\frac{p-ub}{lb-ub}}$. Therefore, demand increases exponentially from $n_D = 1$ when $p = ub$ to $n_D = n$ when $p = lb$.

Within this section, we will perform a rudimentary evaluation of the total profit of RAMP when using the linear demand curve as discussed above. Here, n_D represents the

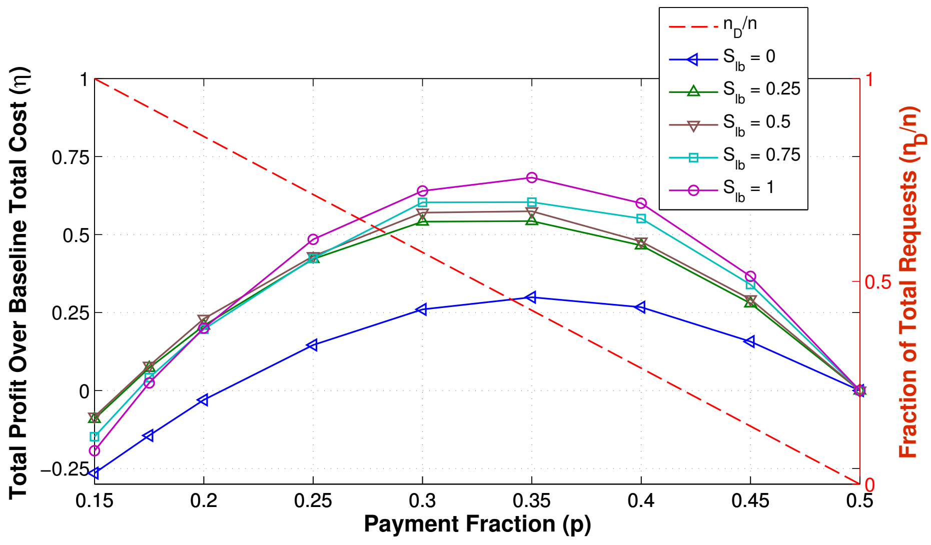


Figure B.1: Measure of total profit for different values of p given specific values of S_{lb} and with $e = 0.25$, using a linear demand function.

number of requests RAMP encounters for a given value of f, lb, ub, and n. Therefore, when assuming a linear demand curve, all 20,000 requests will be seen when $p = lb$ and no requests will be seen when $p = ub$. In this paper, we will let lb be equal to 0.15, the approximate value of p at which total profits fall below 0 when using constant demand, and we will let ub = 0.5, an historical average cost savings fraction for applications making use of Spot instances rather than On-Demand instances (cf. Andrzejak et al. [2010], Voorsluys and Buyya [2012], Voorsluys et al. [2011], Liu [2011]). Once n_D is calculated, the requests are randomly shuffled and the first n_D are used. The demand curves in these experiments are used as rough estimates to examine the price vs. quantity comparison for RAMP; actual demand curves will likely depend on other parameters such as S_{lb}, e, etc.

Figure B.1 illustrates η for $p \in [0.15, 0.5]$, and with various values of e, when the number of requests, n_D, varies linearly with the payment fraction p. When using this linear demand function, η attains a maximum of approximately 0.7 when $p = 0.35$, and less than half of all 20,000 possible requests are seen. For $p < 0.35$, profit falls with η as low as -0.25 when $S_{lb} = 0$. As with constant demand, profits fall below zero when p falls below approximately 0.175, and $S_{lb} = 0$ generally produces the lowest profit among all values.

B.2 Implementations

Implementations of several algorithms introduced for RAMP are included below.

Implementation B.1: Using a Kaplan Meier Estimator to evaluate Spot instance reliability: *reliability*(r,v)

```
/*Get the reliability of an instance defined by <bid, inst_idx, zone_idx> using a ↩
    Kaplan Meier estimator*/
double reliability(double bid, unsigned int start_idx, unsigned int end_idx, vector<↩
    double> & rand_dates, double time, int inst_idx, int zone_idx){
  vector<double> temp_step(rand_dates.size());
  for(unsigned int k = 0; k < rand_dates.size(); ++k){
    int split_idx = binary_search_vector(dates[inst_idx][zone_idx], rand_dates[k], ↩
        start_idx, end_idx);
    if(split_idx == -1){
      continue;
    }
    if(split_idx == end_idx){
      temp_step[k] = time;
      continue;
    }
    double mkt_price;
    mkt_price = prices[inst_idx][zone_idx][split_idx];
    if(mkt_price > bid){
      temp_step[k] = 0;
      continue;
    }
    vector<double>::iterator above_iter = find_if(prices[inst_idx][zone_idx].begin()↩
         + split_idx +1, prices[inst_idx][zone_idx].begin() + end_idx+1, ↩
        greater_than(bid));
    if(above_iter == prices[inst_idx][zone_idx].begin() + end_idx + 1){
      temp_step[k] = time;
    }else{
      unsigned int above_idx = distance(prices[inst_idx][zone_idx].begin(), ↩
          above_iter);
      temp_step[k] = dates[inst_idx][zone_idx][above_idx] - rand_dates[k];
    }
  }
  sort(temp_step.begin(), temp_step.end());
  double score = 1;
  for(unsigned int l = 0; l < temp_step.size(); ++l){
    if(temp_step[l] >= time){
      break;
    } else {
            double nj = temp_step.size() - l;
      score = score*(nj-1)/nj;
    }
  }
  return score;
}
```

Implementation B.2: Implementation of the cost estimation algorithm used by RAMP to calculate the estimated cost in the event of failure

```
/*Estimated cost of execution in the event of failure*/
double C_mkt_fail(SpotInstance* inst, job* job_ptr, double run_time, double cur_date↩
    ){
  unsigned int inst_idx = inst->get_instance();
  unsigned int zone_idx = inst->get_avail();
      double mkt_price = get_mkt_price(inst_idx, zone_idx, cur_date);
      pair<int, int> end_start = window_search(dates[inst_idx][zone_idx], cur_date, ↩
          truncate_value+run_time);
      int end_idx = end_start.first;
      int start_idx = end_start.second;
      if(end_idx == -1 || start_idx == -1 || dates[inst_idx][zone_idx][end_idx] - ↩
          dates[inst_idx][zone_idx][start_idx] <= 0){
         return mkt_price*ceil(run_time);
      }
      double start = max(dates[inst_idx][zone_idx][end_idx] - truncate_value - ↩
          run_time, dates[inst_idx][zone_idx][start_idx]);
      double width = cur_date - start - run_time;
      if(width <=0){
          return mkt_price*ceil(run_time);
      }
      vector<double> rand_dates = random_array1(NUM_DATES, dates[inst_idx][zone_idx↩
          ][start_idx], dates[inst_idx][zone_idx][end_idx]);
      double bid = inst->get_bid();
      double mean_avail_time = 0.0;
      unsigned int num_fails = 0;
  for(unsigned int k = 0; k < rand_dates.size(); ++k){
    int split_index = binary_search_vector(dates[inst_idx][zone_idx], rand_dates[k],↩
         start_idx, end_idx);
            if(split_index == end_idx || split_index == -1){
                continue;
            }
            double mkt_price = prices[inst_idx][zone_idx][split_index];
    if(mkt_price > bid){
      continue;
    }
            vector<double >::iterator above_iter = find_if(prices[inst_idx][zone_idx↩
                ].begin() + split_index +1, prices[inst_idx][zone_idx].begin() + ↩
                end_idx+1, greater_than(bid));
            double true_step;
            if(above_iter == prices[inst_idx][zone_idx].begin() + end_idx +1){
              true_step = run_time;
            }else{
                unsigned int above_index = distance(prices[inst_idx][zone_idx].begin↩
                    (), above_iter);
                true_step = dates[inst_idx][zone_idx][above_index] - rand_dates[k];
            }
            if(true_step < run_time){
                mean_avail_time += true_step;
                num_fails++;
                continue;
            }
  }
      if(num_fails == 0){
            num_fails = 1; // prevent division by 0
      }
  mean_avail_time /= static_cast<double>(rand_dates.size());
  return mean_avail_time*mkt_price;
}
```

Appendix C

RAMC-DC

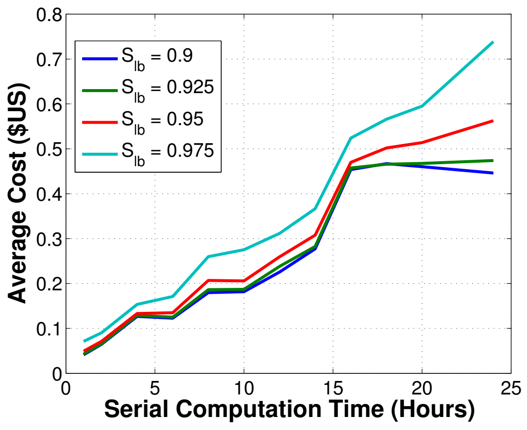
C.1 Preliminary Results Using Amdahl's Law

In this section, we present an analysis of the Kaplan-Meier instance evaluation and bidding strategy when Amdahl's Law is used instead of Downey Speedup Model.

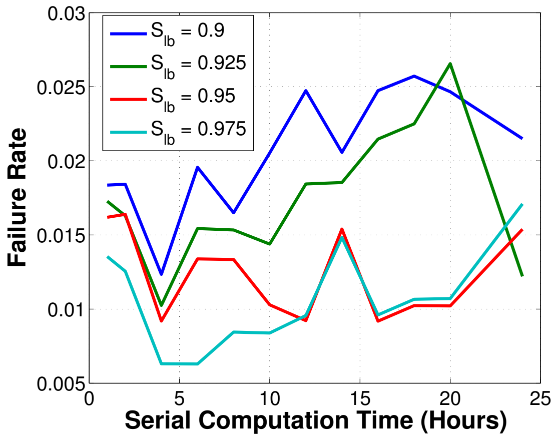
C.1.1 Evaluation and Bidding

Given a confidence constraint $c \in [0,1]$, per-EC2-Compute-Unit bid upper bound, *ub*, and set of tasks, Λ, for each task $\lambda_i \in \Lambda$, the user submits a runtime, t, a requested number of cores, n, a parallelizable fraction, *par*, and a deadline d. Additionally, the user is able to submit groups of tasks simultaneously so that subsets of the group may be run in parallel on a single instance, if possible. For each job, or group of jobs, at a start time, s, determined by the resource allocation algorithm, a search among Spot instance types $I = \{i_1, \ldots, i_j\}$ and availability zones $Z = \{z_1, \ldots, z_k\}$ is conducted to find the optimal bid price, instance type, and zone triple, $\langle i, z, b \rangle$. For each instance type, an adjusted runtime is calculated as t_i, determined by Amdahl's Law with runtime t for the number of EC2 Compute Units, n, and parallelizable fraction *par*. The triple $\langle i, z, b \rangle$ is first found by finding the lowest bids for each instance type and availability zone, such that b is within the upper bound and satisfies the confidence constraint

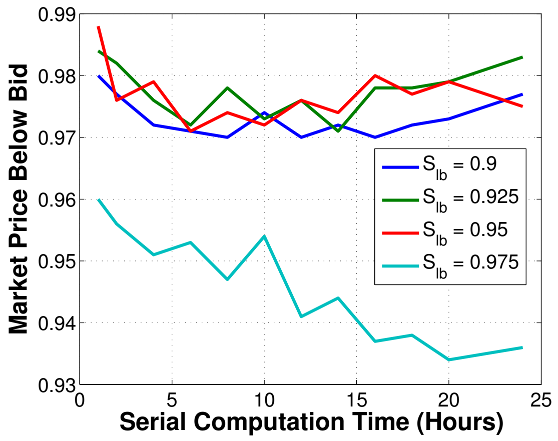
$$P(\widehat{t_{i,z,b}} \geq t_i) \geq S_{lb}, \tag{C.1}$$



(a) Average cost for four confidence values, using random starting dates from after July 2011. Average costs have only a slight difference for lower computation times.



(b) Failure rates for four confidence values, using random starting dates from after July 2011. All failure rates are below $1-c$.



(c) The fraction of computations that started (i.e. the optimal bid was below the market price) for four confidence values, using random starting dates from after July 2011

Figure C.1: Deadline Breaches (Moldable Jobs)

where $\widehat{t_{i,z,b}}$ is the true length of time for which the Spot instance is available to the user. The instance and availability zone with the lowest estimated cost is then determined by the Monte Carlo Estimate

$$\widehat{C}(t_i, i, z, b) = \frac{1}{|X|} \sum_{x \in X} C_x(t_i, i, z, b), \tag{C.2}$$

provided the bid is above the market price. In the above equations, X is a set of random samples from some distribution $\Omega(s-w, s)$ (in the result below, a right-truncated exponential distribution was used), with window w calculated as the minimum of some global truncation parameter and the total length of the price history before s_i, using instance i in zone z.

In Equation 1, $P(\widehat{t_{i,z,b}} \geq t_i)$ is found using a modified Kaplan Meier estimator, $\widehat{S}(t_i, i, z, b)$, over the set of step lengths starting at each $x \in X$, using instance i, zone z, and bid b. Thus, the confidence constraint requires $P(\widehat{t_{i,z,b}} \geq t_i) \geq S_{lb}$.

The cost approximation, C, function in Equation 2 is calculated by finding the true cost when instance i is run in zone z, starting at x and with bid b, and is not terminated before $x + t_{i_i}$. If early termination occurs, or the market price is initially higher than the optimal bid, C represents the cost of running the task on the equivalent On-Demand instance, in order to include some representation of the negative effects of out-of-bid situations.

Since the empirical CDF in Equation 1 is a monotonically decreasing function of the bid price (higher bid prices never imply shorter step lengths), locating the optimal bid, or proving that one does not exist below the upper bound, is relatively simple.

Figures C.1a, C.1b, and C.1c show some preliminary results from simulations run only on Spot instances, using random start dates. In all cases, success rates are above the confidence constraint, average costs are low, and the number of instances that were not initiated (the market price was above the bid) and thus need to be run on statically-priced instance types, or placed in a waiting queue, is small.

C.2 Results Using Earlier Spot Price Traces

Due to the fluctuating nature of Spot instance market prices, results from simulations run at different times can be slightly different than those presented in Section 5.6. Thus, in the section we will present an evaluation of RAMC-DC using simulations run with Spot market prices from February through June, 2012.

C.2.1 Results

Results from the experiments described in the previous section are given below. These results are split into three categories comparing: each cost estimation strategy; the results of different parameters using the scheduling and resource allocation strategy for moldable jobs; and the results of different parameters using the scheduling and resource allocation strategy for rigid jobs.

C.2.1.1 Moldable Jobs

Total costs for the set of moldable jobs for each checkpointing strategy, using various values of the confidence level, S_{lb}, and the short-long split parameter, t_{split}, are shown in Figure C.2. As seen in the figures, total costs tend to decrease as t_{split} increases (with the lowest costs occurring when every instance is started with its availability bid) and as the confidence level decreases. Not implementing a checkpointing strategy tends to yield the lowest cost for low values of t_{split}. However, when $t_{split} \geq 12$, Hourly checkpointing replaces no checkpointing as the lowest cost strategy. The total cost when using only On-Demand instances was \$15,438.40, reflecting savings of between 56% and 68.5% when using Spot instances. For more concrete results, abbreviated tables providing an example of the total costs for different combinations of S_{lb} and t_{split} are provided in Table C.1 for moldable jobs and in Table C.2 for rigid jobs. Results in both tables were observed using Hourly checkpointing, which was chosen due to the similarities in total cost characteristics among all checkpointing strategies, and to provide a direct comparison between both sets of jobs.

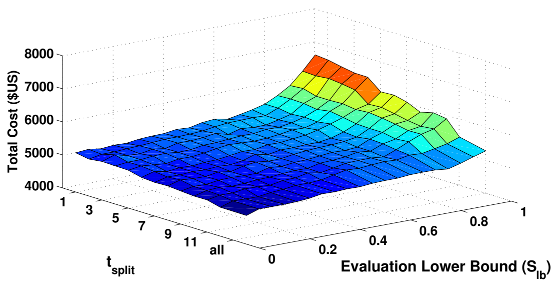
Although one may intuitively expect checkpointing to be useful in lowering total cost and decreasing deadline breaches when running jobs in the face of the volatility inherent in Amazon's Spot instances, the results observed for moldable jobs suggest that for lower values of t_{split}, the added execution time resulting from checkpointing

Table C.1: Total costs for moldable jobs using an Hourly checkpointing strategy

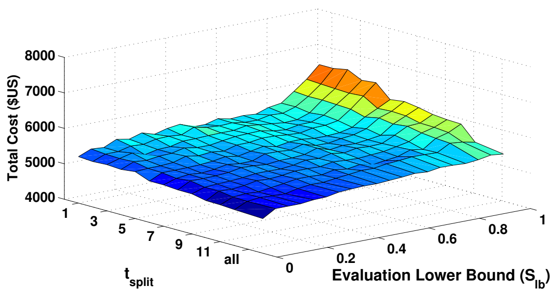
	S_{lb}						
t_{split}	0	0.1	0.25	0.5	0.75	0.9	0.95
1	5186	5167	5242	5388	5596	6380	6802
3	5168	5207	5205	5275	5533	6398	6684
5	5145	5168	5197	5316	5540	6293	6609
7	5021	5104	5146	5291	5480	6012	6312
9	5000	5060	5111	5240	5505	5857	6039
11	4973	5045	5066	5228	5426	5845	6082
ALL	4867	4991	5034	5130	5179	5360	5473

Table C.2: Total costs for rigid jobs using an Hourly checkpointing strategy

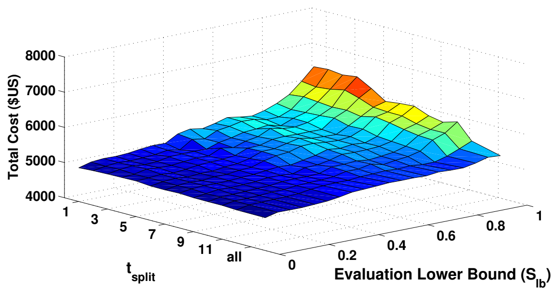
	S_{lb}						
t_{split}	0	0.1	0.25	0.5	0.75	0.9	0.95
1	8045	8330	8366	8674	9152	10525	10984
3	7981	8318	8357	8699	9060	10222	10692
5	7918	8246	8306	8580	9039	10091	10483
7	7835	8202	8268	8504	8929	9601	9973
9	7805	8113	8190	8474	8860	9512	9765
11	7752	8030	8108	8273	8741	9382	9647
ALL	7723	7980	8044	8194	8449	8830	8970



(a) Hourly Checkpointing

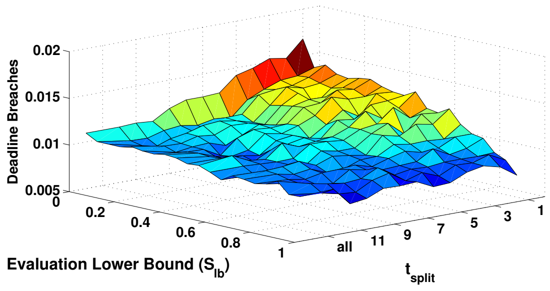


(b) Rising Market Price Checkpointing

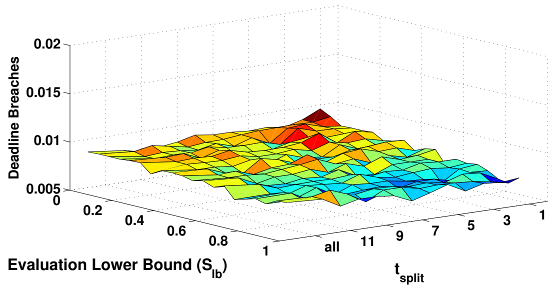


(c) No Checkpointing

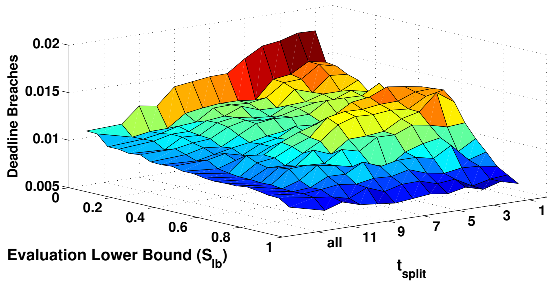
Figure C.2: Total Costs (Moldable Jobs)



(a) Hourly Checkpointing



(b) Rising Market Price Checkpointing

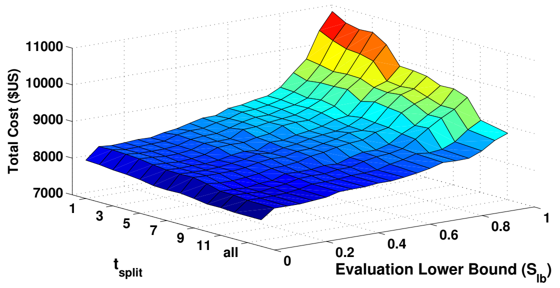


(c) No Checkpointing

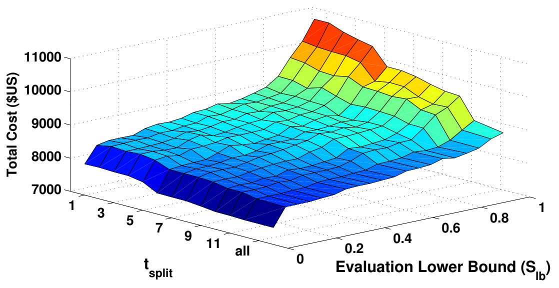
Figure C.3: Deadline Breaches (Moldable Jobs)

can increase the total cost more than that incurred when a job needs to be restarted from scratch. However, for $t_{split} \geq 12$ hours, Hourly checkpointing becomes the lowest cost option available for $S_{lb} \leq 0.9$, achieving the minimum cost of \$4,867.15. Thus, Hourly checkpointing on longer jobs, and with availability bids for all jobs, may be a suitable method to keep both the impact of early terminations, and thus the total cost, low. Furthermore, as seen in Figure C.7a, when $S_{lb} = 0.95$ and $t_{split} = 1$, having no checkpointing strategy results in both the lowest cost and the lowest number of deadline breaches, with a total cost of \$6504.95 with deadline breaches in 0.65% of all jobs, compared to \$6802.12 and 0.735% with Hourly Checkpointing, and \$6608.62 and 0.75% with Rising Market Price Checkpointing. This result arises from the fact that the confidence level and bidding strategy both provide enough prevention of early-terminations that the extra execution time from checkpointing serves to only increase the cost.

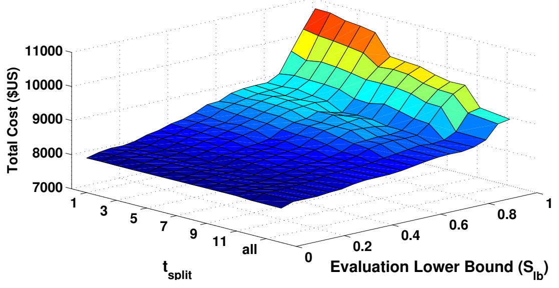
The fraction of deadline breaches are given in Figure C.3 for each checkpointing strategy and with different values of S_{lb} and t_{split}. Deadline breaches occur in a small fraction of the jobs, with results showing deadline breaches ranging from 0.65 % to 1.76% of all jobs using our artificial deadline of twice the estimated execution time, and early terminations ranging from 0.8% to 7.5%. Decreasing t_{split} and increasing S_{lb}



(a) Hourly Checkpointing



(b) Rising Market Price Checkpointing



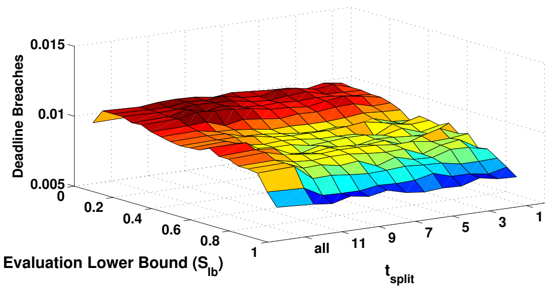
(c) No Checkpointing

Figure C.4: Total Costs (Rigid Jobs)

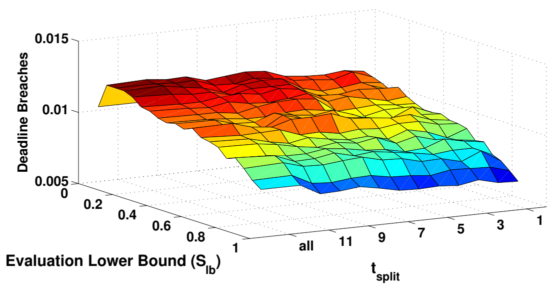
both tend to decrease the number of deadline breaches and early terminations for fixed S_{lb} and t_{split}, respectively. A closer analysis of early terminations and deadline breaches with $t_{split} = 1$ is given in Figure C.6a and with $S_{lb} = 0.95$ in Figure C.6b. In addition, a comparison of total costs for each checkpointing strategy when $t_{split} = 1$ hour, and when $S_{lb} = 0.95$ are shown in Figures C.7a and C.7b, with these values chosen as total costs, early terminations, and deadline breaches are subject to the greatest variation. When $t_{split} = 1$, the number of early terminations decreases from around 7% of all jobs to less than 1% as S_{lb} increases. When $S_{lb} = 0.95$, increasing t_{split} to encompass all execution times increases the number of terminations by less than one percent to between 1.4% and 1.65% for all checkpointing strategies.

C.2.1.2 Rigid Jobs

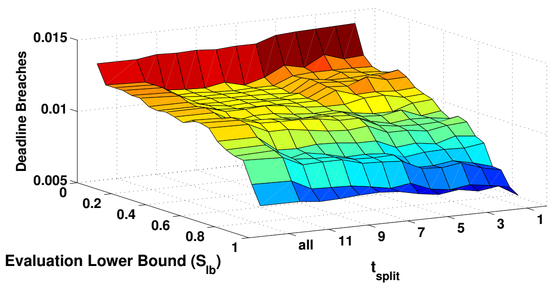
Total costs for the set of rigid jobs and for each checkpointing strategy are shown in Figure C.4. In comparison to moldable jobs, total costs are significantly higher, reflecting the fact that being able to search among a larger set of instance types allows for a broader set of possible trade-offs between cost and execution time, as well as increasing greatly the number of Spot instances with a potentially lower market price.



(a) Hourly Checkpointing



(b) Rising Market Price Checkpointing

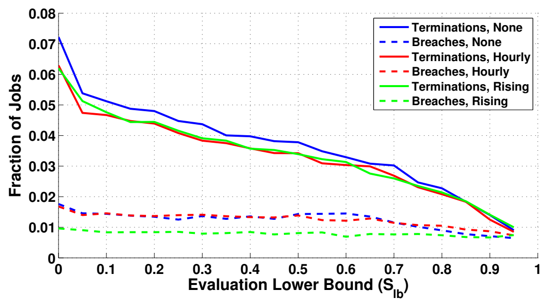


(c) No Checkpointing

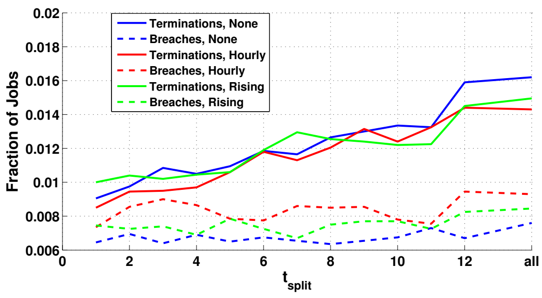
Figure C.5: Deadline Breaches (Rigid Jobs)

As with moldable jobs, total costs also tend to decrease as the confidence level is decreased and as t_{split} increases. Total costs when using only On-Demand instances were \$23,582.80, reflecting savings of between 54% and 68.2% when supplementing with Spot instances. For higher confidence levels, total costs were minimal when using Rising Market Price checkpointing with t_{split} encompassing all execution times and with $S_{lb} = 0$. For $t_{split} \geq 12$ hours, both Hourly and Rising Market Price checkpointing overtook None as lower cost strategies (see Figure C.7b, for example). However, in contrast to moldable jobs, rigid jobs saw Rising Market Price checkpointing achieve the lowest cost in most cases with an overall minimum of \$7,508.87 compared to \$8,008.79 with no checkpointing strategy, and \$7,723.44 with Hourly checkpointing, yielding savings of 6% and 2.8%, respectively. These results stem from the lowered execution time compared to Hourly checkpointing and the better fault tolerance compared to having no checkpointing strategy. The fraction of jobs that surpassed their deadlines tends to be lower for rigid jobs by up to 0.5% when compared to moldable jobs, as smaller instances (and thus longer execution times) are never utilized.

The fraction of deadline breaches are given in Figure C.5 for each checkpointing strategy and with different values of S_{lb} and t_{split}. Deadline breaches range from 0.5% to 1.5% and early terminations from 1.5% when $t_{split} = 1$ and $S_{lb} = 0.95$ to 15.3%



(a) Terminations and Deadline Breaches vs. S_{lb} ($t_{split} = 1$)



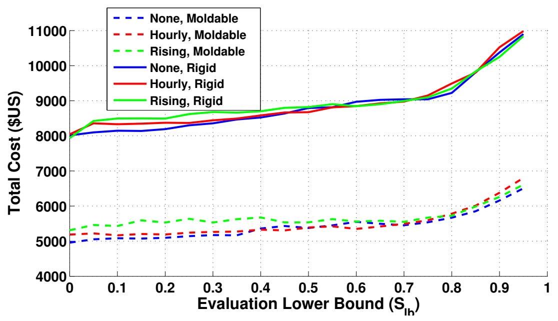
(b) Terminations and Deadline Breaches vs. t_{split} ($S_{lb} = 0.95$)

Figure C.6: A comparison of early terminations and deadline breaches for $t_{split} = 1$ or $S_{lb} = 0.95$ with moldable jobs

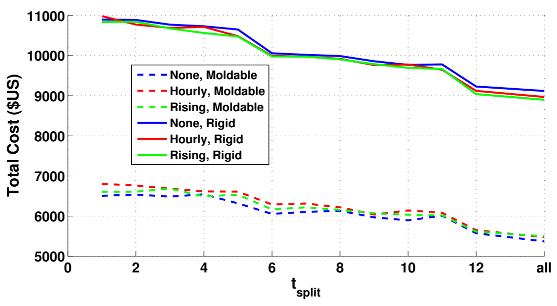
when t_{split} = all and $S_{lb} = 0$, with deadline breaches decreasing as S_{lb} increases and t_{split} decreases.

C.2.2 Discussion

Market Price estimation outperformed all others when S_{lb} was greater than 0.1, with a growing increase in accuracy over other estimates as S_{lb} increases. This increasing disparity between cost estimates reflects the fact that other cost estimate methods rely on the instance's potential bid. As S_{lb} is increased, the bid will also monotonically increase, allowing for a wider range of past market prices to be taken into account when calculating the average prices or the average costs. Since Spot prices currently exhibit periods of little fluctuation punctuated by large price spikes, using data from periods of different market prices in the estimation will be less indicative of the actual cost. For



(a) Total Cost vs. S_{lb} ($t_{split} = 1$)



(b) Total Cost vs. t_{split} ($S_{lb} = 0.95$)

Figure C.7: A comparison of total costs for $t_{split} = 1$ or $S_{lb} = 0.95$ with rigid and moldable jobs

lower values of S_{lb}, the range of bids which satisfy the confidence level is constricted (when $S_{lb} = 0$, the bid will always be equal to the market price) and thus cost estimation methods utilizing past Spot prices will have a more accurate estimate. As expected, cost estimates that do not take into account the current market price tended to perform more poorly than others in each group.

Placing bids as a function of the execution time of a job appears to be inferior, in terms of total cost, when compared to placing bids based on the past overall availability of the instance using that bid, regardless of the length of the job (for example, see Figure C.7b). When all bids are placed as availability bids, total costs decrease by up to nearly 20% compared to when $t_{split} = 1$ hour. This decrease in cost is due to both the recent stability of Spot instance market prices and the interchangeability of jobs and instances inherent in such a bidding strategy. This interchangeability alleviates the necessity of matching an execution time with a bid in order to help prevent early termination as opposed to finding an overall availability bid, which can be used for all jobs rather than primarily for the one the instance is originally intended for. Thus, backfilling jobs and running them on idle instances can be better utilized with such bids due to the fact that instances are matched with jobs regardless of the job's execution time or the instance's bid, therefore decreasing the number of times a new instance is leased while another can satisfy the deadline.

Regardless of checkpointing strategy or t_{split}, total costs are lowest when $S_{lb} = 0$. Such a confidence level implies all bids are equal to the market price of the instance, meaning that total costs are lowest when the bid is simply equal to the market price. However, coupled with this decreased cost is a higher number of early terminations and deadline breaches. For example, when $t_{split} = 1$ hour, setting S_{lb} equal to 0 results in an increase of nearly 6% in the percentage of jobs that are early terminated compared to setting S_{lb} equal to 0.95 (Figure C.6a). For high confidence levels, e.g. $S_{lb} \geq 0.9$, the total cost rises dramatically, especially with lower values of t_{split}. As shown in Figures C.7a and C.7b, when $t_{split} = 1$ hour and the jobs are rigid, an increase of around 37.5% from approximately \$8,000 to \$11,000 is observed compared to when $S_{lb} = 0$ when jobs are rigid, and around 30% from approximately \$5,000 to \$6,500 when jobs are moldable.

Deadline breaches generally occur infrequently, with the number of jobs exceeding their deadline never rising above 1.8%, regardless of parameters or checkpointing strategy. For both sets of jobs and all checkpointing strategies, increasing the confidence level and decreasing t_{split} lowers the number of early terminations and, consequently, the number of deadline breaches. The number of early terminations for rigid jobs are generally lowest for Rising Market Price checkpointing, and, as illustrated in Figure C.5, the percentage of deadline breaches using this strategy never rises more than 1%. This occurs because Rising Market price checkpointing provides sufficient fault tolerance in the face of higher early termination rates for rigid jobs, while having a smaller impact on the job's execution time compared with Hourly checkpointing. Indeed, for higher confidence values, Hourly checkpointing tends to increase the number of deadline breaches that occur, rather than decrease them. This stems from the fact that higher values of S_{lb} tend to provide enough prevention of early terminations that Hourly checkpointing merely adds to the execution time of the job. In contrast, Hourly checkpointing when using rigid jobs generally has the lowest number of deadline breaches due to the inability to extend the job's execution time on a smaller instance. Allowing moldability in the jobs also provides a means to decrease the number of early terminations, with around half as many early terminations compared to rigid jobs when t_{split} encompasses all jobs. This is due to the larger number of search options and the ability to lower a job's execution time on a larger instance.

C.3 Implementations

C++ implementations of several algorithms introduced in RAMC-DC are presented below.

Implementation C.1: Determine the *evaluation* score of a Spot instance based on job length: $S(j,v)$

```
/*Get the score of a Spot instance based on job length, etc. */
double spotScore(SpotInstance* inst, job* current_job, bool availability_bid, double↩
    cur_date){
  unsigned int inst_idx = inst->get_instance();
  unsigned int zone_idx = inst->get_avail();
  pair<int, int> end_start = window_search(dates[inst_idx][zone_idx], cur_date, ↩
      truncate_value+current_job->est_runtime);
  int end_idx = end_start.first;
  int start_idx = end_start.second;
  if(end_idx == -1 || start_idx == -1 || dates[inst_idx][zone_idx][end_idx] - dates[↩
      inst_idx][zone_idx][start_idx] <= 0){
    return -1.0;
  }
  double start = max(dates[inst_idx][zone_idx][end_idx] - truncate_value - ↩
      current_job->est_runtime, dates[inst_idx][zone_idx][start_idx]);
  double width = cur_date - start - current_job->est_runtime;
  if(width <=0){
    return -1.0;
  }
  if(availability_bid){
    return availability(inst->get_bid(), inst->get_avail(), inst->get_instance(),↩
        start_idx, end_idx);
  }else{
    vector<double> rand_dates = random_array1(NUM_DATES, dates[inst_idx][zone_idx][↩
        start_idx], dates[inst_idx][zone_idx][end_idx]);
    return reliability(inst->get_bid(), start_idx, end_idx, rand_dates, current_job↩
        ->est_runtime, inst->get_instance(), inst->get_avail());
  }
}
```

Implementation C.2: Determine the *availability* score of an unleased Spot instance: $availability(\langle i,z,b\rangle)$

```
/*Return the availability (instance available / total time) of this bid for a Spot ↩
    instance of type inst_idx in zone zone_idx.*/
double availability(double bid, unsigned int zone_idx, unsigned int inst_idx, ↩
    unsigned int start_idx, unsigned int end_idx){
  double avail = 0.0;
  for(unsigned int i = start_idx; i < end_idx; ++i){
    double mkt_price;
    mkt_price = prices[inst_idx][zone_idx][i];
    if(mkt_price > bid){
      continue;
    }
    avail += dates[inst_idx][zone_idx][i+1]-dates[inst_idx][zone_idx][i];
  }
  return avail/(dates[inst_idx][zone_idx][end_idx] - dates[inst_idx][zone_idx][↩
     start_idx]);
}
```

Bibliography

Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/, 2013. 1.4

Amazon. AWS Discussion Forums. https://forums.aws.amazon.com/thread.jspa?threadID=76964, 2012. 4.2.2

Amazon. Aws Management Console. http://aws.amazon.com/console/, 2013a. 1.4

Amazon. Amazon EC2 Reserved Instance Marketplace - Beta. http://aws.amazon.com/ec2/reserved-instances/marketplace/, 2013b. 2, 6

Amazon. Amazon EC2 Spot Instances. http://aws.amazon.com/ec2/spot-instances/, 2013c. 2.1.4, 2.1.4.1, 2.2

Amazon. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, 2013d. 1.4, 2.1.2, 2.1.6, 2.2, 5.3.4

Amazon Web Services. Amazon Web Services Blog. URL http://tinyurl.com/853xekg. 2.2.1

A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under SLA constraints. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 257–266. IEEE, 2010. 2.2.1, 2.2.2, B.1

M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010. 1.1, 1.1

O. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. In *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 304–311. IEEE, 2011. 2.1.4, 2.1.5.1, 2.2.1, 2.2.3, 2.2.4, 2, 4.2.1

BOINC. Catalog of boinc projects. `http://boinc-wiki.ath.cx/index.php?title=Catalog_of_BOINC_Powered_Projects.`, 2011. 2.2.2

M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2011. 2.2.2

R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2003.

R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1): 23–50, 2011. ISSN 1097-024X. doi: 10.1002/spe.995. URL `http://dx.doi.org/10.1002/spe.995`. 2.2.1, 2.2.3

J. Chen, C. Wang, B. Zhou, L. Sun, Y. Lee, and A. Zomaya. Tradeoffs between profit and customer satisfaction for service provisioning in the cloud. In *International Symposium on High Performance Distributed Computing (HPDC)*, pages 229–238. ACM, 2011. 1.3.1, 2.2.1, 2.2.4

N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See spot run: using spot instances for MapReduce workflows. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010. 2.2.3

W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. In *IEEE International Workshop on Workload Characterization (IISWC)*, pages 140–148. IEEE, 2001. 5.2.1

C. Cycles. Cyclecloud Achieves Ludicrous Speed! (Utility Supercomputing with 50,000-cores). `http://blog.cyclecomputing.com/2012/04/cyclecloud-50000-core-utility-supercomputing.html`, 2012. 2.1.4

A. Downey. A parallel workload model and its implications for processor allocation. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 112–123. IEEE, 1997. 5.2.1

C. Evangelinos and C. Hill. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazons EC2. *ratio*, 2(2.40):2–34, 2008. 2.2

B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. Bowers, and M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *ACM Symposium on Cloud Computing (SOCC)*, pages 20–28. ACM, 2012. 2.2.4, 1

D. Feitelson. Parallel workloads archive. `http://www.cs.huji.acil/labs/parallel/workload`. 4.5

D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. *Market-Based Control: A Paradigm for Distributed Resource Allocation*, pages 156–183, 1996.

High Scalability. Pinterest cut costs from $54 to $20 per hour by automatically shutting down systems. `http://tinyurl.com/azjjyn9`, 2012. 2.1.1, 2.1.5.1

International Telecommuniation Union. Fg cloud technical report (parts 1 to 7). 2012. 1.1

A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema. The grid workloads archive. *Future Generation Computing Systems*, 24(7):672–686, July 2008. ISSN 0167-739X. doi: 10.1016/j.future.2008.02.003. URL `http://dx.doi.org/10.1016/j.future.2008.02.003`. 2.2.1, 2.2.2

K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright. Performance analysis of high performance computing applications on the Amazon Web Services cloud. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 159–168. IEEE, 2010. 2.2

N. Jain, I. Menache, J. Naor, and J. Yaniv. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 255–266. ACM, 2012.

B. Javadi, R. Thulasiram, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. In *IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 219–228. IEEE, 2011. 2.2.1

Y. Lee, C. Wang, A. Zomaya, and B. Zhou. Profit-driven service request scheduling in clouds. In *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 15–24. IEEE, 2010. 2.2.4

L. Leslie, Y. C. Lee, P. Lu, and A. Zomaya. Exploiting performance and cost diversity in the cloud. *IEEE International Conference on Cloud Computing (CLOUD)*, 2013. To appear. 3

H. Liu. Cutting MapReduce cost with spot market. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 1–5, 2011. 2.1.4.2, 2.2.2, 2.2.3, B.1

H. Liu and D. Orban. Cloud MapReduce: a MapReduce implementation on top of a cloud operating system. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 464–474. IEEE, 2011. 2.2.3

M. Mao and M. Humphrey. A performance study on the VM startup time in the cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 423–430. IEEE, 2012. 2.1.5.2, 5.2.2

M. Mattess, C. Vecchiola, and R. Buyya. Managing peak loads by leasing cloud infrastructure services from a spot market. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 180–188. IEEE, 2010. 2.2.3

M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 296–303. IEEE, 2011. 2.2.3, 2, 4.2.1

P. Mell and T. Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800:145, 2011. 1.1

M. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon S3 for science grids: a viable solution? In *International Workshop on Data-Aware Distributed Computing (DADC)*, pages 55–64. ACM, 2008. 2.2

PlanForCloud. AWS Reserved Instances vs On-Demand: Breakeven point. http://blog.planforcloud.com/2013/02/aws-reserved-instances-vs-on-demand.html/, 2013. 2

F. Popovici and J. Wilkes. Profitable services in an uncertain world. In *IEEE/ACM Conference on Supercomputing (SC)*, page 36. IEEE, 2005. 2.2.4

M. Rahman. Risk aware resource allocation for clouds. 2011. Technical Report. 2.2.3

S. Shang, J. Jiang, Y. Wu, G. Yang, and W. Zheng. A knowledge-based continuous double auction model for cloud market. In *IEEE International Conference on Semantics Knowledge and Grid (SKG)*, pages 129–134. IEEE, 2010. 2.2.1

Y. Song, M. Zafer, and K. Lee. Optimal bidding in spot instance market. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 190–198. IEEE, 2012. 2.2.1

B. Sotomayor, K. Keahey, and I. Foster. Combining batch execution and leasing using virtual machines. In *International Symposium on High Performance Distributed Computing (HPDC)*, pages 87–96. ACM, 2008. 5.2.2

M. Taifi, J. Shi, and A. Khreishah. Spotmpi: a framework for auction-based HPC computing using amazon spot instances. *Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 109–120, 2011. 2.2.3

A. Toosi, R. Calheiros, R. Thulasiram, and R. Buyya. Resource provisioning policies to increase IaaS provider's profit in a federated cloud environment. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 279–287. IEEE, 2011. 2.2.4

K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis. Flexible use of cloud resources through profit maximization and price discrimination. In *IEEE International Conference on Data Engineering (ICDE)*, pages 75–86. IEEE, 2011. 2.2.4

University of Western Sydney. Spot Price Archive. http://spot.scem.uws.edu.au, 2013. 1.4, 4.5

W. Voorsluys and R. Buyya. Reliable provisioning of spot instances for compute-intensive applications. In *IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 542–549. IEEE, 2012. 1.3.1, 2.2.3, 5.2.1, 5.2.2, B.1

W. Voorsluys, S. Garg, and R. Buyya. Provisioning spot market cloud resources to create cost-effective virtual clusters. *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 395–408, 2011. 2.2.3, B.1

E. Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *Usenix ;Login*, 33(5):18–23, 2008. 2.2

D. Warneke and O. Kao. Resource pricing game in geo-distributed clouds. *IEEE Conference on Computer Communications (INFOCOM)*, 2013.

Wired. Amazon Cloud Powered by Almost 500,000 Servers, 2012. URL http://www.wired.com/wiredenterprise/2012/03/amazon-ec2/. 1

Xen. Xen.org, the home of the Xen project. http://www.xen.org/, 2013. 2.1

S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 236–243. IEEE, 2010. 2.2.2

S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on Amazon Cloud spot instances. In *IEEE Transactions on Services Computing (TSC)*, pages 236–243. IEEE, 2011. 2.2.2

L. Youseff, K. Seymour, H. You, J. Dongarra, and R. Wolski. The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software. In *ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 141–152. ACM, 2008. 2.2

M. Zafer, Y. Song, and K. Lee. Optimal bids for spot VMs in a cloud for deadline constrained jobs. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 75–82. IEEE, 2012. 2.2.1, 2

Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao. Dynamic resource allocation for spot markets in clouds. In *USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, pages 1–1. USENIX Association, 2011. 2.2.1, 2.2.3

H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang. Optimal resource rental planning for elastic applications in cloud market. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 808–819. IEEE, 2012. 1.3.1, 2.2.3