TRANSIENCY-DRIVEN RESOURCE MANAGEMENT FOR CLOUD COMPUTING PLATFORMS

A Dissertation Presented

by

PRATEEK SHARMA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2018

College of Information and Computer Sciences

TRANSIENCY-DRIVEN RESOURCE MANAGEMENT FOR CLOUD COMPUTING PLATFORMS

A Dissertation Presented

by

PRATEEK SHARMA

ABSTRACT

TRANSIENCY-DRIVEN RESOURCE MANAGEMENT FOR CLOUD COMPUTING PLATFORMS

SEPTEMBER 2018

PRATEEK SHARMA

B.S., BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

M.S., INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant Shenoy

Modern distributed server applications are hosted on enterprise or cloud data centers that provide computing, storage, and networking capabilities to these applications. These applications are built using the implicit assumption that the underlying servers will be stable and normally available, barring for occasional faults. In many emerging scenarios, however, data centers and clouds only provide transient, rather than continuous, availability of their servers. Transiency in modern distributed systems arises in many contexts, such as green data centers powered using renewable intermittent sources, and cloud platforms that provide lower-cost transient servers which can be unilaterally revoked by the cloud operator.

Transient computing resources are increasingly important, and existing fault-tolerance and resource management techniques are inadequate for transient servers because applications typically assume continuous resource availability. This thesis presents research in distributed systems design that treats transiency as a first-class design principle. I show that combining transiency-specific fault-tolerance mechanisms with resource management policies to suit application characteristics and requirements, can yield significant cost and performance benefits. These mechanisms and policies have been implemented and prototyped as part of software systems, which allow a wide range of applications, such as interactive services and distributed data processing, to be deployed on transient servers, and can reduce cloud computing costs by up to 90%.

This thesis makes contributions to four areas of computer systems research: transiency-specific fault-tolerance, resource allocation, abstractions, and resource reclamation. For reducing the impact of transient server revocations, I develop two fault-tolerance techniques that are tailored to transient server characteristics and application requirements. For interactive applications, I build a derivative cloud platform that masks revocations by transparently moving application-state between servers of different types. Similarly, for distributed data processing applications, I investigate the use of application level periodic checkpointing to reduce the performance impact of server revocations. For managing and reducing the risk of server revocations, I investigate the use of server portfolios that allow transient resource allocation to be tailored to application requirements.

Finally, I investigate how resource providers (such as cloud platforms) can provide transient resource availability without revocation, by looking into alternative resource reclamation techniques. I develop resource deflation, wherein a server's resources are fractionally reclaimed, allowing the application to continue execution albeit with fewer resources. Resource deflation generalizes revocation, and the deflation mechanisms and cluster-wide policies can yield both high cluster utilization and low application performance degradation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Many enterprises and software systems rely in large part on cloud computing platforms for their computing needs. Today's cloud platforms enable customers to rent computing resources and deploy applications on them in an *on demand* manner. This utility-computing model offers numerous benefits, including pay-as-you-go pricing, the ability to quickly scale capacity when necessary, and low costs, due to their high degree of statistical multiplexing and massive economies of scale.

To handle the growing number and diversity in applications, cloud platforms offer computing resources with a wide range of cost, availability, and performance characteristics. This thesis looks at one such type of computing resource, called *transient servers*. In contrast to traditional cloud servers whose availability can be assumed to be continuous, transient servers only offer intermittent and *transient* availability, and applications can have their access forcibly *revoked* by the resource provider.

Running modern distributed applications on transient servers raises a slew of new challenges. Most applications are designed and built with the implicit assumption that its computing resources will continue to be available until relinquished. Transient server revocations can cause loss of application-state, which can result in application downtimes, degraded performance due to failure-recovery, and end-user dissatisfaction in general. While transient servers introduce many challenges for applications, they are also *significantly* cheaper compared to their non-revocable counterparts. For example, transient servers offered by large public cloud providers such as Amazon EC2's spot servers can be upto 50-90% cheaper compared to the traditional, non-revocable, "on-demand" servers.

This thesis examines and addresses some of the challenges of running applications on cloud transient servers. These challenges are addressed by designing and building systems that introduce new mechanisms, policies, and abstractions—that together enable more effective use of transient servers for a wide range of applications.

The reminder of this chapter provides a brief overview of transient servers and their characteristics, the resulting systems research challenges, and provides a summary of the research contributions made in this thesis.

1.1 Motivation

Due to the rising popularity of cloud computing, the number of data centers, and their size, continues to grow at a rapid pace. Importantly, the distributed applications that run in data centers are generally built with the implicit assumption that the common case is for servers to be available—they may fail, but failures are uncommon, and when they happen, well-known techniques for fault-tolerance mitigate their performance impact.

However, many emerging scenarios are now altering this long-standing basic assumption. Rather than attempt to ensure continuous server availability and then design systems to mask rare failures, these scenarios instead offer transient availability, such that servers are available only temporarily for an uncertain amount of time. These *transient servers* are often cheaper and more energy-efficient than stable servers, which provide continuous availability.

Transient servers arise in many data center and cloud environments. In data centers running on renewable energy (such as solar and wind), deactivation and reactivation of servers is required for handling intermittent power supply. In public Infrastructure-as-a-Service clouds, cloud operators offer their idle, surplus computing resources as low-cost servers that can be preempted can revoked when demand for higher-priced on-demand servers increases.

Handling transient server availability raises new challenges in systems design. While transient server unavailability can be masked by treating server unavailability as fail-stop failures and using fault-tolerance techniques, we argue that these traditional techniques are inappropriate and insufficient. Since transiency arises from a desire to cut costs by relaxing the requirement for near-continuous server availability, employing expensive fault-tolerance techniques (such primary-secondary replication) would eliminate its benefits. Thus, optimization techniques for transient servers must be lightweight to maintain the low cost of transient servers.

Importantly, transient resource availability is fundamentally different from classic fault-tolerance in three important aspects:

High unavailability: Transient servers can become unavailable at a much higher rate than hardware failures in conventional servers, since their availability is controlled by the operational policies of the data center or the cloud.

Advance warning: Transient server unavailability is different from sudden fail-stop failures in that the unavailability is a result of higher-level operator policies. Thus unlike, say, hardware failures that occur without warning, transient server revocations are often preceded by a revocation-warning signal.

Heterogeneity: Transient servers trade off availability for lower cost. In many scenarios, *multiple types* of transient servers may be offered with different availability-cost tradeoffs. Thus in addition to fault-tolerance, applications can also mitigate the effects of revocations by carefully selecting transient server types based on their sensitivity to cost and revocations.

Thus there is an opportunity to leverage transient servers' unique mix of characteristics—high failure rates, failure warnings, and server selection—to design low-cost techniques that mitigate impact on performance and availability. For example, Amazon's EC2 spot instances cost only 0.1–0.5$\times$ on-demand (non-revocable) servers, and by mitigating the effects of the spot instance revocations, we can achieve significant cost savings.

As transient resources become increasingly prevalent and ubiquitous, we argue that transiency-aware resource management is crucial for increasing the utility of the transient resources as well as reduce computing costs. The effective utilization of transient resources raises many interesting questions and challenges:

- How to mask transient server revocations such that applications requiring nearly-continuous availability can still use them?
- How can fault-tolerance techniques be adapted to transient servers to mitigate the performance overheads of revocations?
- How to manage and allocate transient resources for applications that have different cost and availability requirements?
- How to reclaim transient resources without resorting to preemption?

The questions above cover some of the challenges in deploying applications on transient servers. By tackling these problems, we can provide low-cost computing to applications and make transient resources as "first-class" computing resources.

1.2 Thesis Contributions

Most applications are designed and built with the implicit assumption that its computing resources will continue to be available until relinquished. This assumption is not compatible with transient servers that only offer intermittent resource availability. Transient server revocations can cause loss of application-state, which can result in application downtimes, degraded performance due to failure-recovery, and end-user dissatisfaction in general.

This thesis develops novel techniques that combine fault-tolerance techniques with transiency-specific resource management, that enable a wide range of applications to make effective use of transient resources.

This thesis makes contributions in these four areas of computer systems research:

1. Transiency-specific fault tolerance techniques
2. Transient resource allocation
3. Abstractions for cloud resources
4. Transient resource reclamation

One of the main themes of this thesis is that combining fault-tolerance mechanisms with transient resource management policies, to suit application characteristics and requirements, has significant cost and performance benefits. These mechanisms and policies developed as part of this thesis have been implemented and prototyped as part of four systems, that are summarized below:

SpotCheck: Provides bounded-time virtual machine live migration as part of a *derivative cloud platform*, which can run unmodified interactive applications on low-cost cloud transient servers.

Flint: Runs batch-interactive distributed data processing applications on cloud transient servers. Flint reduces the impact of revocations through periodic application-level checkpointing, and selects transient servers to minimize cost and running time.

ExoSphere: A cluster manager for transient servers that runs multiple applications with different cost and availability preferences, by using the notion of *server portfolios*.

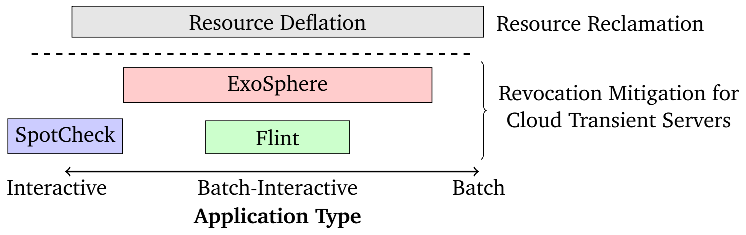


Figure 1.1: This thesis develops systems for running a wide range of applications on cloud transient servers.

Resource Deflation: A virtualized cluster management framework that uses resource deflation—a fractional resource reclamation technique, that allows data center operators to achieve high utilization without necessitating preemption.

Together, these systems allow a range of applications to run on transient servers, as shown in Figure 1.1.

1.2.1 Transiency-specific Fault-tolerance Techniques

Transient server revocations can affect the availability and performance of applications that run on them. Transiency-specific fault-tolerance is one of the revocation-mitigation approaches that this thesis explores. We develop two new fault-tolerance techniques that leverage the unique characteristics of transient server revocations.

Bounded-time Live Migration: To allow applications requiring continuous availability to run on transient servers, we use nested virtualization and bounded-time live migration to move application state upon revocation to stable, non-revocable servers. This technique leverages the small pre-revocation warning, allowing us to safely migrate and transfer an application's in-memory state to a fall-back non-revocable cloud server. This system-level technique allows unmodified applications, including interactive services, to run on low-cost cloud transient servers with minimal downtime (Chapter 3). This greatly expands the potential uses of transient servers, whose use has conventionally been restricted to disruption-tolerant batch jobs.

Periodic Application-level Checkpointing: The second fault-tolerance technique we develop uses application-level checkpointing for distributed data processing applications. Server revocations can severely impact the completion times of data processing jobs, since the revocations often require expensive recomputation of lost data. Furthermore, low-latency data processing is an increasingly common requirement, and is even less tolerant to revocations. We develop an application-level automated checkpointing technique that adapts checkpointing frequency to server availability characteristics. This allows both batch and interactive data processing workloads to run on low-cost transient servers with minimal performance overheads (Chapter 4).

1.2.2 Transient Resource Allocation

In addition to fault-tolerance, revocations can also be mitigated if the application is deployed on transient servers in a manner that minimizes the number and frequency of revocations.

Server Selection: Transient server types offer different cost and availability tradeoffs. Moreover, applications also have different requirements for cost, availability, and performance. Existing applications usually assume that computing resources have a fixed price and availability—which doesn't hold in the case of transient servers that have different cost and availability tradeoffs. Thus selecting the "right" servers for an applications requires jointly optimizing the cost, availability, and application requirements. This thesis develops multiple *server selection* policies for a range of applications. These server selection policies have been implemented as part of the systems we have developed (SpotCheck, Flint, and ExoSphere).

Specifically, the server selection policies developed as part of this thesis focus on *heterogeneous* server selection, where a distributed application can be deployed on a collection of servers of different types (and hence different costs and availabilities). This thesis develops many heterogeneous server selection policies—from simple and application-specific (Chapters 3, 4) to more general-purpose solutions that can work for a wide array of applications. For general-purpose heterogeneous server selection, we use techniques that are inspired by portfolio construction in financial economics. This allows the allocation of transient servers to applications with different resource requirements. Server portfolios enable construction of an "optimal" mix of severs to meet an application's sensitivity to cost and revocation risk. Portfolios enable and exploit diversification, and can reduce revocation risk. Such a portfolio-based transient server selection policy is implemented as part of the ExoSphere system (Chapter 5).

1.2.3 Abstractions For Cloud Resources

One of the goals of this thesis is to enable the use of transient servers by a wide range of applications. To this end, we develop abstractions that enable applications to make use of transient cloud resources in an effective and seamless manner.

Derivative Clouds: We develop the notion of *derivative cloud platforms*, which repackage and resell different server types. Analogous to a financial derivative, a derivative cloud can offer resources to customers with different pricing models and availability guarantees not provided by the native cloud platform. Derivative clouds enable third parties to develop their own cloud platform on top of public Infrastructure-as-a-Service clouds, by running their own virtualization layer on the cloud servers. We develop SpotCheck as an example derivative cloud, which can be used to reduce revocation risk by combining revocable and non-revocable servers, and transparently moving application-state between native cloud servers in the case of revocations.

Server Portfolios: We develop an abstraction based on the portfolio-driven server selection called *server portfolios.* Server portfolios allow applications to select from a large range of heterogeneous server combinations based on the application cost and availability preferences. Server portfolios, when combined with a fault-tolerance

API developed as part of the ExoSphere system (Chapter 5), allow a wide range of applications be made transiency-aware with minimal effort.

1.2.4 Transient Resource Reclamation

While the fault-tolerance mechanisms, server selection policies, and abstractions developed as part of this thesis enable a wide range of applications to make effective use of transient servers, they still have to contend with the ill-effects and risks of revocations. This thesis also looks at an orthogonal approach to mitigating transient server revocations—through the lens of resource *reclamation*.

Cloud and data center operators reclaim transient resources by revoking/preempting them, and allocating them to higher paying/priority applications. We develop an alternative resource reclamation mechanism, called *resource deflation*, that provides continuous availability to applications, but at reduced average performance. This mechanism allows cloud and data center operators to fractionally reclaim resources from their low-priority applications during times of resource pressure.

Resource deflation attempts to shrink the resources allocated to virtual servers dynamically based on the resource pressure. Instead of revoking servers, we reduce their resources instead. These deflatable servers are analogous to conventional transient servers in that they provide only transient resource availability. Deflation generalizes transient server revocations, and offers a continuum of reclamation options.

Resource deflation ameliorates the state-loss effects of revocations, and allows unmodified applications to continue execution even during periods of low resource availability. We develop mechanisms for low-overhead deflation, and policies for managing deflatable servers at a cluster-wide level (Chapter 6).

1.3 Thesis Outline

The reminder of this thesis is structured as follows. Chapter 2 provides background on transient servers, and discusses related work on addressing the challenges of running applications on transient servers. Chapter 3 describes a derivative cloud platform called SpotCheck, and focuses on the challenges in transparently masking and reducing the risk of revocations of transient cloud servers. Chapter 4 describes application-level fault-tolerance techniques for running distributed data-processing applications on transient cloud servers. Chapter 5 presents a technique for transiency-aware resource management, called server portfolios, that generalizes and improves upon resource management policies described in Chapters 3 and 4. Chapter 6 develops and presents resource reclamation using resource deflation as an alternative to server revocation. Finally Chapter 7 summarizes the work done in this thesis, places it in a broader context, and presents some directions for future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter provides background on transient servers in clouds and data centers. We also discuss related work on resource management and applications for transient servers.

2.1 Cloud Computing

Cloud computing platforms are a popular choice for hosting a wide range of applications: such as latency-sensitive web services, large-scale distributed data analytics, machine learning, scientific computing workloads, etc. Cloud platforms provide computing as a utility—allowing users to rent data center resources for various timescales, and pay as they go for the resources consumed. This "on-demand" availability of computing resources has enabled the emergence of new applications and paradigms, such as large scale machine learning applications that power advanced artificial intelligence applications for autonomous cars, personalized assistants, image recognition, etc. The computing infrastructure required for these applications is provided by public cloud platforms, allowing individuals and organizations to rapidly develop, train, and refine their machine learning models without the need for investing in large-scale computing infrastructure.

Public cloud platforms such as Amazon AWS [3], Google Cloud [4], Microsoft Azure [8], IBM Cloud [5], Joyent [6], Alibaba Aliyun [1], etc., offer computing and storage resources to a large number of applications. To handle the increase in the scale and diversity in cloud-based applications, cloud platforms offer resources with different abstractions and characteristics. Cloud platforms can manage different layers of the software stack, and offer one or more of infrastructure, software platforms, software, or functions, *as a service*. Infrastructure as a service (IaaS) clouds offer computing resources in the form of virtual machines (VMs) that are managed by the cloud operator, and on which customers can deploy their own applications. Infrastructure cloud platforms offer cloud VMs with a variety of different hardware configurations ("sizes"), operating systems, geographical locations, and with different Service Level Agreements (SLAs) that govern the price, availability, and performance of the VMs. Large public cloud platforms such as Amazon AWS have multiple choices in each of the above dimensions—offering servers with over 50 different hardware configurations, with a choice of more than four operating systems, across more than ten geographic locations, and with multiple SLA's such as on-demand, spot, and burstable instances [2]. Deploying an application on a cloud platform entails selecting a cloud resource with the appropriate characteristics, based on the application resource requirements.

Large public cloud platforms have millions of servers, host millions of applications [98], and are now central components in the computing infrastructure. Their efficient operation is predicated on addressing many computer systems challenges, such as: resource management and allocation, geographical and temporal workload elasticity, capacity planning and provisioning, security, monitoring, programming models, networking, data management and storage, etc. Many of these areas continue to receive significant attention from both academia and industry.

However, the immense scale of cloud platforms has given rise to new challenges in resource management. This thesis focuses on one such emerging facet: transiency, and in particular, transiently available servers. Transiency is an emerging trend in cloud computing platforms that requires rethinking many assumptions in the design and deployment of applications and software systems. Transiency arises in many contexts, but has only received limited attention. We look at the issues surrounding transiency in cloud computing platforms—why it occurs, how it affects applications, and how its effects can be mitigated.

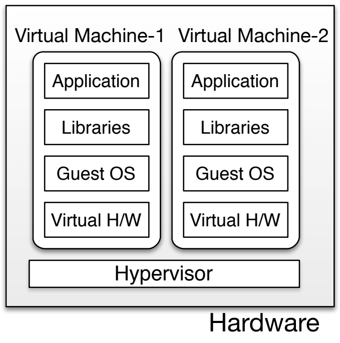
2.2 Data Center Resource Management

The computing, networking, and storage resources used by cloud applications are provided by large data centers, comprising of tens of thousands of servers, storage and networking devices, and the necessary power and cooling infrastructure. The features and flexibility of cloud computing such as pay-as-you-go pricing, elastic scaling of resources, and low-cost, are made possible through careful allocation and management of resources in the data centers. In this section, we shall look at how the challenges of providing flexibility and low-cost to cloud applications translate to challenges in data center resource management, and how these resource management challenges are addressed by data center operators.

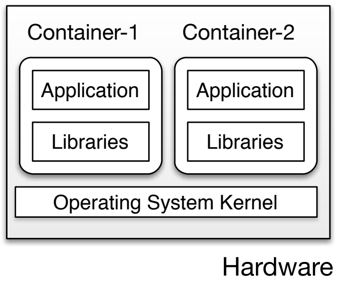
2.2.1 Virtualization

Data centers host multiple applications, and to decouple applications from the hardware they run on, data centers use virtualization to multiplex and manage resources. Virtualization provides a number of benefits. It enables a flexible allocation of physical resources to virtualized applications where the mapping of virtual to physical resources as well as the amount of resources to each application can be varied dynamically to adjust to changing application workloads. Furthermore, virtualization enables multi-tenancy, which allows multiple instances of virtualized applications ("tenants") to share a physical server. Multi-tenancy allows data centers to consolidate and pack applications into a smaller set of servers and reduce operating costs. Virtualization also simplifies replication and scaling of applications.

There are two types of server virtualization technologies that are common in data center environments—hardware-level virtualization and operating system level



(a) Virtual Machines



(b) Containers

Figure 2.1: VM and Container architectures

virtualization (Figure 2.1). Hardware level virtualization involves running a hypervisor which virtualizes the server's resources across multiple virtual machines. Each hardware virtual machine (VM) runs its own operating system and applications. By contrast, operating system virtualization virtualizes resources at the OS level. OS-level virtualization encapsulates standard OS processes and their dependencies to create "containers", which are collectively managed by the underlying OS kernel. Examples of hardware virtualization include Xen [55], KVM [126], and VMware ESX [37]. Operating system virtualization is used by Linux containers (LXC [26]), Ubuntu LXD [33], Docker [19], BSD Jails [122], Solaris Zones [58] and Windows Containers [39].

Both types of virtualization technologies also have management frameworks that enable VMs and applications to be deployed and managed at data center scale. Examples of VM management frameworks include commercial offerings like vCenter [38] and open source frameworks like OpenStack [27], CloudStack [29]. Kubernetes [32] and Docker Swarm [30] are recent container management frameworks. Hardware and operating system virtualization have different performance, isolation, security, application deployment, software development, and cluster management characteristics, which are compared in [175].

Virtualization is vital in the context of cloud resource management. Hardware and OS virtual machines serve as units of resource allocation and management, especially in the case of Infrastructure-as-a-Service cloud platforms. Cloud platforms can instantiate virtual machines with different resource allocations (“sizes”) on their server clusters, and control them dynamically via the cluster management software. Virtualization offers easier dynamic resource allocation using overcommitment [204] and migration [217], and fault-tolerance capabilities such as checkpointing [9], migration [77], and replication [81].

Problems in cloud resource management and transient availability often translate to virtual machines. For instance, transient servers usually refer to transient *virtual machines*, since virtual machines provide the server abstraction to applications. Thus, many of the systems challenges in transiency, such as fault-tolerance, can be addressed in the context of virtual machines—Chapter 3 looks at one such technique for low-overhead continuous checkpointing for virtual machines in the cloud.

2.2.2 Cost and Energy Efficiency

To provide low-cost computing to applications, data centers must be cost efficient. Data centers are expensive to set up and to run, and improving their efficiency is challenging.

Computing equipment (such as servers) is energy hungry, with each server consuming 100s of watts. Thus, the power consumption of data centers, which house several thousands of servers, is several megawatts. Thus due to their large energy footprint, improving energy efficiency is one of the primary means to improve data center cost efficiency.

Not all energy consumed by a data center goes into powering its IT equipment (servers, networking, and storage). As much as 50% of the energy is consumed for cooling the IT equipment, power transmission, and other overhead [177]. The metric for data center energy efficiency is Power Usage Effectiveness (PUE), which is defined as:

$$\text{PUE} = \frac{\text{Total Power Consumption}}{\text{IT Power Consumption}} \tag{2.1}$$

Modern data centers use a plethora of design and optimization techniques to achieve PUE's close to 1 [177]. Improvements in cooling, power supply, and server designs have resulted in large data centers achieving PUE's as low as 1.1, indicating that only 10% of the energy is consumed by non-IT components.

While improvements in energy efficiency lead to a corresponding reduction in the operating costs of data centers, the low PUE's indicate that any further improvements in data center design will mostly result in diminishing returns. Another important component of a data center's cost is the cost of building and setting them up (also known as the capital expenditure, or CAPEX). A large part of the data center's expenses are the capital expenses required to purchase the IT equipment, which is amortized over a relatively short time period of 3-5 years, due to short lifespans of most computing equipment. With improving energy (and hence operational) efficiency, the amortized capital expenses contribute to a large portion of the total cost of ownership of a data center. The breakdown of the total cost of ownership (amortized capital expenses + operating expenses) of a data center, shows that computing equipment accounts for more than 50% of the total expenses [105, 129], and with improving energy efficiency, this fraction is only expected to grow.

To reduce the overall costs and recoup their large capital expenditure, data center operators thus seek to maximize the utilization of their servers, since idle servers are indicative of sunk costs. However, achieving high server and data center utilization is fraught with many challenges.

The utility-computing model of cloud computing incentivizes users to only instantiate cloud resources (such as cloud VMs) when needed. This leads to a high degree of dynamism in the workloads and hence the data center utilization. In order to provide computing "on demand", data centers must be able to handle load spikes, time-of-day variations, etc. To do so, data centers are usually significantly overprovisioned, and the average data center utilization is low and in the range of 20–40% [201, 139]. Thus a significant portion of data center resources are idle, threatening the low-cost application deployment that the cloud promises.

Cloud data center operators have tried to address the challenge of low utilization by incentivizing the use of data center resources during periods of low demand, by offering low-cost transiently available cloud servers to applications. The next section discusses transiency in cloud data centers and in other environments.

2.3 Transiency in Modern Distributed Systems

The notion of transient resource availability arises in many computing environments, which we summarize below:

Cloud Platforms. Infrastructure-as-a-Service cloud providers have started offering their surplus resources as low-cost transient servers. Cloud operators can unilaterally revoke these transient servers during periods of high demand. This thesis is largely focused on transiency in cloud computing platforms, and we provide a more detailed background of transient cloud servers in Section 2.4.

Green Data Centers. Data centers are being increasingly powered by renewable "green" energy sources [12, 163]. Renewable sources, such as solar and wind, generate power intermittently, such that if a data center relies entirely on such sources, there may be periods of energy shortfalls that result in transient availability of a portion of the servers [191, 192].

Data Centers Connected To Smart Grids. The interactions between electric grids and data centers also results in transiency. Data centers can participate in demand-response schemes where the smart grid signals the data center to curtail power usage during peak periods of energy shortfalls. Real-time pricing of electricity by the smart grid, where the price of electricity fluctuates based on demand, also encourages data centers to voluntarily curtail or regulate their power usage during peak price periods to reduce their monthly electricity bills. Both scenarios will result in the transient availability of a portion of the servers in a data center.

Data Center Scheduling. Enterprise data centers host applications with different roles and priorities, and cluster management software can prioritize access to data center resources. For example, high-priority user-facing interactive services may preempt lower-priority batch jobs. In case of resource contention between different applications, a common strategy is to preempt the lower-priority application and offer its resources to one with a higher priority [32, 201]. Opportunistically leveraging idle resources within a data center can also result in transient availability for applications running on erstwhile idle resources [237]. Thus, transiency arises in data centers as a result of scheduling policies that are in place to maximize the utilization of data center resources.

2.4 Transient Cloud Servers

Many enterprises, especially technology startup companies, rely in large part on Infrastructure-as-a-Service (IaaS) cloud platforms for their computing infrastructure [52]. Today's IaaS cloud platforms, which enable their customers to rent computing resources on demand in the form of virtual machines (VMs), offer numerous benefits, including a pay-as-you-use pricing model, the ability to quickly scale capacity when necessary, and low costs due to their high degree of statistical multiplexing and massive economies of scale.

To meet the needs of a diverse set of customers, IaaS platforms rent VM servers under a variety of contract terms that differ in their cost and availability guarantees. Traditional cloud servers are leased on an on-demand basis—cloud customers may request them when needed and the cloud platform provisions these servers until the customer relinquishes them. Since customer demand for cloud resources can be highly dynamic, the cloud platform needs to over-provision the aggregate server capacity to handle peak demand. Consequently, a significant portion of the cloud server capacity tends to be idle during non-peak periods.

Cloud providers have begun to lease this surplus capacity at highly discounted prices to cost-sensitive customers. Doing so enables providers to earn revenue from otherwise idle resources. These surplus servers have transient availability since the cloud provider can reclaim them from the customer at any time, e.g., when demand for standard on-demand servers begins to rise.

Commercial platforms such as Amazon EC2 and Google Cloud Platform now support transient servers—Amazon EC2 started offering spot instances [17] since 2009; Google's Cloud Platform has been offering Preemptible VMs [24] since 2015; and Microsoft Azure has been offering low-priority preemptible Batch VMs since 2017 [40]. Cloud providers typically provide a brief advance warning prior to preempting/revoking a transient server to enable the customer to gracefully shutdown the machine. The warning time currently ranges from two minutes in Amazon EC2 cloud to 30 seconds on Google's cloud platform.

Even though transient servers in current cloud platforms arise out of the need to utilize surplus resources, transiency can also be a result of other allocation models. Resource-as-a-service [44] clouds provide dynamically priced computing resources, and charge by usage instead of allocation. In such scenarios, applications have to explicitly bid for resources, and being "out-bid" results in loss of resources. Thus in highly oversubscribed clouds, application resources can always be revoked by higher paying applications, and potentially *every* application may have to run on transiently available resources.

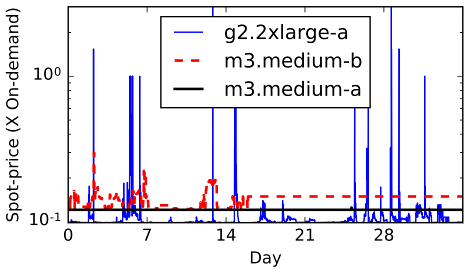


Figure 2.2: EC2 spot prices for three servers over May 2015.

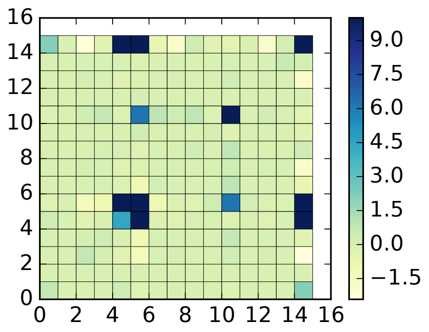


Figure 2.3: Heatmap showing the covariances of the spot prices of different EC2 spot markets. Most markets are weakly correlated (covariance close to 0).

2.4.1 Transient Server Pricing

Different cloud providers have employed different approaches for pricing transient servers. Google's transient servers, called preemptible instances [24], offer a fixed 80% discount but also have a maximum lifetime of 24 hours (with the possibility of earlier preemption). In contrast, Amazon's transient servers (which are called spot instances [17]) offer a variable discount—the price of spot instances varies continuously based on market supply and demand for each server type (Figure 2.2). Spot instances are typically 0.1–0.5× the cost of non-revocable on-demand instances.

Since transient servers are surplus idle machines, the resources available in the transient server pool fluctuate continuously depending on the supply and demand of on-demand servers. Thus, whether a certain transient server is available depends on current market conditions. A combination of server-type (such as large/small), geographical region, and availability zone (data center failure domains within a region), define a separate *market* of transient servers. The price and/or availability characteristics of individual markets can differ, as seen in Figure 2.2, which shows EC2 spot prices. In this example, the `m3.medium` in availability zone `a` has the most stable prices, `g2.2xlarge` in the same availability zone has a lower average price but

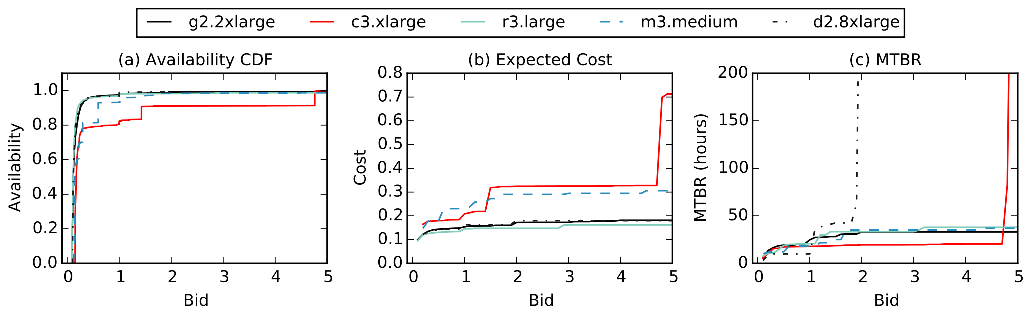


Figure 2.4: The effect of bidding on availability, expected cost, and MTBR for selected instance types. Bids and the expected costs are normalized to a factor of the corresponding on-demand price.

high variance, and the `m3.medium` in availability zone `b` has higher price than in zone `a`. The `g2.2xlarge` price spikes are not correlated with the other two servers. The example shows that larger servers may occasionally be more heavily discounted than smaller servers, and that identical servers in two availability zones may also be priced differently. The supply and demand of different server types across different regions may not always be correlated, and this is reflected in the general lack of correlation in their spot prices (Figure 2.3).

Bidding for EC2 spot instances. Amazon EC2 spot prices are determined by continuous sealed-bid second-price auction. Users place a single, fixed bid, which represents the maximum hourly price that they are willing to pay. The market price is based on all the bids and the available supply. Importantly, all users pay the same market price, which may be lower than the bid. The price of a spot instances in EC2 thus fluctuates continuously in real-time based on market demand and supply [61]. If the spot price rises above a user's maximum bid price due to increased market demand, EC2 revokes the spot server from the user after providing a two minute warning (and presumably allocates it to a higher paying user).

Bidding strategies optimize the cost-availability tradeoff for spot instances: as a user increases their bid, they may pay more per-hour, but their availability also increases. Since EC2 introduced its spot market, there has been significant research both on analyzing and modeling spot prices and developing bidding strategies based on real data and models. One of the first papers analyzing spot price data raised questions about whether EC2's mechanisms for setting the spot price were market driven [62]. However, as the authors note later, the characteristics of the spot price changed, making it consistent with a market driven allocation [62]. A number of related papers also analyze spot price data to better understand its statistical characteristics [119, 149, 198, 210, 219, 234, 227, 240, 194]. Analyzing and modeling spot price data is a prerequisite to developing bidding strategies that select the optimal bid to ensure a target level of performance at the minimum cost.

However, our analysis of spot price traces of over 1500 spot markets from March to August 2015, and of some markets from 2011–2015, shows that bidding strategies have minimal effect on the cost, availability, and revocation rate for most markets [178]. This is because spot prices are "spiky", and the resulting availability, cost, and MTBR (Mean Time Between Revocations) CDFs are long-tailed (Figure 2.4). Thus a very wide range of bids result in approximately the same price and availability characteristics, and careful bid selection may not be necessary.

Appendix A provides additional discussion on transient server pricing and the role of bidding for EC2's spot instances.

2.5 Overview of Related Work

While the effective use of transient servers presents many challenges, their cost and efficiency benefits have resulted in a burgeoning interest. This section looks at the related work on addressing the systems challenges of transiency for different classes of applications.

The low-cost cloud transient servers (such as EC2 spot instances) makes them ideal for running delay-tolerant batch jobs requiring large amounts of computing resources. Early research on cloud transient servers primarily targeted fault-tolerance techniques

Application	Related Work
Batch	[197, 203, 125, 225]
Hadoop	[136, 226, 75, 124, 130]
Spark	[176, 221]
Distributed databases	[65, 168]
Machine learning	[108]
Web services	[110, 212, 91, 167, 106], [182]
MPI	[142]
Key-value stores	[208, 220]

Table 2.1: A significant amount of work has gone into deploying different applications on transient servers.

for batch jobs [75, 136, 203], with the focus being on restarting or checkpointing jobs. A significant volume of prior work addresses cost-effective use of Amazon EC2 spot instances that expose a price vs. availability tradeoff due to the bidding mechanism. Since servers in the spot market are significantly cheaper than the equivalent on-demand servers, they are attractive for running delay-tolerant batch jobs [197, 118, 226, 212, 136, 75, 91]. Checkpointing is a common fault-tolerance mechanism for mitigating the impact of revocations on batch jobs in the spot market [203, 125, 225].

While early work on cloud transient servers limited their use to simple batch jobs and stateless web services, there has been a growing interest to make cloud transient servers applicable to a wider range of applications. Table 2.1 provides an overview of related work to address transiency challenges for different classes of applications.

The challenges of running stateful interactive services are addressed through migration-based fault-tolerance techniques in [110, 182]. Optimizing MPI jobs for spot servers is presented in [142]. Checkpointing and task-scheduling policies for distributed in-memory data processing applications like Spark [230] are described in [221, 176]. Prior work has also explored running a distributed database on spot instances [65, 168]. More recently, the use of cloud transient servers for large scale distributed data processing and machine learning has been examined in [223, 107].

Fault-tolerance	Checkpointing	[203, 125, 225, 142, 221, 65, 168], **[176]**
	Migration	[110, 186, 184], **[182]**
	Replication	[203, 106]
Server selection	Homogenous	[197, 91, 73, 226]
	Hetereogenous	[142], **[183]**, **[176]**, **[180]**
Spot pricing	Price analysis	[185], [215], [240], [207], **[179]**
	Price-reactive	[111, 106, 184]

Table 2.2: Related work on transiency mitigation incorporates many different approaches. Work part of this thesis is in bold.

Running a wider gamut of applications has required the development of application and transiency specific fault-tolerance and resource management techniques. Prior work has primarily explored three classes of transiency-mitigation techniques: fault-tolerance, server selection, and bidding/spot pricing. Table 2.2 presents an overview of the different transiency-mitigation techniques, with related work categorized by their main contribution.

The choice of the appropriate fault-tolerance technique (such as checkpointing, migration, or replication) depends on the application and server availability characteristics, and [197] examines the problem of selecting the appropriate fault-tolerance technique to minimize overall cost. The small revocation warning time makes migration challenging, and checkpointing has been adapted for many different classes of applications. Replicating applications on to different transient servers can also mitigate revocations, but at increased cost, and is thus only applicable in a narrower set of scenarios and applications.

Transient resource management often involves the problem of "selecting" the right transient servers based on application requirements of cost, availability, and performance. Prior work has looked at strategies for selecting a single cloud server [91, 197, 75, 119, 198]. In this thesis we also consider selection of a heterogeneous mix of transient cloud servers, a problem that has received relatively little attention.

Transiency In Enterprise Data Centers. While this thesis focuses on transient servers in the context of cloud platforms, transiency also arises in enterprise data centers looking to increase utilization. Recent work has sought to identify some of the benefits and challenges of transient availability in the context of enterprise data centers [237].

The internal dynamics of transiency in enterprise data centers are similar to the external dynamics of transient cloud servers, as they also arise from opportunistically leveraging idle resources. The different environments results in slightly different challenges, such as data storage [153]. The key difference is that internal supply/demand dynamics are generally well-known by the data center, while the external supply/demand dynamics of transient cloud servers are only indirectly conveyed through price signals, if at all. Data centers can also use short lived transient servers (few minutes instead of few hours) for distributed applications [224].

CHAPTER 3

RUNNING INTERACTIVE APPLICATIONS ON TRANSIENT SERVERS

While transient servers present an opportunity for low-cost computing, their revocable nature presents a hindrance for many applications, especially interactive services that require continuous availability. This chapter presents fault-tolerance techniques for allowing unmodified interactive applications to run virtually uninterrupted on revocable transient cloud servers. We also present the *derivative cloud* abstraction for transparently managing risks associated with transient servers. Our fault-tolerance techniques and policies are implemented in a derivative cloud platform, called SpotCheck [182, 183].

3.1 Motivation and Contributions

Many enterprises, especially technology startup companies, rely in large part on Infrastructure-as-a-Service (IaaS) cloud platforms for their computing infrastructure [52]. Today's IaaS cloud platforms, which enable their customers to rent computing resources on demand in the form of virtual machines (VMs), offer numerous benefits, including a pay-as-you-use pricing model, the ability to quickly scale capacity when necessary, and low costs due to their high degree of statistical multiplexing and massive economies of scale.

To meet the needs of a diverse set of customers, IaaS platforms rent VM servers under a variety of contract terms that differ in their cost and availability guarantees. The simplest type of contract is for an *on-demand* server, which a customer may request at any time and incurs a fixed cost per unit time of use. On-demand servers are *non-revocable*: customers may use these servers until they explicitly decide to relinquish them. In contrast, *spot* servers provide an entirely different type of contract for the same resources. Spot servers incur a *variable* cost per unit time of use, where the cost fluctuates continuously based on the spot market's instantaneous supply and demand. Unlike on-demand servers, spot servers are *revocable*: the cloud platform may reclaim them at any time. Typically, a customer specifies an upper limit on the price they are willing to pay for a server, and the platform reclaims the server whenever the server's spot price rises above the specified limit. Since spot servers incur a risk of unexpected resource loss, they offer weaker availability guarantees than on-demand servers and tend to be cheaper.

This chapter focuses on the design of a *derivative cloud platform*, which repackages and resells resources purchased from native IaaS platforms. Analogous to a financial derivative, a derivative cloud can offer resources to customers with different pricing models, features, and availability guarantees not provided by native platforms using a mix of resources purchased under different contracts. The motivation for derivative clouds stems from the need to better support specialized use-cases that are not directly supported (or are complex for end-users to implement) by the server types and contracts that native platforms offer. Derivative clouds rent servers from native platforms, and then repackage and resell them under contract terms tailored to a specific class of user.

Nascent forms of derivative clouds already exist. PiCloud [14] offers a batch processing service that enables customers to submit compute tasks. PiCloud charges customers for their compute time, and is able to offer lower prices than on-demand servers by using cheaper spot servers to execute compute tasks. Similarly, Heroku [13] offers a Platform-as-a-Service by repackaging and reselling IaaS resources as containers.

As with PiCloud, Heroku constrains the user's programming model—in this case, to containers.

In this chapter, we design a derivative IaaS cloud platform, called SpotCheck, that intelligently uses a mix of spot and on-demand servers to provide high availability guarantees that approach those of on-demand servers at a low cost that is near that of spot servers. Unlike the examples above, SpotCheck does not constrain the programming model by offering unrestricted IaaS-like VMs to users, enabling them to execute any application. The simple, yet key, insight underlying SpotCheck is to host customer applications (within nested VMs) on spot servers whenever possible, and transparently migrate them to on-demand servers whenever the native IaaS platform revokes spot servers. SpotCheck offers customers numerous benefits compared to natively using spot servers. Most importantly, SpotCheck enables interactive applications, such as web services, to seamlessly run on revocable spot servers without sacrificing high availability, thereby lowering the cost of running these applications. We show that, in practice, SpotCheck provides nearly five nines of availability (99.9989%), which is likely adequate for all but the most mission critical applications.

SpotCheck raises many interesting systems design questions, including i) how do we transparently migrate a customer's application before a spot server terminates while minimizing performance degradation and downtime? ii) how do we manage multiple pools of servers with different costs and availability guarantees from native IaaS platforms and allocate (or re-sell) them to customers? iii) how do we minimize costs, while mitigating user risk, by renting the cheapest mix of servers that minimize spot server revocations, i.e., to yield the highest availability? In addressing these questions, we make the following contributions:

Derivative Cloud Design. We demonstrate the feasibility of running disruption-*intolerant* applications, such as interactive multi-tier web applications, on spot servers, by migrating them i) to on-demand servers upon spot server revocation, and ii) back when spot servers become available again. SpotCheck requires live migrating applications from spot servers to on-demand servers within the bounded amount of time between the notification of a spot server revocation and its actual termination. SpotCheck combines several existing mechanisms to implement live bounded-time migrations, namely nested virtualization, live VM migration, bounded-time VM migration, and lazy VM restoration.

Intelligent Server Pool Management. We design server pool management algorithms that balance three competing goals: i) maximize availability, ii) reduce the risk of spot server revocation, and iii) minimize cost. To accomplish these goals, our algorithms intelligently map customers to multiple pools of spot and on-demand servers of different types, and handle pool dynamics caused by sudden revocations of spot servers or significant price changes.

Implementation and Evaluation. We implement SpotCheck on Amazon's Elastic Compute Cloud (EC2) and evaluate its migration mechanisms and pool management algorithms. Our results demonstrate that SpotCheck achieves a cost that is nearly 5× less than equivalent on-demand servers, with nearly five 9's of availability (99.9989%), little performance degradation, and negligible risk of losing VM state.

3.2 Background and Overview

Our work assumes a native IaaS cloud platform, such as EC2, that rents server resources to customers in the form of VMs, and offers a variety of server types that differ in their number of cores, memory allotment, network connectivity, and disk capacity. We also assume the native platform offers at least two types of service contracts—on-demand and spot—such that it cannot revoke on-demand servers once it allocates them, but it can revoke spot servers. Finally, we assume on-demand

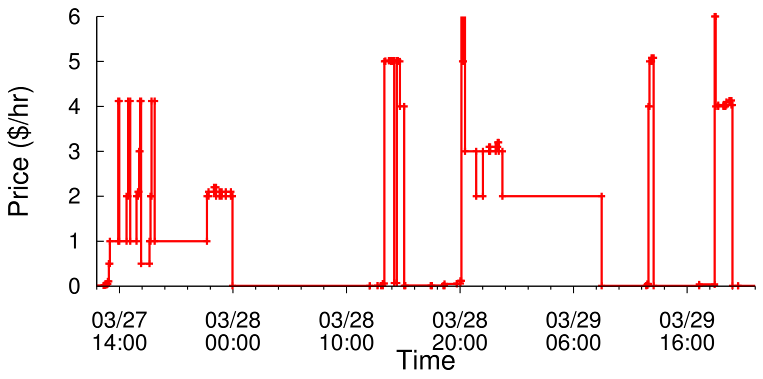


Figure 3.1: Spot price of the `m1.small` server type in EC2 fluctuates over time and can rise significantly above the on-demand price ($0.06 per hour) during price spikes. Note that the y-axis is denominated in dollars and not cents.

servers incur a fixed cost per unit time of use, while the cost of spot servers varies continuously based on the market's supply and demand, as shown in Figure 3.1.[1]

Given the assumptions above, SpotCheck must manage pools of servers with different costs and availability values. While our work focuses on spot servers, largely as defined in EC2, such cost and availability tradeoffs arise in other scenarios. As one example, data centers that participate in demand response (DR) programs offered by electric utilities may have to periodically deactivate subsets of servers during periods of high electricity demand in the grid [137]. While participation in DR programs significantly reduces electricity rates, it also reduces server availability.

Like the underlying native IaaS platform, SpotCheck offers the illusion of dedicated servers to its customers. In particular, SpotCheck offers its customers the equivalent of non-revocable on-demand servers, where only the user can make the decision to relinquish them. SpotCheck's goal is to provide server availability that is close to that of native on-demand servers for a cost that is near that of spot servers. To do so, SpotCheck uses low-cost spot servers whenever possible and "fails over" to high-cost on-demand servers, or other spot servers, whenever the native IaaS platform revokes

[1]Spot price data is from either Amazon's publicly-available history of the spot price's past six months, or from a third-party spot price archive [119].

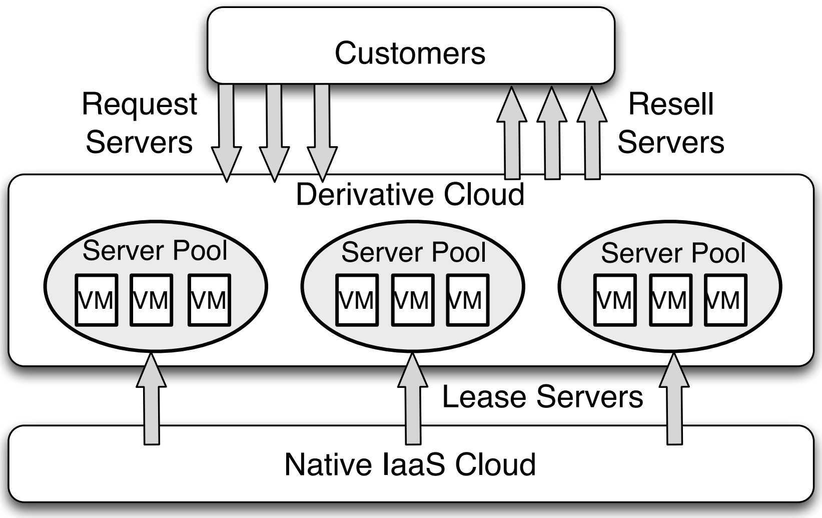


Figure 3.2: A depiction of a derivative IaaS cloud platform.

spot servers. To maintain high availability, migrating from one type of native cloud server to another must be transparent to the end-user, which requires minimizing application performance degradation and server downtime. Section 3.7 quantifies how well SpotCheck achieves these goals.

SpotCheck supports multiple customers, each of which may rent an arbitrary number of servers. Since SpotCheck rents servers from a native IaaS cloud and repackages and resells their resources to its own customers, it must manage pools of spot and on-demand servers of different types and sizes, as depicted in Figure 3.2. Upon receiving a customer request for a new server, SpotCheck must decide which server pool should host the new instance. Upon revocation of one or more native servers from a spot pool, SpotCheck must migrate hosted customers to either an on-demand server pool or another spot pool. SpotCheck intelligently maps customers to pools to spread the risk of concurrent revocations across customers, which reduces the risk of a single customer experiencing a "revocation storm." In some sense, allocating customer requests to server pools is analogous to managing a financial portfolio where funds are spread across multiple asset classes to reduce volatility and market risk.

In addition to server pool management, SpotCheck's other key design element is its ability to seamlessly migrate customer VMs from one server pool to another, e.g., from a spot pool to an on-demand pool upon a revocation, or from an on-demand pool to a spot pool when cheaper spot servers become available. To do this, we rely on the native IaaS platform to provide a small advance warning of spot server termination. SpotCheck then migrates its customers' VMs to native servers in other pools upon receiving a warning, and ensures that the migrations complete in the time between receiving the warning and the spot server actually terminating.

3.3 SpotCheck Migration Strategies

We describe SpotCheck's migration strategies and mechanisms from the perspective of migrating an individual VM from one native cloud server to another. There are a variety of reasons why such a migration may be necessary or desirable—the native IaaS platform may force a migration by revoking the underlying spot server, or a cheaper spot server may become available, which incentivizes migrating a VM running on a more expensive on-demand server to it. Regardless of the reason, SpotCheck combines several virtualization mechanisms to implement its migration strategies.

3.3.1 Nested Virtualization

SpotCheck rents VMs from native IaaS platforms that do not expose all of the functionality of the VM hypervisor. For example, EC2 allocates VMs to its customers, but does not expose control over VM placement or support VM migration to different physical servers. To address this limitation, SpotCheck uses *nested virtualization*, where a nested hypervisor runs atop a traditional VM, which itself runs on a conventional hypervisor [60, 214]. The nested hypervisor enables the creation of nested VMs on the host VM. Since the nested hypervisor does not need special support from the host VM, SpotCheck can install it on VMs rented from native IaaS platforms and use it to migrate nested VMs from one cloud server to another, as depicted in Figure 3.3(a). Nested hypervisors provide a uniform and standard platform for repackaging and reselling virtualized server resources. Nested VMs currently provide paravirtualized I/O devices and hide advanced features, such as SR-IOV [16], which may reduce I/O performance. However, as with the native VM platforms, we expect nested VM technology to continue to improve.

Our SpotCheck prototype uses the XenBlanket nested hypervisor [214]. One benefit of using nested virtualization is that SpotCheck can create multiple nested VMs on a single host VM, allowing it to slice large native VMs into smaller nested VMs and allocate them to different customers, similar to how an IaaS platform slices a physical server into multiple VMs. SpotCheck could also use lighter-weight mechanisms, such as resource containers [54], to isolate partitions of virtualized resources. We chose to use nested VMs in our prototype because the contemporary resource container implementations, e.g., Linux Containers and Docker, do not support the advanced migration features that SpotCheck requires. Besides, SpotCheck's design requirement is to offer an execution environment that are as similar to that provided by the IaaS provider. SpotCheck thus uses nested VMs that are nearly identical to the IaaS VMs from an application's point of view.

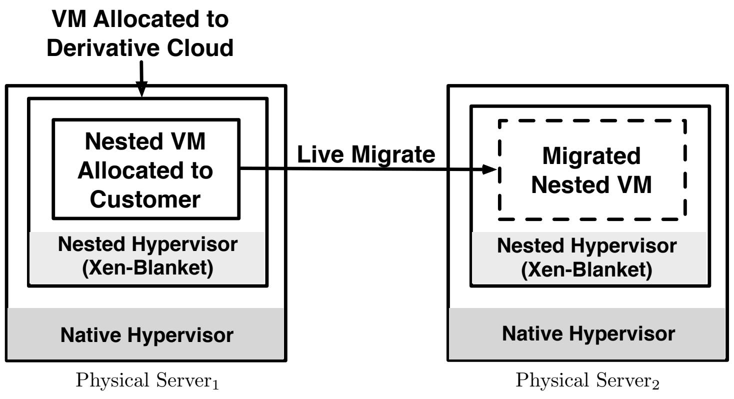
3.3.2 VM Migration

Since SpotCheck runs nested hypervisors on VM servers acquired from native IaaS platforms, it has the ability to migrate nested VMs from one server to another. SpotCheck leverages two VM migration mechanisms to implement its migration strategy: live migration and bounded-time VM migration. Live VM migration enables SpotCheck to migrate a nested VM from one server to another, while incurring nearly zero downtime to a customer's resident applications, as depicted in Figure 3.3(a).

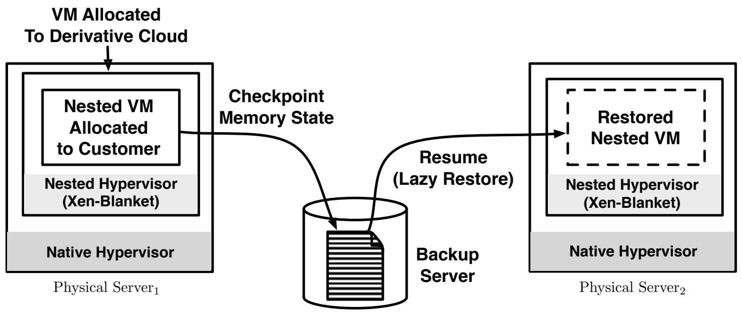
Prior work proposes a variety of live VM migration mechanisms and optimizations, such as the pre-copy [77] and post-copy [114] migration variants.

In general, the total latency to live migrate a VM, whether nested or not, is proportional to the size of the VM's memory. Thus, larger VMs with tens of gigabytes of RAM may take several minutes, while smaller VMs with a few gigabytes of RAM may take tens of seconds. In addition to memory size, the write (or dirtying) rate of memory pages, which depends on application characteristics, also influences live migration latency. As a result, live VM migration is not suitable in all of SpotCheck's migration scenarios. In particular, an IaaS platform may revoke a spot server at any time, while providing only a small warning period for the server to complete a graceful shutdown. Once the warning period ends, the IaaS platform forcibly terminates the VM. For example, EC2 provides a warning of 120 seconds before forcibly terminating a spot server. While the 120 second warning has always been a well-known hidden feature of spot servers, Amazon publicly acknowledged it in January 2015 and now supports official 120 second termination notices for spot servers through its external web services API [56]. Importantly, if the latency to live migrate a VM exceeds the warning period, as it often does with large memory sizes, then the IaaS platform will terminate the spot server and any resident nested VMs before their migrations complete, resulting in the loss of memory state at best and VM failure at worst.

In this scenario, SpotCheck leverages an alternative migration approach, called bounded-time VM migration [189, 190], which provides a guaranteed upper bound on migration latency that is independent of a VM's memory size or the dirtying rate of memory pages. Supporting bounded-time VM migration requires maintaining a partial checkpoint of a VM's memory state on an external disk by running a background process that continually flushes dirty memory pages to a backup server to ensure the size of the dirty pages does not exceed a specified threshold. This threshold is chosen such that any outstanding dirty pages can be safely committed upon a revocation


VM Allocated to Derivative Cloud
Nested VM Allocated to Customer
Nested Hypervisor (Xen-Blanket)
Native Hypervisor
Physical Server$_1$
Live Migrate
Migrated Nested VM
Nested Hypervisor (Xen-Blanket)
Native Hypervisor
Physical Server$_2$

(a) Live migration using nested virtualization.



(b) Memory checkpointing and restoration when using a bounded-time VM migration.

Figure 3.3: SpotCheck uses live and bounded-time VM migration to migrate nested VMs within an IaaS platform.

within the time bound [189, 190]. The VM may then resume from the saved memory state on a different server, as depicted in Figure 3.3(b).

SpotCheck adapts and applies both live [77] and bounded-time VM migration [189, 190] to nested VMs. Depending on the scenario, SpotCheck uses the most appropriate technique for VM migration. When migrating a nested VM from an on-demand server to a spot server, e.g., when a cheaper spot server becomes available, SpotCheck uses live migration regardless of the nested VM's memory size, since there is no constraint on the migration latency. SpotCheck then voluntarily relinquishes the native VM as soon as the migration completes. When migrating a nested VM from a revoked spot server, bounded-time VM migration is usually necessary, since the migration must complete before the spot server terminates. The only exception is for “small" nested

VMs that do not use much memory, such that a live migration is able to reliably complete within a spot server's warning period, e.g.,120 seconds for EC2.

Of course, the shorter the warning period, the smaller the nested VM memory size that cannot use a conventional live migration and will require a bounded-time VM migration. SpotCheck may also perform *proactive* migrations from a spot server if it predicts that a revocation is imminent. In this case, the system has less stringent time constraints on the migration latency, since it triggers the migration before the IaaS platform explicitly signals a revocation. Such predictive approaches make it feasible to employ live migration with spot servers and avoid the overhead and complexity of bounded-time VM migration, which requires continually backing up memory state to a remote disk. However, such optimizations incur significant risk of losing VM state unless they are able to predict an imminent revocation with high confidence, e.g., by tracking and predicting a rise in market prices of spot servers that causes revocations.

To support bounded-time VM migration, SpotCheck must manage a pool of backup servers that store the memory state of nested VMs on spot servers, and continuously receive and commit updates to nested VM memory state. As we show in our experiments in Section 6.7, each backup server is able to host tens of nested VMs without degrading their performance, which makes the incremental cost of using such additional backup servers small in practice.

3.3.3 Lazy VM Restoration

Bounded-time VM migration is a form of VM suspend-resume that saves, or suspends, the VM's memory state to a backup server within a bounded time period, and then resumes the VM on a new server. Resuming a VM requires restoring its memory state by reading it from the disk on the backup server into RAM on the new server. The VM cannot function during the restoration process, which causes downtime until the VM state is read completely into memory. Since the downtime of

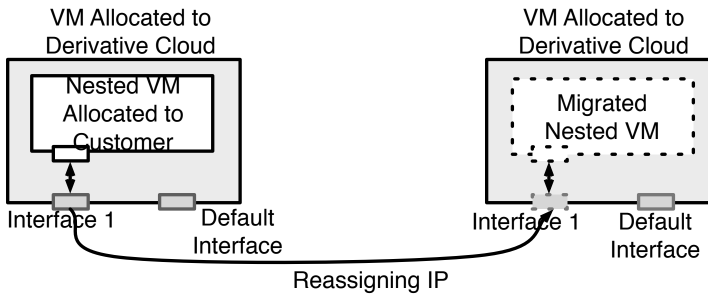

VM Allocated to Derivative Cloud
Nested VM Allocated to Customer
Interface 1
Default Interface
VM Allocated to Derivative Cloud
Migrated Nested VM
Interface 1
Default Interface
Reassigning IP

Figure 3.4: SpotCheck migrates the network interface of nested VMs from the source to the destination host using VPN functions provided by the underlying IaaS platform.

this traditional VM restoration is disruptive to customers, SpotCheck employs lazy VM restoration, as proposed in prior work [114, 132], to reduce the downtime to nearly zero. Lazy VM restoration involves reading a small number of initial VM memory pages—the *skeleton state*—from disk into RAM and then immediately resuming VM execution without any further waiting.

The remaining memory pages are fetched from the backup server on demand, akin to virtual memory paging, whenever the VM's execution reads or writes any of these missing pages. A background process also runs in parallel and proactively reads memory pages into RAM to reduce the frequency of page faults. Lazy VM restoration substantially reduces the latency to resume VM execution at the expense of a small window of slightly degraded performance, due to any page faults that require reading memory pages on demand. Combining lazy VM restoration with bounded-time VM migration enables a new "live" variant of bounded-time VM migration that minimizes the downtime when migrating VMs within a bounded time period upon revocation.

3.3.4 Virtual Private Networks

While the migration mechanisms above minimize customers' downtime and performance degradation during migrations, maximizing transparency also requires that the IP address of customers' nested VMs migrate to the new host to prevent breaking any active network connections. In a traditional live migration, the VM emits an `arp` packet to inform network switches of its new location, enabling switches to forward subsequent packets to the new host and ensuring uninterrupted network connections for applications [77]. However, in SpotCheck, the underlying IaaS platform is unaware of the presence of nested VMs on the host VMs. SpotCheck currently employs a separate physical interface on the host VM to provide each nested VM its own IP address, in addition to the host's default interface and IP address. Thus, SpotCheck configures Network Address Translation (NAT) in the nested hypervisor to forward all network packets arriving at an IP address to its associated nested VM. IaaS platforms, such as EC2, make this feasible by supporting the creation of multiple interfaces and IP addresses on each host. However, since the IP address is associated with the host VM, the address does not automatically migrate with the nested VM. Instead, SpotCheck must take additional steps to detach a nested VM's address from the host VM of the source and reattach it to the destination host.

While many IaaS platforms still treat IP address creation and assignment as privileged operations, a few platforms, including EC2, have introduced virtual private networking (VPN) functions to provide users control over their own private IP address space. EC2 supports VPNs through its Virtual Private Cloud (VPC) feature, which enables users to directly assign IP addresses to their VMs. SpotCheck creates a VPC and places all of its spot and on-demand servers into it. As a result, SpotCheck is able to create a private IP address for each nested VM. Upon migration, SpotCheck uses available VPC functions to deallocate the IP address associated with a nested VM on its source server, and reassign it to a new (unused) network interface on the destination server, as depicted in Figure 3.4. This ensures the IP address of nested VMs remains unchanged after migration. SpotCheck currently allocates a subnet within a shared data plane, defined by the VPC, to each customer. By default, SpotCheck assigns one public IP address per customer, attached to a designated "head" nested VM, to provide access to the public Internet from within the private VPC subnet.

3.3.5 Putting it all together

SpotCheck combines nested virtualization, virtual private networks, VM migration, and lazy VM restoration to implement its migration strategies, as summarized below. Upon initial allocation, SpotCheck assigns a backup server to each nested VM on a spot server, which stores its memory state, unless the nested VM's memory size is small enough to ensure a live migration completes within the warning period. SpotCheck might also not assign a backup server if it decides to migrate nested VMs proactively in advance of a revocation. Nested VMs hosted on on-demand servers do not require a backup server, since they are always capable of a live migration. If the underlying IaaS platform revokes a spot server, SpotCheck must migrate each resident nested VM to a new destination server via bounded-time VM migration.

The destination server is chosen by a higher-level server pool management algorithm, discussed in Section 4. Once the VM's migration completes, SpotCheck uses VPC functions to deallocate the IP address on the source server, and then reallocate the IP address on the destination server and configure the nested hypervisor to forward packets to the new address. SpotCheck also must detach the VM's network-attached disk volume and reattach it to the destination server before the VM resumes operation. We discuss SpotCheck's treatment of storage more in Section 5.4. If SpotCheck employs bounded-time VM migration, it uses lazy VM restoration to minimize the migration downtime.

3.4 Server Pool Management

SpotCheck rents VM servers from native IaaS platforms under different service contracts that specify different levels of price and availability, and then repackages and resells their resources to its customers. The ability to rent and manage servers of different types, and intelligently multiplex their resources across multiple customers is central to the design of any derivative cloud, including SpotCheck. Note that, similar

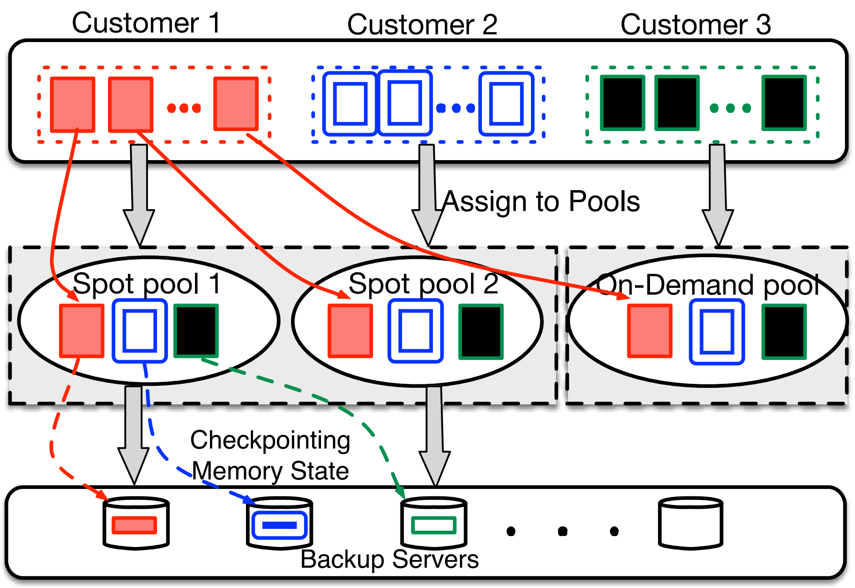


Figure 3.5: SpotCheck's architecture using multiple pools.

to traditional virtualization, nested virtualization enables multiple nested VMs to run on a host VM, such that the nested hypervisor in the host VM isolates the nested VMs and prevents cross-VM attacks. As with a native IaaS platform, SpotCheck controls the nested hypervisor and has full access to the memory state of each of its customer's nested VMs. In this section, we describe the techniques SpotCheck uses to manage resources from multiple pools of servers.

3.4.1 SpotCheck Architecture

At an architectural level, SpotCheck maintains multiple pools of servers, as shown in Figure 3.5, where each pool contains multiple native VM servers of a particular type, specifying an allotment of CPU cores with specified performance, memory-size, network bandwidth, etc. For each server type, SpotCheck maintains separate spot and on-demand pools, comprising spot and on-demand servers of the same type, respectively. SpotCheck exposes a user interface similar to that of a native IaaS platform, where customers may request and relinquish servers of different types. However, SpotCheck offers its customers the abstraction of non-revocable servers, despite often executing them on revocable spot servers.

SpotCheck maps its customers' nested VMs, which may be of multiple types, to different server pools, as illustrated in Figure 3.5. In addition, SpotCheck also maintains a pool of backup servers, each capable of maintaining checkpoints of memory state for multiple nested VMs hosted on spot servers. Thus, SpotCheck assigns each native server from a *spot pool* to a distinct backup server, such that any nested VMs hosted on it write their dirty memory pages to their backup server in the background. SpotCheck does not assign native servers in the on-demand pool to a backup server, since they can live migrate any nested VMs hosted on them without any time constraints. Given the architecture above, we next describe the techniques and algorithms SpotCheck employs to manage server pools and handle pool dynamics.

3.4.2 Mapping Customers to Pools and Pools to Backups

When a customer requests an instance of a specific size, e.g. small server, SpotCheck must make trade-offs between cost, stability, and the frequency of concurrent revocations. That is, SpotCheck ideally would allocate stable server resources at cheap prices and avoid any customer losing significant (or all of their) spot servers at once. To satisfy such a requirement, SpotCheck makes a sequences of decisions by taking into account both a spot server's price history and a customer's existing spot allocation.

In the simplest case, when a customer requests a new VM of a certain type, SpotCheck satisfies the request by allocating a native VM of the same type from the underlying IaaS platform, and then configures a nested VM within the native VM for use by the customer. Since nested virtualization supports the ability to run multiple nested VMs on a single host VM, SpotCheck also has the option of i) requesting a larger native VM than the one requested by the customer, ii) slicing it into smaller nested VMs of the requested type, and then iii) allocating one of the nested VMs to the customer. Slicing a native VM into smaller nested VMs is useful, since prices for spot servers of different types vary based on market-driven supply and demand. As a result, the price of a spot server that is two or four times the size of a requested nested VM may be less (or more) than two to four times the price of a smaller spot server of the requested type.

Presented with a set of spot markets to choose from, SpotCheck employs three different policies in choosing the spot server type. The first strategy, referred to as *cheapest-first*, is a simple greedy policy that chooses the cheapest spot server, based on the current prices, to satisfy a request. We exploit the fact that the server size-to-price ratio is not uniform: a large server, say a `m3.large`, which is able to accommodate two medium VM servers of size `m3.medium` may be *cheaper* than buying two medium servers. Since the pricing of on-demand servers is roughly proportional to their resource allotment, such that a server with twice the CPU and RAM of another costs roughly twice as much, under ideal market conditions, the price of spot servers should also be roughly proportional to their resource allotment. However, we have observed that different server types experience different supply and demand conditions. In general, smaller servers appear to be more in demand than larger servers because their spot price tends to be closer to their on-demand price. As a result, larger servers are often cheaper, on a unit cost basis, than smaller server for substantial periods of time, which enables SpotCheck's greedy approach to exploit the opportunity for arbitrage. However, note that whenever SpotCheck slices a spot server into multiple nested VMs, it does incur additional risk, as a revocation requires migrating *all* of its resident nested VMs.

An alternative to the greedy cheapest-first strategy above is a conservative *stability-first* policy that allocates a native spot server (from the various possible choices) with the most stable prices. To increase availability, SpotCheck must reduce both the frequency of revocation events and the impact of each one, e.g., due to downtime. Allocating a spot server with a stable market price reduces the probability of a spot server revocation, which in turn increases availability.

Both cheapest-first and stability-first strategies do not consider the existing allocation of a customer's spot servers. Such strategies might be problematic when a customer's spot instances all belong to a single server pool, incurring concurrent revocations upon price spikes. A revocation event due to a price spike for a particular type of spot server can cause concurrent revocations within a single spot pool. However, different pools are *independent*, since spot prices of different server types fluctuate independently of one another and are uncorrelated, as seen in Figures 3.6(c) and (d). Hence, SpotCheck also supports a more sophisticated policy that bounds the maximum concurrent revocations per customer, defined as r, by distributing a customer's nested VMs across multiple pools. Revocation storms degrade nested VM performance and increase downtime by overloading backup servers, which must simultaneously broker the migration of every revoked nested VM. SpotCheck employs this policy to reduce the risk of a sudden price spike causing mass revocations of spot servers of a particular type at one location (or availability zone in EC2 parlance).

The key idea of this bounded greedy algorithm is to first identify the cost and stability ranges using cheapest-first and stability-first strategies, and then search for a specific spot server type that has cost and stability within the above ranges without violating concurrent threshold r. SpotCheck also favors the spot server type that incurs fewer concurrent revocations, i.e. smaller instances, as a tie breaker. This tie breaker is beneficial because it allows SpotCheck to maintain a reasonable amount of sliced servers, in case of customer shortage. Further, SpotCheck sets up a threshold of maximum number of concurrent revocations allowed, constraining the candidate server types.

Finally, SpotCheck must assign each nested VM within a spot pool to a distinct backup server. SpotCheck also distributes nested VMs in a spot pool across multiple backup servers. Since each spot pool is subject to concurrent revocations, spreading one pool's VMs across different backup servers reduces the probability of any one backup server experiencing a large number of concurrent revocations. The approach also spreads the read and write load due to supporting bounded-time VM migration across multiple backup servers. SpotCheck employs a simple round-robin policy to map nested VMs within each pool across the set of backup servers. Once every backup server becomes fully utilized, SpotCheck provisions a native VM from the IaaS platform to serve as a new backup server, and adds it to the backup server pool. Of course, a backup server is not necessary for running stateless services on nested VMs, e.g., a single web server that is part of a tier of replicated web servers, since these services are designed to tolerate failures. However, as with any IaaS platform, SpotCheck does not make any assumptions about the applications that run on it. This does mean that SpotCheck may incur slightly higher costs than necessary for stateless services, since these servers can use spot servers directly without incurring extra costs for a backup server or requiring any application modifications. The policies for mapping VMs to backup servers are expanded on in Section 3.5.2.

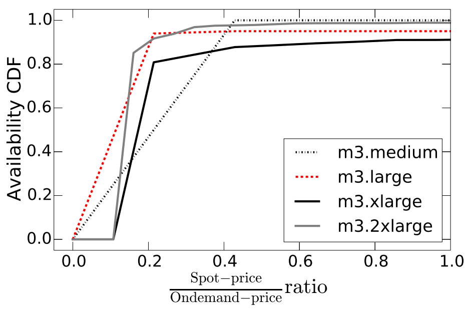
3.4.3 Handling Pool Dynamics

After the initial mapping of a nested VM onto a server in a pool, SpotCheck will likely migrate it to servers in other pools over the course of its lifetime due to pool dynamics. There are two types of pool dynamics caused by changing spot prices that SpotCheck must handle. The first is *revocation dynamics*, which cause the sudden revocation of one or more spot servers within a pool due to prices rising above the bid price. The second is *allocation dynamics*, which dictates when to transition a nested VM back from an on-demand to a spot server when a price spike abates and the spot price again drops below the on-demand price. Note that, in EC2, spot prices often rise substantially above the on-demand price during a price spike, as depicted in Figure 3.1.

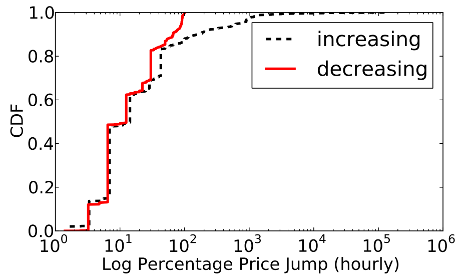
Although SpotCheck has no control over the fluctuating price of spot servers, it does have the ability to determine a maximum bid price it is willing to pay for servers in each of its spot pools. Designing "optimal" bidding strategies in spot markets in various contexts is an active research area, and prior work has proposed a number of different policies [119, 219, 61]. Adapting these policies to SpotCheck's context may be possible. However, since our focus is on designing a derivative IaaS cloud, rather than bidding strategies, SpotCheck currently employs one of two simple policies: either bid the equivalent on-demand price for a spot server or bid k times the on-demand price. With the first policy, SpotCheck retains spot servers in a pool as long as those servers' spot price remains below the equivalent on-demand price of the servers. If the spot price rises above the on-demand price, the IaaS platform revokes the spot servers in the pool, which forces SpotCheck to migrate them to on-demand servers. Of course, this revocation only occurs if the equivalent on-demand servers are now cheaper than the spot servers, so migrating to on-demand servers at these times is the cheapest, most cost-effective strategy.

The second policy bids a price that is k times the on-demand price, where $k > 1$. In general, the higher the bid price, the lower the probability of an IaaS platform revoking the spot servers in a pool. Bidding a high price that exceeds the on-demand price lowers a pool's revocation frequency at the expense of a higher cost. This policy also makes proactive migrations more feasible, since SpotCheck can periodically monitor prices and proactively trigger live migrations to on-demand servers whenever prices rise above the on-demand price, but are still lower than the bid price. Thus, SpotCheck currently only uses proactive migrations in conjunction with this second policy.

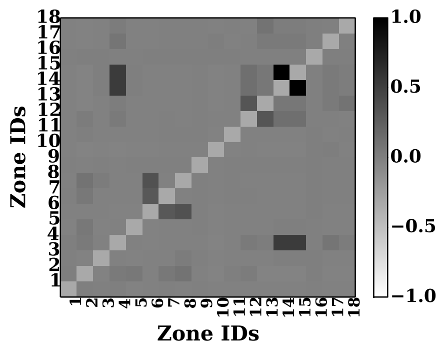
In the case of EC2's spot market, empirical data shows that the probability of revocation decreases with higher bid prices, but it flattens quickly, such that the "knee" of the curve, as depicted in Figure 3.6(a), is slightly lower than the on-demand

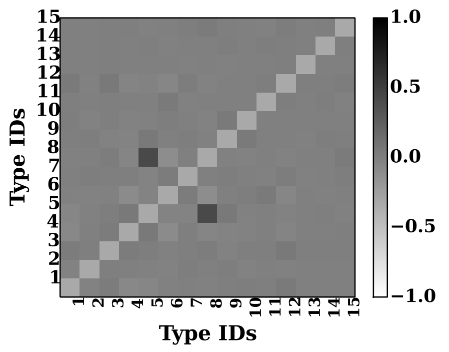


(a) CDF of prices



(b) Price changes are large





(c) Correlations of prices between zones.

(d) Correlations between instance types.

Figure 3.6: Price dynamics across EC2 spot markets from April to October 2014 for all `m3.*` types: the spot price distribution (a) has a long tail, (b) exhibits large price changes, and (c) is uncorrelated across locations and server types (d).

price. Thus, simply bidding the on-demand price is an approximation of bidding an “optimal" value that is equal to the knee of this availability-bid curve. This implies that large price spikes are the norm, with spot prices frequently going from well below the on-demand price to well above it, as shown in Figure 3.6(b). Figure 3.6(a) also shows that the spot prices are extremely low on average compared to the equivalent prices for on-demand servers. This is likely due to the complexity of modifying applications to effectively use the spot market, which suppresses demand by limiting spot servers to a narrow class of batch applications.

In either case, in SpotCheck's current implementation, all servers within a spot pool have the *same* bid price. As a result, when the market price rises above the bid price, the IaaS platform revokes all servers within a pool at the same time, resulting in a *revocation storm*. A simple approach to handling concurrent revocations is to request an equivalent number of on-demand servers from the IaaS platform and migrate each nested VM to a new on-demand server. An alternative approach is to request spot servers of a different, larger type where prices are stable, and then migrate to new spot servers. However, requesting new servers in a lazy fashion when necessary is only feasible if the latency to obtain them is smaller than the warning period granted to a revoked server. For example, empirical studies have shown that it takes up to 90 seconds to start up a new on-demand server in EC2 [141], while the warning period for a spot server is two minutes, which leaves only 30 seconds to migrate the spot server's state to the new server. If the allocation latency were to exceed the warning time, such a lazy strategy is not possible due to the risk of significant VM downtime.

To handle this scenario, SpotCheck is able to maintain a pool of hot spares to immediately receive nested VMs from revoked spot servers without waiting for a new server to come online. Hot spares increase SpotCheck's overhead cost, while reducing the risk of downtime. Note that there is never a risk of losing nested VM state, since the backup server stores it even if there is not a destination server available to execute the nested VM. An alternative approach to using dedicated hot spares is to use existing servers in other stable pools as staging servers. This approach is attractive if these existing servers are not fully utilized by the nested VMs running on them. Here, the staging servers only run the nested VMs from a revoked spot server temporarily, while SpotCheck makes concurrent requests for new on-demand or spot servers to serve as the final destination. Of course, this strategy doubles the number of migrations and the associated overhead, but it also enables the system to reduce risk without increasing its costs. Hot spares and staging servers may also serve as a temporary haven for displaced spot VMs, in the rare case when requests for on-demand servers fail because they are unavailable from the IaaS platform. While native IaaS platforms attempt to provision resources to stay ahead of the demand curve, they occasionally run out of on-demand servers if the demand for them exceeds their supply.

Of course, regardless of the risk mitigation strategies above, SpotCheck cannot provide higher availability than the underlying IaaS platform. For example, if the IaaS platform fails or becomes disconnected, as occasionally happens to EC2 [78], SpotCheck would also fail. Since we do not have access to long-term availability data for EC2 or other IaaS platforms, in our experiments, the term "availability" refers to relative availability with respect to the underlying IaaS platform, which we assume in this paper is 100% available.

3.4.4 Cost and Availability Analysis

SpotCheck's goal is to provide resources, in the form of nested VMs, with high availability that resembles that of on-demand servers, but at low prices that resembles those of spot servers. In this section, we analyze the costs incurred by SpotCheck's server pool management and migration strategies, and their resulting availability.

Given n customers, each with C_i servers, SpotCheck must provision a total of $V = \sum_i^n C_i$ nested VMs. Since SpotCheck maps these V nested VMs onto multiple pools, the total cost L of renting native servers from the IaaS platform is equal to the cost of the necessary spot servers plus the cost of the necessary on-demand servers plus the cost of any backup servers. Thus, the amortized cost per nested VM is L/V. We represent the expected cost $E(c)$ of an individual nested VM as $E(c) = (1-p)E(c_{spot}(t)) + p \cdot c_{od}$, where p denotes the probability of a revocation when it resides on a spot server, $c_{spot}(t)$ denotes the variable price of the spot server, and c_{od} denotes the price of the equivalent on-demand server. We note that p is simply the probability of the spot price rising above the bid price, i.e., $p = P(c_{spot}(t) > bid)$, which is given by the cumulative distribution shown in Figure 3.6(a) that we derive empirically for different spot pools.

To compute a nested VM's availability, assume that the market price of a spot server changes once every T time units, such that the server will be revoked once every

T/p time units, yielding a revocation rate of $R = p/T$. Here, we assume live migration does not result in significant downtime, while bounded-time VM migration incurs the downtime required to i) read sufficient memory state after a lazy restoration, ii) attach a networked disk volume to the new server, and iii) reassign the IP address to the new server. If D denotes the delay to perform these operations, the downtime experienced by the nested VM is $D \cdot R$ per unit time, i.e., $D \cdot p/T$. Thus, our expected cost equation above allows us to analyze different pool management and bidding policies. This expected cost includes the cost of running the nested VM on either a spot or on-demand server, and the cost of any backup servers. We also assume that nested VMs use an associated EBS volume in EC2 to provide persistent network-attached storage. However, we do not include storage costs, since they are negligible at the backup server, and thus the same when using SpotCheck or the native IaaS platform. Similarly, our analysis does not include costs associated with external network traffic, since these costs are the same when using SpotCheck or the native IaaS platform. Note that there is no cost in EC2 associated with the network traffic between nested VMs and their backup server, since network traffic between EC2 servers incurs no charge.

One caveat in our analysis is that we do not consider the second-order effects of our system on spot prices and availability. While it is certainly possible that widespread use of SpotCheck may perturb the spot market and affect prices, our analysis assumes that the market is large enough to absorb these changes. Regardless, our work demonstrates that a substantial opportunity for arbitrage exists between the spot and on-demand markets. Consumers have a strong incentive to exploit this arbitrage opportunity until it no longer exists. SpotCheck also benefits the IaaS platform, since it should raise the demand and price for spot servers by opening them up to a wider range of applications. Thus, there is no incentive for EC2 to hinder (or prevent) SpotCheck by reducing (or eliminating) the warning notification for spot servers.

The increasing popularity and demand of derivative clouds might also incentivize IaaS platforms to increase their pool of spot servers. However, our analysis assumes that on-demand servers of some type will always be available. While on-demand servers of a particular type may become unavailable, we assume the market is large enough such that on-demand servers of some type are always available somewhere. As we discuss, SpotCheck's pool management strategies operate across multiple markets by permitting the unrestricted choice of server types and availability zones (within a region). These strategies protect against the rare event where one type of on-demand server becomes unavailable.

3.5 Risk Management

This section describes the various risks encountered when running a derivative cloud on inherently volatile markets and presents multiple policies to manage these risks. SpotCheck's migration strategies provide system-level mechanisms that leverage spot servers by migrating applications away from spot servers upon revocation. However, running applications using these revocable spot servers requires managing multiple risks in order to reduce the number of revocation events, reduce the impact of revocation storms, and maintain the efficiency of restoration. SpotCheck manages potential risks in all three facets with a combination of policies that intelligently manages customer server pools, the backup server pool, and hot spare servers.

3.5.1 Reducing Revocation Risks using Bidding

Once a spot market has been decided for a VM, SpotCheck must determine a bid price. Although SpotCheck has no control over the fluctuating price of spot servers, it does have the ability to determine a maximum bid price it is willing to pay for servers in each of its spot pools. Designing "optimal" bidding strategies in spot markets in various contexts is an active research area, and prior work has proposed a number of different policies [119, 219, 61]. Adapting these policies to SpotCheck's context may be possible. However, since our focus is on designing a derivative IaaS cloud, rather than bidding strategies, SpotCheck currently employs one of two simple policies: either place a single bid or bid at multiple different prices for a specific server type.

3.5.1.1 Single Level Bidding

With the single bid policy, SpotCheck picks a single bid price for every spot pool. The bid is chosen to minimize the expected cost of running on the spot instances and on-demand instances due to the revocation. A low bid implies a higher revocation rate and more time spent running on on-demand servers, and thus nullifies the lower average spot price. Similarly, a high bid price reduces revocations, but results in increased spot instance costs. In order to balance the tradeoff, SpotCheck finds a biding price b^* that is at the "knee" point of the revocation probability curve. Put simply, a "knee" point appears when the probability curve flattens out and can be found by calculating the local maxima of the curve.

In the case of EC2's spot market, empirical data shows that the probability of revocation decreases with higher bid prices, but it flattens quickly, such that the "knee" of the curve, as depicted in Figure 3.6(a), is slightly lower than the on-demand price. Thus, simply bidding the on-demand price is an approximation of bidding an "optimal" value that is equal to the knee of this availability-bid curve. This implies that large price spikes are the norm, with spot prices frequently going from well below the on-demand price to well above it, as shown in Figure 3.6(b). Figure 3.6(a) also shows that the spot prices are extremely low on average compared to the equivalent prices for on-demand servers. This is likely due to the complexity of modifying applications to effectively use the spot market, which suppresses demand by limiting spot servers to a narrow class of batch applications.

The cost-optimal bidding level can be found with the following model. Given n customers, each with C_i servers, SpotCheck must provision a total of $V = \sum_i^n C_i$ nested VMs. Since SpotCheck maps these V nested VMs onto multiple pools, the total cost L of renting native servers from the IaaS platform is equal to the cost of the necessary spot servers plus the cost of the necessary on-demand servers plus the cost of any backup servers. Thus, the amortized cost per nested VM is L/V.

We represent the expected cost of running a spot server with a bid b as $E[c(b)]$ and it is:

$$E[c(b)] = (1-p) \cdot E[c_{spot}(b)] + p \cdot c_{od} + \epsilon \tag{3.1}$$

where p denotes the probability of a revocation when it resides on a spot server, $E[c_{spot}(b)]$ denotes the average price of the spot server at a bid b, and c_{od} denotes the price of the equivalent on-demand server. We note that p is simply the probability of the spot price rising above the bid price, i.e., $p = P(c_{spot}(b) > bid)$, which is given by the cumulative distribution shown in Figure 3.6(a) that we derive empirically for different spot pools. Finally, the additional small constant cost, ϵ denotes the amortized cost to run the backup servers. A single backup server with cost c_b can be shared by multiple (N) VMs, yielding $\epsilon = c_b/N$. SpotCheck's optimized backup server design can support upto 40 VMs, and thus the extra cost associated with backup servers is quite small.

The expected costs can be calculated for any bid level, and only requires availability and price information, both of which are obtained using the publicly available price traces published by Amazon. In order to find the optimum bid level b^* which minimizes $E[C(b)]$ in Equation 3.1, a simple numerical search using gradient descent is used to find the minima and the associated bid level. This operation is performed only once per spot market, and is re-run only upon significant price changes.

To compute a nested VM's availability, assume that the market price of a spot server changes once every T time units, such that the server will be revoked once every T/p time units, yielding a revocation rate of $R = p/T$. Here, we assume live migration does not result in significant downtime, while bounded-time VM migration incurs the downtime required to i) read sufficient memory state after a lazy restoration, ii) attach a networked disk volume to the new server, and iii) reassign the IP address to the new server. If D denotes the delay to perform these operations, the downtime experienced by the nested VM is $D \cdot R$ per unit time, i.e., $D \cdot p/T$.

Thus, our expected cost equation above allows us to analyze different pool management and bidding policies. This expected cost includes the cost of running the nested VM on either a spot or on-demand server, and the cost of any backup servers. We also assume that nested VMs use an associated EBS volume in EC2 to provide persistent network-attached storage. However, we do not include storage costs, since they are negligible at the backup server, and thus the same when using SpotCheck or the native IaaS platform. Similarly, our analysis does not include costs associated with external network traffic, since these costs are the same when using SpotCheck or the native IaaS platform. Note that there is no cost in EC2 associated with the network traffic between nested VMs and their backup server, since network traffic between EC2 servers incurs no charge.

3.5.1.2 Multi Level Bidding

Alternatively, SpotCheck also supports multi-level bidding within a spot market. The goal of this bidding strategy is to reduce the number of concurrent revocations and thus mitigate the occurrence of revocation storms. Bidding at multiple levels means that a price increase does not necessarily affect *all* the servers in a market. We use a simple, two-level bidding strategy wherein we have a *low* and a *high* bid. Servers are randomly placed either in the low bid pool or the high bid pool. A spot price increase is going to affect the low-bid servers first and cause them to be revoked; it only affects the high-bid servers when the price crosses the high bid, which may happen after a small delay as the price ramps up, or may not happen at all. Of course, a sudden increase in price above the high-bid mark will cause all the servers to be revoked simultaneously. If the gap between revocation of the low and high bid servers is large enough, then the impact of the revocation storm is reduced, because the backup server will have to bear the brunt of only half the number of concurrent migrations. Additionally, requesting a smaller number of on-demand servers may also reduce the latency of server acquisition [141].

Our two-level bidding strategy is as follows. The low-bid is set to the on-demand price (as before), and then we use a numerical search approach to find the high-bid. Servers are equally and randomly distributed among the two bid levels. Just like in single level bidding, there is a tradeoff between the bid and the cost. A bid higher than the on-demand price means that we are ready to pay that price, and thus bidding too high is not cost optimal.

Therefore, when choosing the bid levels, we seek to minimize the i) revocation storm size and ii) expected cost. In single level bidding, a revocation storm affects all n of the servers in that market. With two-level bidding, some storms affect only $n/2$ servers, and thus their impact is said to be mitigated. We thus use the *fraction of revocation storms mitigated*, f_r as a metric. A storm is mitigated if the gap between revocation of high and low bid servers is at least t_h. Once the t_h threshold is crossed, the high and low bid revocations will not be simultaneous because the backup server will have finished lazily restoring the VMs. Based on experimental analysis, we set $t_h = 10$ minutes.

Since the low-bid is fixed (equal to on-demand price), we use a simple numerical search for the high-bid which maximizes the fraction of revocation storms mitigated, f_r, such that the increase in cost stays under a threshold. Thus, we have the constraint:

$E[c_r] \leq \alpha \cdot E[C]$, where $E[C]$ is the expected cost for the single-level bidding policy. Expectations for both f_r, c_r are obtained by using historical price traces, and we use an $\alpha = 0.2$, i.e., we limit the increase in cost to 20%. The upper bound on the search for the high-bid is enforced by Amazon, which limits the maximum bid to be 10 times the on-demand price.

3.5.2 Reducing Concurrent Revocations with Backup Servers

After requesting spot servers from the native IaaS platform, SpotCheck must assign each nested VM within a spot pool to a distinct backup server. SpotCheck also distributes nested VMs in a spot pool across multiple backup servers. The task of assignning VMs to backup servers is analogous to VM placement and server consolidation [150] where the goal is to pack VMs onto a minimum number of physical servers.

Since each spot pool is subject to concurrent revocations, spreading one pool's VMs across different backup servers reduces the probability of any one backup server experiencing a large number of concurrent revocations. The approach also spreads the read and write load due to supporting bounded-time VM migration across multiple backup servers. To this end, SpotCheck employs a round-robin policy to map the nested VMs within each pool across the set of backup servers. With the **round-robin policy**, SpotCheck simply assigns each nested VM to the next available backup server[2]. If any backup server becomes fully utilized, SpotCheck provisions a native VM from the IaaS platform to serve as a new backup server, and adds it to the backup server pool. A backup server in SpotCheck can host multiple($N = 40$) VMs, and may not always be fully utilized. The under-utilization can occur because VM arrivals and lifetimes are dynamic, and SpotCheck does not have apriori knowledge about VM creation/termination requests which it can use to provision the minimum number of backup servers.

SpotCheck's round-robin policy might lead to unbalanced backup servers in terms of concurrent revocations. The optimal mapping from spot servers to backup servers to minimize the maximum number of concurrent revocations can be formulated as an integer linear program (ILP).

Let N spot servers belong to different spot pools (S), and p_{is} denote the mapping between servers and server-pools. The backup servers are denoted by M, and their capacity is denoted by U. Our goal is to find an optimal mapping X for every spot server, where x_{ij} denotes assignment of server i to backup j. We can then represent the number of concurrent revocations cr_j of backup server j: $cr_j = \arg\max_{s \in S} \sum_{i=0}^{m} p_{si} x_{ij} w_i$. Intuitively, cr_j is defined by the largest server pool hosted. We define revocation storm severity to be the maximum concurrent revocations on any backup server $cr = \max_j cr_j$. Our objective is to minimize cr with the following constraints:

$$\sum_{i \in N} w_i x_{ij} \leq U \qquad \forall j \in M \tag{3.2}$$

$$\sum_{j \in M} x_{ij} = 1 \qquad \forall i \in N \tag{3.3}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in N, j \in M$$

Constraint 3.2 ensures no backup servers will be overloaded and the other constraints make sure all spot servers are assigned to only one backup server. This ILP can be solved by an off-the-shelf solver like CPLEX. However, this ILP formulation

[2]A backup server is not necessary for running stateless services e.g., a single web server that is part of a tier of replicated web servers, since these services are designed to tolerate failures. However, as with any IaaS platform, SpotCheck does not make any assumptions about applications that run on it, and may incur slightly higher costs than necessary for stateless services.

requires remapping of VMs to backup servers periodically, and is not feasible in the current SpotCheck implementation. We develop an online version of this backup assignment which doesn't require remappings below.

Online greedy backup assignment policy. The round-robin policy discussed earlier does not try to minimize the number of concurrent revocations, and the ILP formulation is an offline approach. We have developed an online policy (called *online-greedy*) which seeks to minimize the number of concurrent revocations, and can be run dynamically as VMs are launched.

The online-greedy policy runs after a VM has been assigned to a server pool i. It places the VM into a backup server which has the least number of VMs from pool i, and which still has capacity available to host one more VM. Since the number of concurrent revocations is simply the number of VMs from the same pool, by picking the backup server with the smallest number of VMs from that pool, the backup servers are not overloaded with VMs from the same pool. Thus, the online-greedy policy seeks to equalize the number of VMs from each pool across all the backup servers.

3.5.3 Reducing Downtime with Hot Spares

When the market price rises above the bid price, the IaaS platform revokes all servers within a pool at the same time, resulting in a revocation storm. A simple approach to handling concurrent revocations is to request an equivalent number of on-demand servers from the IaaS platform and migrate each nested VM to a new on-demand server. However, requesting new servers in a lazy fashion when necessary is only feasible if the latency to obtain them is smaller than the warning period granted to a revoked server. Note that there is never a risk of losing nested VM state, since the backup server stores it even if there is not a destination server available to execute the nested VM. For example, empirical studies have shown that it takes up to 90 seconds to start up a new on-demand server in EC2 [141], while the warning period for a spot server is two minutes, which leaves only 30 seconds to migrate the spot servers state to the new server. If the allocation latency were to exceed the warning time, such a lazy strategy is not possible due to the risk of significant VM downtime. To handle this scenario, SpotCheck proactively acquires a pool of hot spares, servers that are ready to receive nested VMs from revoked spot servers immediately without waiting for a new server to come online. While reducing the risk of downtime, hot spares inevitably increase SpotCheck's overhead cost. Therefore, it is important to only maintain a necessary amount of hot spare servers.

SpotCheck's hot-spare policy seeks to ensure that a small fraction of VMs affected by a revocation storm have a stand-by on-demand server. If the expected maximum number of simultaneous revocations is $E[R_M]$, then we deploy $\beta \cdot E[R_M]$ hot spares. Thus the cost of the hot spares is proportional to the simultaneous revocations, which in turn is a result of pool management and bidding policy. For example, we can set $\beta = 0.1$, which means that 10% of VMs migrating face minimal downtime, while the rest could potentially be affected due to the delay in acquiring on-demand servers from the native IaaS. The hot-spare pool is replenished after the hot spares are used up during migrations.

An alternative approach to using dedicated hot spares is to use existing servers in other stable pools as staging servers. This approach is attractive if these existing servers are not fully utilized by the nested VMs. Here, the staging servers only run the nested VMs from a revoked spot server temporarily, while SpotCheck makes concurrent requests for new on-demand or spot servers to serve as the final destination. This strategy doubles the number of migrations and the associated overhead, but it also enables the system to reduce risk without increasing its costs. Hot spares and staging servers may also serve as a temporary haven for displaced spot VMs, in the rare case when requests for on-demand servers fail because they are unavailable from the IaaS platform[3].

3.5.4 Providing Security Isolation using VPCs

By default, SpotCheck VMs share the cloud servers (using nested virtualization) and the backup servers with different VMs belonging to other customers. This sharing may reduce both the performance and security isolation among VMs. To provide improved isolation, SpotCheck also provides a *Private Cloud* mode, which removes sharing of cloud servers and backup servers between different customers.

In *private cloud* mode, SpotCheck does not place different customers' VMs on the same cloud server, but instead provides a dedicated VPC to each customer to improve network isolation. More importantly, VMs which run in this mode have their own dedicated backup servers. All the bidding, pool management, and other policies are still applicable in this mode, and the VMs among different VPCs do not interact in any way. The key difference is the non-sharing of backup servers, which prevents VMs from one user from interfering with other users' VMs. Multiple VMs sharing a backup server amortizes the backup server cost among them, and in the private cloud mode, the number of VMs run by a customer may not be large enough to completely pack the backup servers. This under-utilization backup servers increases the cost of running in private cloud mode if the number of VMs is small. Thus, the private cloud mode provides increased isolation, at a potentially higher cost, which is a function of the number of VMs that a customer is running in this mode. While we use a relatively large and powerful backup server (`m3.xlarge`) which can service up to 40 VMs, it may be excessive for smaller private clouds. If the number of customer VMs is significantly less than 40, SpotCheck automatically chooses progressively smaller and

[3]IaaS platforms attempt to provision resources to stay ahead of the demand curve, but they may run out of on-demand servers if demand exceeds supply.

cheaper backup servers. For example, 20 VMs can be serviced by the `m3.large` server type, at half the cost of the extra large server. Note that the ratio of computing and storage resources on the backup servers to the multiplexing factor remains the same, and the performance of VMs in the private cloud mode is unaffected. This allows the private cloud mode to be cost feasible even at small sizes.

While the above private cloud mode provides isolation, it can also result in higher costs for customers with low VM requirements. To address this, SpotCheck also offers a *VPC-only* mode, which provides VPCs to customers but shares backup servers among VPCs. The VPCs provide network isolation, and the shared backup servers remove the cost overhead. Thus, foregoing the backup server isolation results in lower costs. Backup servers can be shared among VPCs by EC2's VPC-peering mechanism.

3.5.5 Arbitrage Risks

One caveat in our analysis is that we do not consider the second-order effects of our system on spot prices and availability. While it is certainly possible that widespread use of SpotCheck may perturb the spot market and affect prices, our analysis assumes that the market is large enough to absorb these changes. Regardless, our work demonstrates that a substantial opportunity for arbitrage exists between the spot and on-demand markets. Consumers have a strong incentive to exploit this arbitrage opportunity until it no longer exists. SpotCheck also benefits EC2, since it should raise the demand and price for spot servers by opening them up to a wider range of applications.

The increasing popularity and demand of derivative clouds might also incentivize IaaS platforms to increase their pool of spot servers. However, our analysis assumes that on-demand servers of some type will always be available. While on-demand servers of a particular type may become unavailable, we assume the market is large enough, so that on-demand servers of some type are always available somewhere. As we discuss, SpotCheck's pool management strategies operate across multiple markets by permitting the unrestricted choice of server types and availability zones (within a region). These strategies protect against the rare event where one type of on-demand server becomes unavailable.

Of course, regardless of the risk mitigation strategies above, SpotCheck cannot provide higher availability than the underlying IaaS platform. For example, if the IaaS platform fails or becomes disconnected, as occasionally happens to EC2 [78], SpotCheck would also fail. Since we do not have access to long-term availability data for EC2 or other IaaS platforms, in our experiments, the term "availability" refers to relative availability with respect to the underlying IaaS platform, which we assume is 100% available.

3.6 SpotCheck Implementation

We implemented a prototype of SpotCheck on EC2 that is capable of exercising the different policy options from the previous section, allowing us to experiment with the cost-availability tradeoffs from using different policies. SpotCheck provides a similar interface as EC2 to its customers for managing virtualized cloud servers, although the servers are provisioned in the form of nested VMs.

SpotCheck Controller. The controller, which we implement in `python`, is SpotCheck's main component, and interfaces between customers and the underlying native IaaS platform. The controller is centralized, runs on a dedicated server, and maintains a global and consistent view of SpotCheck's state, e.g., the information about all of its provisioned spot and on-demand servers and all of its customers' nested VMs and their location. While we do not expect the controller's performance to be a bottleneck, if it is, replicating it by partitioning customers across multiple independent controllers is straightforward. In addition, we do not include controller costs in our estimates, since we expect them to be negligible, as they are amortized across all the VMs of all the customers.

Customers interact with SpotCheck's controller via an API that is similar to the management API EC2 provides for controlling VMs. Internally, the controller uses the EC2 REST APIs to issue requests to EC2 and manage its server pools. The controller monitors SpotCheck's state by tracking the cloud server each nested VM runs on, the IP address associated with the nested VM, and the customer's access credentials, and stores this information in a database.

The controller also implements the various pool management strategies from the previous section, e.g., by determining the bids for spot instances and triggering nested VM migrations from one server pool to another. Finally, the controller monitors the load of nested VMs, the mapping of nested VMs to backup servers, and the current spot price in each spot pool. Our prototype implementation uses the XenBlanket [214] nested hypervisor running on a modified version of Xen 4.1.1. The driver domain (`dom-0`) runs Linux 3.1.2 with modifications for supporting XenBlanket. XenBlanket is compatible with all EC2 instance types that support hardware virtual machines (HVM). SpotCheck assumes that the customer-provided disk image used to boot the nested VM resides on a network-attached disk volume in EBS. Due to the use of Xen as the nested hypervisor, the image must support Xen's paravirtualization extensions.

Since network-attached storage is the primary storage medium in many IaaS platforms, including EC2, our current prototype requires the VM to use one (or more) network-attached EBS volumes to store the root disk and any persistent state, and does not support backing up local storage to a remote disk. However, since the speed of the local disk and a backup server's disk are similar in magnitude, EC2's warning period permits asynchronous mirroring of local disk state to the backup server, e.g., using DRBD [90], without significant performance degradation. Our experiments primarily focus on memory-intensive workloads, since using a backup server to store the live in-memory state of multiple nested VMs imposes a significantly larger cost and performance overhead than maintaining disk backups.

To implement SpotCheck, we modified XenBlanket to support bounded-time VM migration in addition to live migration. For the former, we adapt a version of the bounded-time VM migration technique implemented in Yank [189] for use with nested virtualization and implement additional optimizations to reduce downtime during migration. In particular, the continuous checkpoints due to bounded-time VM migration guarantee that during the last checkpoint the nested VM is able to transfer the stale state within the warning time. In Yank [189], the VM pauses execution and incurs downtime when transferring the stale state after receiving a warning. To reduce this downtime, our implementation increases the checkpointing frequency after receiving a warning, which reduces the amount of dirty pages the nested VM must transfer. By gradually increasing the checkpointing frequency, we reduce downtime at the cost of slightly degrading VM performance during the warning period.

SpotCheck configures nested VMs mapped to a spot server pool to use bounded-time VM migration, while it configures those mapped to an on-demand pool to use live migration. Nested VMs mapped to a spot server pool are also mapped to a backup server, which must process a write-intensive workload during normal operation and must process a workload that includes a mix of reads and writes during revocation events, e.g., to read the memory state of a revoked nested VM and migrate it. As a result, we optimize each backup server's file system and kernel memory management options for write-heavy traffic. Specifically, we use the ext4 filesystem, and avoid costly metadata updates by using the write-back journalling mode and the `noatime` option. This is safe, since the backup server stores a small number of large files, representing the memory state of each nested VM it backs up, with no read/write concurrency, i.e., the files storing VM memory state are either being written or read but not both. To maximize the use of the page cache and absorb write storms, we set a high `dirty_ratio` and `dirty_background_ratio`, which retains file data in the page cache for a long period, allowing the I/O scheduler to increase batching of write requests.

During revocations, the backup server prepares for nested VM restoration by loading images into memory using `fadvise`, setting the `WILL_NEED` flag, and using the appropriate `RANDOM` or `SEQUENTIAL` access flags, depending on whether SpotCheck is lazily restoring the VM or not. In addition, we also implement bandwidth throttling using `tc` on a per-VM basis to limit the network bandwidth used for each migration/restoration operation, and to avoid affecting nested VMs that are not migrating. Thus, we optimize our backup server implementation for the common case of efficiently handling a large number of concurrent revocations without degrading performance for long durations. Our SpotCheck prototype uses the `m3.xlarge` type as backup servers, since they currently offer the best price/performance ratio for our workload mix. Our prototype uses a combination of SSDs and EBS volumes to store the memory images.

Lazy restoration requires transferring the "skeleton" state of a VM, comprising the vCPU state, all associated VM page tables, and other hardware state maintained by the hypervisor, to the destination host and immediately beginning execution. This skeleton state is small, typically around 5MB, and is dominated by the the size of the page tables. The skeleton state represents the minimum amount of state sufficient for the hypervisor on the destination host to create the domain for VM and begin executing instructions. To allow the hypervisor to trap accesses to missing memory pages during execution, our implementation of lazy restoration enables shadow paging during the restore process. As a result, the missing memory pages, which reside on the backup server's disk, are mapped to the domain's memory when available and the VM resumes execution. A background process concurrently reads all other unrestored pages without waiting for them to be paged in by the executing VM.

We conducted extensive measurements on EC2 to profile the latency of SpotCheck's various operations. Table 3.1 shows the results for one particular server type, the `m3.medium`. We conducted these experiments when there was no explicit documentation of a revocation warning for spot servers on EC2. Our measurements found that, at that the time, EC2 provided an opportunity to gracefully shutdown the VM, by issuing a shutdown command, before forcibly terminating the VM two minutes after issuing the shutdown. Thus, we replaced the default shutdown script with our own script, which EC2 would invoke upon revocation to notify SpotCheck of the two minute warning. However, as we mention previously, as of January 2015 [56], EC2 now provides an explicit two minute notification of shutdown through the EC2 management interface.

When employed natively our live bounded-time VM migration incurs a brief millisecond-scale downtime similar to that of a post-copy live migration. However, Table 3.1 shows that EC2's operations also contribute to downtime. In particular, SpotCheck can only detach a VM's EBS volumes and its network interface after the VM is paused, and it can only reattach them after the VM is resumed. From Table 3.1, these operations (in bold) cause an average downtime of 22.65 seconds. While significant, this downtime is not fundamental to SpotCheck: EC2 and other IaaS platforms could likely significantly reduce the latency of these operations, which would further improve the performance and availability we report in Section 6. Even now, this $\sim$23 second downtime is not long enough to break TCP connections, which generally requires a timeout of greater than one minute.

Finally, SpotCheck's implementation builds on our prior work on Yank [189] by including the performance optimizations above. In particular, these optimizations enable i) SpotCheck's backup servers to support a much larger number of VMs and ii) lazy on-demand fetching of VM memory pages to drastically reduce restoration

	Median(sec)	Mean(sec)	Max(sec)	Min(sec)
Start spot instance	227	224	409	100
Start on-demand instance	61	62	86	47
Terminate instance	135	136	147	133
Unmount and detach EBS	10.3	10.3	11.3	9.6
Attach and mount EBS	5	5.1	9.3	4.4
Attach Network interface	3	3.75	14	1
Detach Network interface	2	3.5	12	1

Table 3.1: Latency for various SpotCheck operations on EC2 for the `m3.medium` server type based on 20 separate measurements executed over a one week period.

time, e.g., to <0.1 seconds. We quantify the impact of these optimizations on cost, performance, and availability in the next section.

3.7 SpotCheck Evaluation

Our evaluation consists of a mix of end-to-end experiments and simulations. For our end-to-end experiments, we quantify SpotCheck's performance under different scenarios using a combination of EC2 servers and our own local servers. For our simulations, we combine performance measurements from our end-to-end experiments with historical spot pricing data on EC2 to estimate SpotCheck's cost savings and availability at scale over a long period. As mentioned previously, SpotCheck uses Virtual Private Clouds (VPCs) in EC2 to create and assign IP addresses to nested VMs. We run all the microbenchmark experiments in a single EC2 availability zone, while our simulations include cross-availability zone experiments within a single region. Since XenBlanket is only compatible with servers that have HVM capabilities, SpotCheck is only capable of using HVM-enabled EC2 servers. Thus, for our experiments, we primarily use `m3.*` server types. In particular, we use `m3.xlarge` server types for our backup servers, and, by default, host nested VMs on `m3.medium` server types. The `m3.medium` is the smallest HVM-enabled server. We evaluate SpotCheck using two well-known benchmarks for interactive multi-tier web applications: TPC-W [11] and SPECjbb2005 [10]. We are primarily interested in memory-intensive workloads, since the continuous checkpointing of memory pages imposes the most performance overhead for these workloads.

TPC-W simulates an interactive web application. We use Apache Tomcat (v6.26) as the application server and MySQL (v5.0.96) as the database. We configure clients to perform the "ordering workload" in our experiments.

SPECjbb is a server-side benchmark that is generally more memory-intensive than TPC-W. The benchmark emulates a three-tier web application, and particularly stresses the middle application server tier when executing the test suite.

All nested VMs run the same benchmark with the same 30 second time bound for bounded-time migration, which we choose conservatively to be significantly lower than the two minute warning provided by EC2. Thus, our cost and availability results are worse than possible if using a more liberal time bound closer to the two minute warning time. In our experiments, we compare SpotCheck against i) Xen's pre-copy live migration, ii) an unoptimized bounded-time VM migration that fully restores a nested VM before starting it (akin to Yank [189]), (iii) SpotCheck's optimized Full restore, iv) an unoptimized bounded-time VM migration that uses lazy restoration, and finally v) SpotCheck's optimized bounded-time VM migration with lazy restoration.

3.7.1 End-to-End Experiments

SpotCheck uses a backup server to checkpoint VM state and support bounded-time VM migration. SpotCheck's cost overhead is primarily a function of the number of VMs each backup server multiplexes: the more VMs it multiplexes on a backup server, the lower its cost (see Section 3.4.4). Figure 3.7 shows the effect on nested VM performance for SpecJBB and TPC-W as the load on the backup server increases.

First, we evaluate the overhead of continuously checkpointing memory and sending it over the network to the backup server. The "0" and "1" columns in Figure 3.7

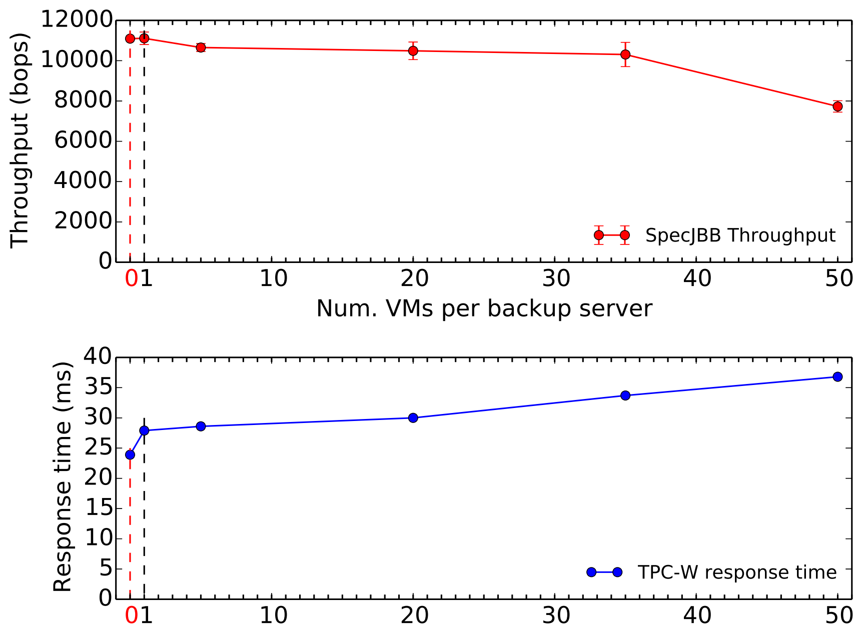


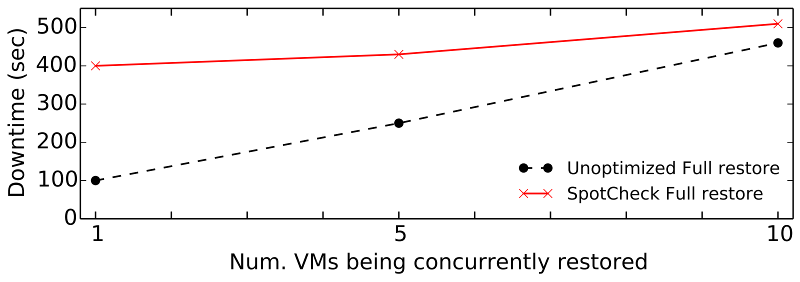
Figure 3.7: Effect on performance as the number of nested VMs backing up to a single backup server increases.

represent performance difference between no checkpointing and checkpointing using a dedicated backup server, respectively. By simply turning checkpointing on and using a dedicated backup server, we see that TPC-W experiences a 15% increase in response time, while SpecJBB experiences no noticeable performance degradation during normal operation. With an increasing number of nested VMs all backing up to a single server, saturation of the disk and network bandwidth on the backup server leads to a decrease in nested VM performance after 35 VMs, where SpecJBB throughput decreases and TPC-W response time increases significantly, e.g., by roughly 30% each. Note that the nested VM incurs this performance degradation as long as it is running on a spot server. Thus, to ensure minimal performance degradation during normal operation, SpotCheck assigns at most 35-40 VMs per backup server. As a result, SpotCheck's cost overhead for backing up each nested VM is roughly $1/40 = 2.5\%$ of the price of a backup server. For our `m3.xlarge` backup server, which costs $0.28 per hour in the

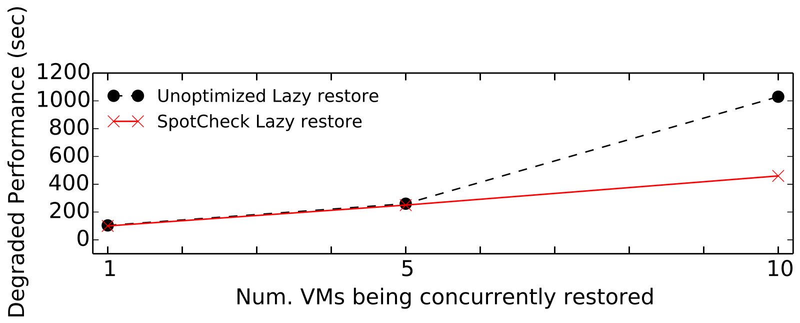
East region of EC2, the amortized cost per-VM across 40 nested VMs is $0.007 or less than one cent per VM.

In addition to performance during normal operation, spot server revocations and the resulting nested VM migrations and restorations impose additional load on the backup server. Figure 3.8 shows the length of the period of downtime or performance degradation when migrating nested VMs via the backup server. In this case, we compare migrations that utilize lazy restoration with those that use a simple stop-and-copy migration. A stop-and-copy approach results in high *downtime*, whereas a lazy restore approach results in much less downtime but some *performance degradation* when memory pages must be fetched on-demand across the network on their first access. Since lazy restore incurs less downtime, it reduces the effect of migrations on interactive applications. Figure 3.8 shows that when concurrently restoring 1 and 5 nested VMs the time required to complete the migration is similar for both lazy restoration and stop-and-copy migration, which results in performance degradation or downtime, respectively, over the time window.

However, when executing 10 concurrent restorations, the length of the lazy restoration is much longer than that of the stop-and-copy migration. This occurs because lazy restoration uses random reads that benefit less from prefetching and caching optimizations than a stop-and-copy migration, which uses sequential reads. This motivates SpotCheck's lazy restoration optimization that uses the `fadvise` system call to inform the kernel how SpotCheck will use the VM memory images stored on disk, e.g., to expect references in random order in the near future. The optimization results in a significant decrease in the restoration time for lazy restore. Thus, SpotCheck's optimizations significantly reduce the length of the period of performance degradation during lazy restorations. Of course, SpotCheck also assigns VMs to backup servers to reduce the number of revocation storms that cause concurrent migrations. We evaluate SpotCheck's bidding and pool assignment policies below.



(a) Duration of downtime with Full restore



(b) Duration of degraded performance with Lazy restore

Figure 3.8: Duration of downtime during a traditional VM restore, and performance degradation during a lazy restore.

Finally, in addition to the time to complete a migration, SpotCheck also attempts to mitigate the magnitude of performance degradation during a migration and lazy VM restoration. During the lazy restoration phase the VM experiences some performance degradation, which may impact latency-sensitive applications, such as TPC-W. Since the first access to each page results in a fault that must be serviced over the network, lazy restoration may cause a temporary increase in application response time. Figure 3.9 shows TPC-W's average response time as a function of the number of nested VMs being concurrently restored, where zero represents normal operation. The graph shows that when restoring a single VM the response time increases from 29ms to 60ms for the period of the restoration. Additional concurrent restorations do not significantly degrade performance, since SpotCheck partitions the available bandwidth equally among nested VMs to ensure restoring one VM does not negatively affect the performance of VMs using the same backup server.

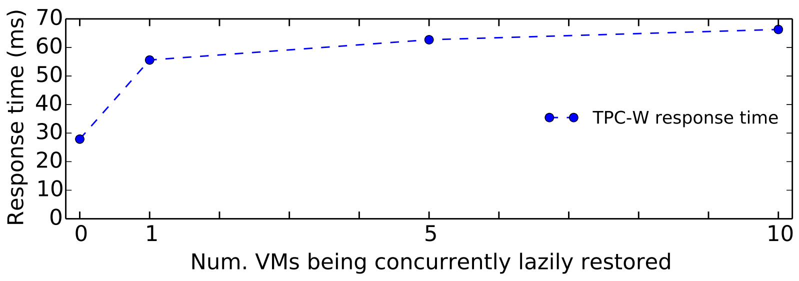


Figure 3.9: Effect of lazy restoration on VM performance.

Policy	Description
1P-M	VMs mapped to a single m3.medium pool
2P-ML	VMs equally distributed between two pools : `m3.medium` and `m3.large`.
4P-ED	VMs equally distributed to four pools consisting of four `m3` server types
4P-COST	VMs distributed based on past prices. The lower the cost of the pool over a period, the higher the probability of mapping a VM into that pool.
4P-ST	VMs distributed based on number of past migrations. The fewer the number of migrations over a period, the higher the probability of mapping a VM into that pool.

Table 3.2: SpotCheck's customer-to-pool mapping policies.

Note that SpotCheck's policies attempt to minimize the number of evictions and migrations via pool management, and thus the performance degradation of applications *during the migration process* is a rare event. Even so, our evaluation above shows that application performance is not adversely affected even when the policies cannot prevent migrations.

Result: *SpotCheck executes nested VMs with little performance degradation and cost overhead during normal operation using a high VM-to-backup ratio and migrates/restores them with only a brief period of performance degradation.*

3.7.2 SpotCheck Policies and Cost Analysis

As we discuss in Section 4, SpotCheck may employ a variety of bidding and VM assignment policies that tradeoff performance and risk. Here, we evaluate SpotCheck's cost using various bidding policies based on the EC2 spot price history from April 2014 to October 2014. In particular, Table 3.2 describes the policies we use to assign VMs to spot pools. The simplest policy is to place all VMs on servers from a single spot market (1P-M); this policy minimizes costs if SpotCheck selects the lowest price pool, but increases risk, since it may need to concurrently migrate all VMs if a price spike occurs. We examine two policies (2P-ML and 4P-ED) that distribute VMs across servers from different spot markets to reduce risk, albeit at a potentially higher cost. We also examine two policies (4P-COST and 4P-ST) that probabilistically select pools based on either their weighted historical (rather than current) cost or their weighted historical price volatility. The former lowers cost, while the latter reduces performance degradation from frequent migrations.

Figure 3.10 shows SpotCheck's average cost per hour when using each policy. As expected, the average cost for running a nested VM using live migration, i.e., without a backup server, is less than the average cost using SpotCheck, since live migration does not require a backup server. Of course, using only live migration is not practical, since, without a backup server, SpotCheck risks losing VMs before a live migration completes. In this case, 1P-M has the lowest average cost, since SpotCheck maps VMs to the lowest priced spot pool. Distributing VMs across two (2P-ML) and then four (4P-ED) pools marginally increases costs. The two policies that probabilistically select pools based on either their historical cost or volatility have roughly the same cost as the policy that distributes across all pools. Note that the average cost SpotCheck incurs for the equivalent of an `m3.medium` server type is ∼\$0.015 per hour, while the cost of an `m3.medium` on-demand server type is \$0.07, or a savings of nearly 5×.

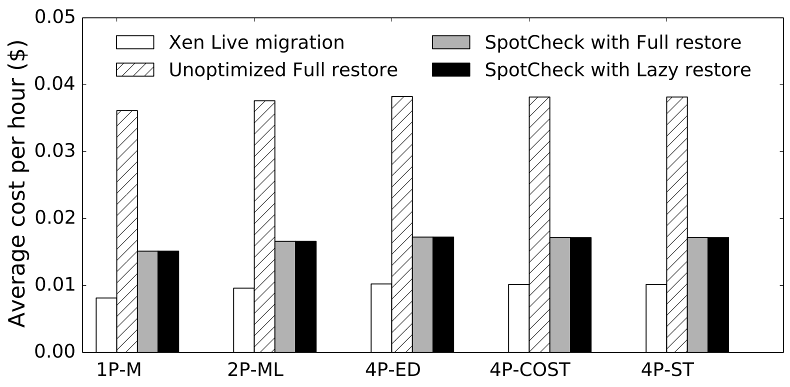


Figure 3.10: Average cost per VM under various policies.

The cost of SpotCheck VMs also depends on the utilization of the backup servers, since the backup server costs are shared by all the VMs. Due to the dynamic arrival and lifetimes of VMs, SpotCheck's *online* backup server policy may leave backup servers under-utilized, and thus increase effective costs. We evaluate the costs of backup servers using the Eucalyptus cloud workload trace [23]. Figure 3.11 shows the backup server costs of SpotCheck's backup-server allocation policy relative to the optimal bin-packing policy which minimizes the number of backup servers. We can see from Figure 3.11 that the increase in backup-server costs (compared to the optimal) ranges from 2% to 65% (for the short trace #5). This translates to a per-VM cost increase of 1-33% compared to the full utilization scenario. Taking under-utilization of backup servers into account, the worst-case cost savings for SpotCheck is still more than 2× compared to the on-demand instances.

While reducing cost is important, maximizing nested VM availability and performance by minimizing the number of migrations is also important. Here, we evaluate the unavailability of VMs due to spot server revocations. For these experiments, we assume a period of performance degradation due to detaching and reattaching EBS volumes, network reconfiguration, and migration. We seed our simulation with measurements from Table 3.1 and the microbenchmarks from the previous section. In particular, we assume a downtime of 23 seconds per migration due to the latency of

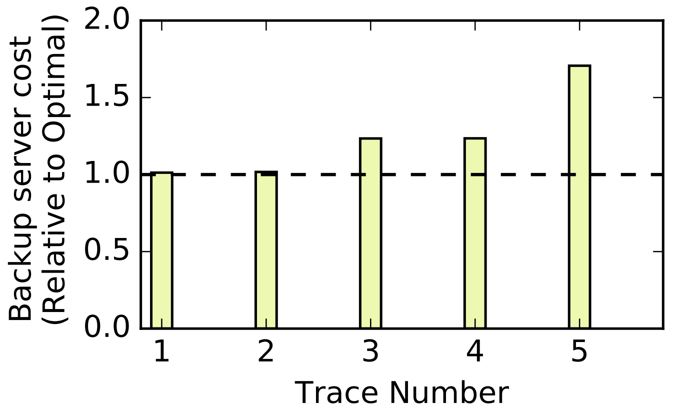


Figure 3.11: Cost of backup servers relative to optimal packing for the Eucalyptus cloud traces.

EC2 operations. Based on these values and the spot price history, Figure 3.12 shows nested VM unavailability as a percentage over the six month period from April to October for each of our policies. As above, we see that live migration has the lowest unavailability, since it incurs almost no downtime, but is not practical, since it risks losing VM state. We also examine both an unoptimized version of bounded-time VM migration requiring a full restoration before resuming (akin to Yank) and our optimized version that also requires a full restoration. The graph demonstrates that the optimizations in Section 5 increase the availability. The graph also shows that, even without lazy restoration, SpotCheck's unavailability is below 0.25% in all cases, or an availability of 99.75%.

However, we see that using lazy restore brings SpotCheck's unavailability close to that of live migration. Since the `m3.medium` spot prices over our six month period are highly stable, the 1P-M policy results in the highest availability of 99.9989%, as well as the lowest cost from above. This level of availability is roughly 10× that of directly using spot servers, which, as Figure 3.6(a) shows, have an availability between 90% and 99%. The other policies exhibit slightly lower availability ranging from 99.91% for 2P-ML to 99.8% for 4P-ED. In addition to availability, performance degradation is also important. Figure 3.13 plots the percentage of time over the six month period a nested

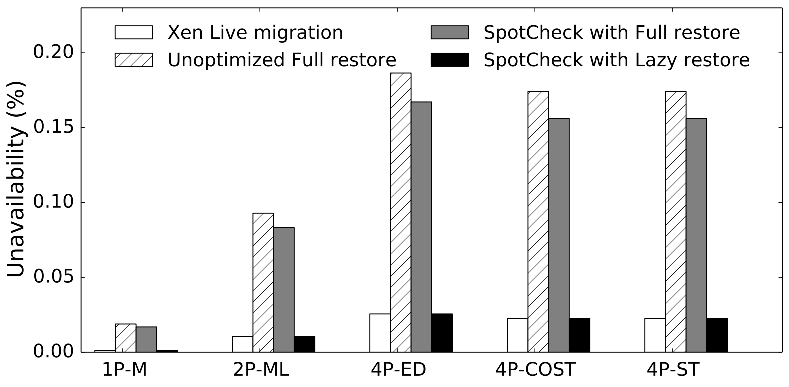


Figure 3.12: Unavailability for live migration and SpotCheck (with and without optimizations and lazy restore).

VM experiences performance degradation due to a migration and restoration. The graph shows that, while SpotCheck with lazy restoration has the most availability, it has the longest period of performance degradation. However, for the single pool 1P-M policy, the percentage of time the nested VM operates under degraded performance is only 0.02%, while the maximum length of performance degradation (for 4P-ED) is only 0.25%. For perspective, over a six month period, SpotCheck using the 1P-M policy has only 2.85 combined minutes of degraded performance due to migrations and restorations.

Result: *SpotCheck achieves nearly 5× savings compared to using an equivalent on-demand server from an IaaS platform, while providing 99.9989% availability with migration-related performance degradation only* 0.02% *of the time.*

The cost-risk tradeoff between choosing a *single pool* versus *two pools* versus *four pools* is not obvious. While, in the experiments above, 1P-M provides the lowest cost and the highest availability, the risk of SpotCheck having to concurrently migrate all nested VMs at one time is high, since all VMs mapped to a backup server are from a single pool. For the six month period we chose, the spot price in the `m3.medium` pool rarely rises above the on-demand price, which triggers the migrations and accounts for its high availability. The other policies mitigate this risk by increasing the number of

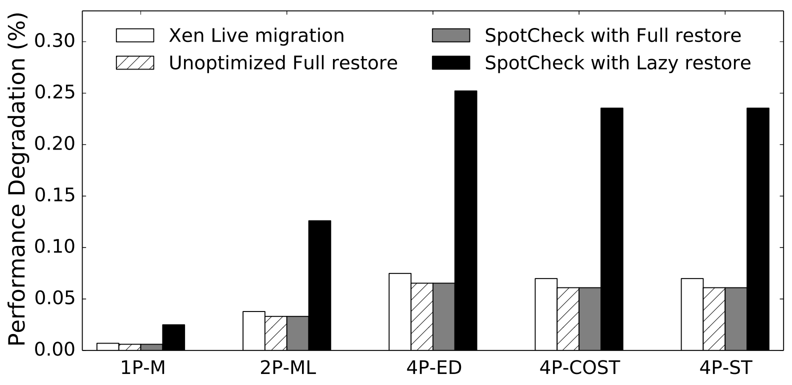


Figure 3.13: Performance degradation during migration.

pools by distributing the VMs across these pools. Since the price spikes in these pools are not correlated, the risk of losing all VMs at once is much lower. Table 3.5 shows the probability of concurrent revocations of various sizes as a factor of the total number of VMs N. We note that the probability of all N VMs migrating in a single pool scenario is higher compared to the two-pool scenario and nearly non-existent in the case of the four-pool policy. Also, by distributing VMs across pools, SpotCheck increases the overall frequency of migration, but reduces the number of mass migrations.

Result: *Distributing nested VMs mapped to each backup server across pools enables SpotCheck to lower the risk of large concurrent migrations. For example, comparing 1P-M to 4P-ED, the average VM cost in 4P-ED increases by \$0.002 and the availability reduces by 0.15%, but the approach avoids all mass revocations.*

Policy Comparison. Our results demonstrate that each of SpotCheck's policies provide similar cost savings (Figure 3.10) and availability (Figure 3.12). Performance degradation is lowest for single-pool policy (1P-M), but negligible even for the worst-performing policy (4P-ED as shown in Figure 3.13), while the four-pool policies drastically reduce the risk of mass migration events (from Table 3.5).

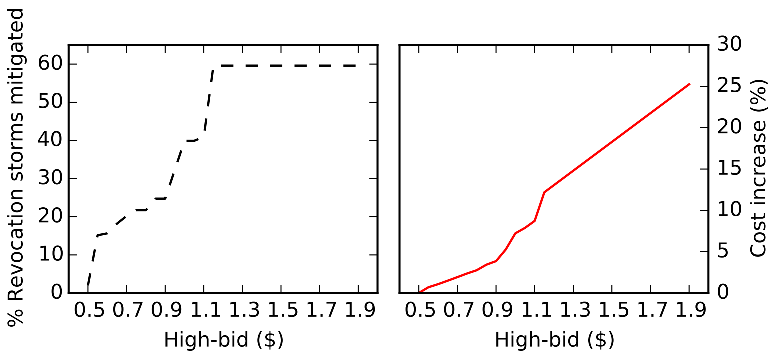


Figure 3.14: Two-level bidding reduces impact of revocation storms because only half the servers are affected. As the high-bid is increased, the percentage of storms mitigated increases upto a limit, and so does the cost (compared to single-level bidding).

3.7.3 Comparison of Risk Mitigation Policies

Two-level Bidding: To evaluate the impact of two-level bidding, we use pick two bids and observe the impact on the revocations and the expected cost. For two-level bidding, the low-bid is the on-demand price. Thus we keep the low-bid equal to the on-demand price and vary the high-bid. We are interested in comparing with the single-level bidding policy, and keep all parameters such as the workload and other policies constant. The impact of two-level bidding is shown in Figure 3.14, which shows the decrease in revocation storms and increase in cost vs. the high-bid. As the high-bid increases, the fraction of revocation storms which are mitigated (only low-bid servers affected) increases upto a limit, after which it starts to flatten out. Correspondingly, the cost also increases because of the higher bids. For the `m1.2xlarge` instance, the two-level bidding strategy can mitigate almost 60% of revocation storms with a 20% increase in cost. That is, whenever a revocation event occurs, it will only affect half of the servers 60% of the time. Thus, two-level bidding can be an effective strategy to increase the number of effective pools and mitigate revocation storms.

Backup Selection: The backup server selection policies determine the load on the backup server during revocations, for which we measure the number of concurrent revocations faced by each backup server. During a revocation, the backup server

Num. Spot Pools	Reduction in concurrent revocations
1 Pool	2%
2 Pool	13.7%
4 Pool	18.3%

Table 3.3: Reduction in max number of concurrent revocations with the online greedy backup server selection, when compared to round-robin.

is faced with increased checkpointing frequency and must provide pages to the lazy restoration process. An overloaded backup server servicing a large number of lazy restorations is detrimental to smooth migrations. Accordingly, we compare the different backup selection policies in terms of the number of concurrent revocations in Table 3.3. For different pool management policies, the impact of backup selection varies, because the number of pools determines the "spread" of VMs. We compare the online-greedy policy with the default round-robin policy. When using a single pool, there is a slight reduction in the number of concurrent revocations with the online-greedy policy (2%), whereas the reduction is 18% with 4 pools. Thus, the online-greedy backup selection policy reduces the concurrent revocation load on the backup servers.

Hot spares: Hot spares are readily available, already running on-demand servers used to migrate VMs upon a revocation. Hot spares reduce the downtime during migration, but incur an additional cost, which is shown in Table 3.4. For different policies, the number of simultaneous revocations (size of revocation storm) affects the hot spare cost, and is lower when the number of markets is larger, and also when two-level bidding is employed. With a single pool, having 10% servers as hot spares results in a 50% increase in expected cost, whereas the overhead of hot spares is only 15% when using 2 pools and 2-level bidding.

VPC: The cost of running in the private cloud mode is higher than the default shared mode of operation because the backup server cost is not shared by a larger number of VMs. The cost of running VMs in the private cloud mode is shown in Figure 3.15. As the private cluster size increases, the cost decreases because the multiplexing of

Policy	Price Increase
1 Pool	50%
2 Pool	25%
1 Pool 2-level bidding	30%
2 Pool 2-level bidding	15%

Table 3.4: Percentage increase in cost due to hot spares

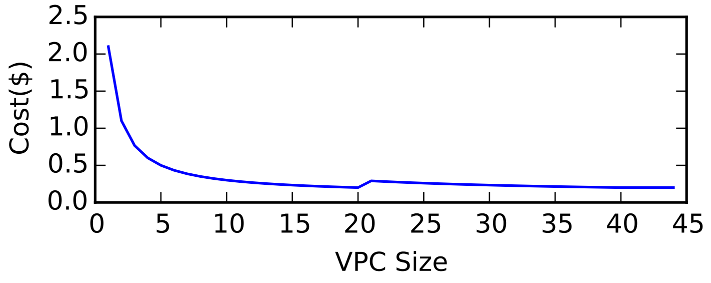


Figure 3.15: Cost of running in private cloud mode for different sizes.

backup servers increases. SpotCheck is able to select smaller backup servers for smaller number of VMs, and cost of running 5 VMs is 40% higher per VM when compared to the default shared-everything mode.

3.8 Related Work

Designing Derivative Clouds. Prior work on interclouds [63] and superclouds [213, 133] propose managing resources across multiple IaaS platforms by using nested virtualization [214, 60, 233] to provide a common homogeneous platform. While SpotCheck also leverages nested virtualization, it focuses on exploiting it to transparently reduce the cost and manage the risk of using revocable spot servers on behalf of a large customer base. Our current prototype does not support storage migration or inter-cloud operation; these functions are the subject of future work. Cloud Service Brokers [164], such as RightScale [15], offer tools that aid users in aggregating and integrating resources from multiple IaaS platforms, but without abstracting the underlying resources like SpotCheck. PiCloud [14] abstracts spot and

	Max. num. of concurrent revocations			
	N/4	N/2	3N/4	N
1-Pool	0	0	0	1.74×10^{-4}
2-Pool	0	3.75×10^{-3}	0	2.25×10^{-5}
4-Pool	7.4×10^{-3}	7.71×10^{-5}	1.92×10^{-5}	0

Table 3.5: Probability of the maximum number of concurrent revocations for different pools. N is the number of VMs.

on-demand servers rented from IaaS platforms by exposing an interface to consumers that allows them to submit batch jobs. Derivative clouds can also offer containers instead of nested VMs, and [151, 165] look at the problem of resource management in this context. In contrast, SpotCheck provides the abstraction of a complete IaaS platform that supports any application. Finally, SpotCheck builds on a long history of research in market-based resource allocation [64], which envisions systems with a fluid mapping of software to hardware that enable computation and data to flow wherever prices are lowest.

Spot Market Bidding Policies. Prior work on optimizing bidding policies for EC2 spot instances are either based on analyses of spot price history [119, 219, 61] or include varying assumptions about application workload, e.g., job lengths, deadlines [231, 194, 227, 198, 144], which primarily focus on batch applications. By contrast, SpotCheck's bidding strategy focuses on reducing the probability of mass revocations due to spot price spikes, which, as we discuss, may significantly degrade nested VM performance in SpotCheck.

Virtualization Mechanisms. Prior work handles the sudden revocation of spot servers either by checkpointing application state at coarse intervals [203, 125, 225] or eliminating the use of local storage [75, 136]. In some cases, application modifications are necessary to eliminate the use of local storage for storing intermediate state, e.g., MapReduce [75, 136]. SpotCheck adapts a recently proposed bounded-time VM migration mechanism [189, 190],which is based on Remus [81] and similar to microcheckpointing [9], to aggressively checkpoint memory state and migrate nested

VMs away from spot servers upon revocation. Our lazy restore technique is similar to migration mechanisms, such as post-copy live migration [114] and SnowFlock [132].

3.9 SpotCheck Summary

SpotCheck is a derivative IaaS cloud that offers low-cost, high-availability servers using cheap but volatile servers from a native IaaS platforms. To do this, SpotCheck must simultaneously ensure high availability, reduce the risk of mass server revocations, maintain high performance for applications, and keep its costs down. We design SpotCheck to balance these competing goals. By combining recently proposed virtualization techniques, SpotCheck is able to provide more than four 9's availability to its customers, which is more than 10× that provided by the native spot servers. At the same time, SpotCheck's VMs cost nearly 5× less than the equivalent on-demand servers

CHAPTER 4

BATCH-INTERACTIVE DATA-INTENSIVE PROCESSING ON TRANSIENT SERVERS

The previous chapter tackled the problem of running interactive applications using nested virtualization and bounded-time live migration, resulting in low application downtime.

In this chapter, we look at the problem of running distributed data processing workloads on transient servers with low performance overheads and cost. In particular, we look at an emerging class of workloads, which we call Batch-Interactive Data-Intensive (BIDI), that are becoming increasingly important for data analytics. BIDI workloads require large sets of servers to cache massive datasets in memory to enable low latency operation. We illustrate the challenges of executing BIDI workloads on transient servers, and address them in our system called Flint, which includes automated checkpointing and server selection policies.

We evaluate a prototype of Flint using EC2 spot instances, and show that it yields cost savings of up to 90% compared to using on-demand servers, while increasing running time by $< 2\%$.

4.1 Motivation

The distributed data-parallel processing frameworks, such as MapReduce [83], that now dominate cloud platforms, have historically executed their workload as non-interactive batch jobs. Since these frameworks were intended to operate at large scales, they were also designed from the outset to handle server failures by replicating their input and output data in a distributed file system. As a result, they required few modifications to run efficiently on transient servers [75, 136], where revocations are akin to failures. However, recently, there has been an increasing interest in better supporting interactivity in data-parallel frameworks. Interactivity enables data exploration, stream processing, and data visualization through ad-hoc queries. These new *batch-interactive* frameworks, including Spark [230] and Naiad [154], execute both batch and interactive applications and effectively enable a new class of workload, which we call *Batch-Interactive Data-Intensive* (BIDI).

BIDI workloads differ from the Online Data-Intensive (OLDI) workloads [146] processed by web applications in that the magnitude and variance of their acceptable response latency is much larger. For example, to avoid frustrating users, web applications often target strict latency bounds for rendering and serving each web page, typically on the order of 100 milliseconds with low variance. In contrast, users interactively executing a BIDI workload often have much more relaxed latency expectations, in part, because the amount of data each operation acts on (and the time it takes to complete) varies widely. Thus, users may expect query latency to vary anywhere between a few seconds to a few minutes. We argue that BIDI workloads' relaxed performance requirements still make them amenable to transient servers. Further, the low price of transient servers is particularly attractive to these new frameworks, since they require large sets of servers to cache massive datasets in memory.

Applications can employ fault-tolerance mechanisms, such as checkpointing and replication, to mitigate the impact of server revocations without rerunning the application. Checkpointing intermediate state enables restarting an application on a new server, and requires only partial recomputation from the last checkpoint. Of course, each checkpoint introduces an overhead proportional to the size of the local disk and memory state. Likewise, replicating the computation across multiple transient servers enables the application to continue execution if a subset of servers are revoked. However, replication is only feasible if the cost of renting multiple transient servers is less than the cost of an on-demand server. Prior work has only applied such fault-tolerance mechanisms at the systems level, e.g., using virtual machines (VMs) or containers [197]. While a systems-level approach is transparent to applications, we argue that an *application-aware* approach is preferable for distributed BIDI workloads, as it can i) improve efficiency by adapting the fault-tolerance policy, e.g., the checkpoint frequency and the subset of state to checkpoint, to each application's characteristics and ii) avoid implementing complex distributed snapshotting [72] schemes.

Since BIDI workloads support interactivity and low latency by caching large datasets in memory, revocations may result in a significant loss of volatile in-memory state. To handle such losses, batch-interactive frameworks natively embed fault-tolerance mechanisms into their programming model. For example, Naiad periodically checkpoints the in-memory state of each vertex, and automatically restores from these checkpoints on failure [154]. Similarly, Spark enables programmers to explicitly checkpoint distributed in-memory datasets—if no checkpoints exist, Spark automatically recomputes in-memory data lost due to server failures from its source data on disk [230]. Importantly, since failures are rare, these systems do not exercise sophisticated control over these fault-tolerance mechanisms. However, an application-aware approach can leverage these existing mechanisms to implement automated policies to optimize BIDI workloads for transient servers.

Since cloud providers offer many different types of transient servers with different price and availability characteristics, selecting the set of transient servers that best balances the per unit-time price of resources, the risk of revocation, and the overhead of fault-tolerance presents a complex problem. To address the problem, we design Flint, a batch-interactive framework based on Spark tailored to run on, and exploits the characteristics of, transient servers. Specifically, Flint includes automated fault-tolerance and server selection policies to optimize the cost and performance of executing

BIDI workloads on transient servers. Our hypothesis is that Flint's application-level approach can significantly decrease the cost of running Spark programs by using transient servers efficiently to maintain high performance—near that of using on-demand servers. In evaluating our hypothesis, we make the following contributions:

Checkpointing Policies. Flint defines automated checkpointing policies to bound the time spent recomputing lost in-memory data after a revocation. Flint extends prior work on optimal checkpointing for single node batch jobs in the presence of failures to a BIDI programming model that decomposes program actions into collections of fine-grained parallel tasks. Flint dynamically adapts its checkpointing policy based on transient server characteristics and the characteristics of each distributed in-memory dataset.

Transient Server Selection Policies. Flint defines server selection policies for batch and interactive workloads. For batch workloads, the policy selects transient servers to minimize expected running time and cost, while considering both the current price of resources and their probability of revocation. In contrast, for interactive workloads, the policy selects transient servers to provide more consistent performance by reducing the likelihood of excessively long running times that frustrate users (for a small increase in cost).

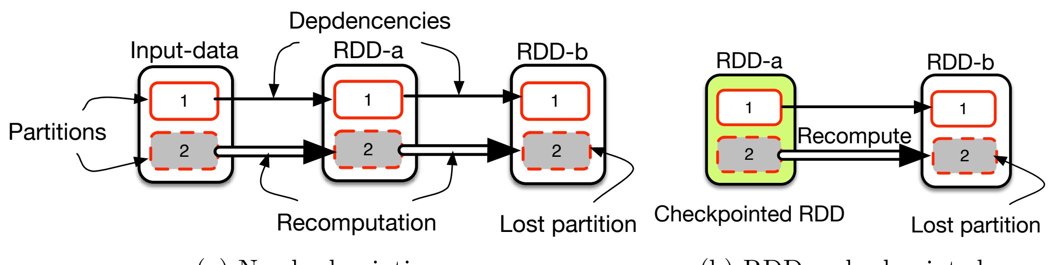
Implementation and Evaluation. We implement Flint on top of Spark and Mesos, and deploy it on spot instances on EC2. We evaluate its cost and performance benefits for multiple BIDI-style workloads relative to running unmodified Spark on on-demand and spot instances using existing systems-level checkpointing and server selection policies. Our results show that compared to unmodified Spark, Flint yields cost savings of up to 90% compared to on-demand instances and 50% when compared to spot instances, while increasing running time by $< 2\%$. For interactive workloads, Flint achieves $10\times$ lower response times when compared to running unmodified Spark on spot instances.

4.2 Spark Background

We use Spark [230] as a representative distributed data-parallel data processing framework. Spark is a general-purpose data-parallel processing engine that supports a rich set of data transformation primitives. Spark supports both long-running "big-data" batch jobs, as well as interactive data processing. Interactive jobs may come in several varieties. For example, users can used a Read-Eval-Print-Loop (REPL) for interactive and exploratory analysis. As another example, Spark can be employed as a database engine with SQL queries executed via a translation layer such as Spark-SQL [51]. Spark's growing popularity is due to its performance and scalability, as well as the ease with which many tasks can be implemented as Spark programs. For example, Spark supports batch and MapReduce jobs, streaming jobs [229], SQL queries [51], graph processing [218], and machine learning [147] tasks on a single platform with high performance.

Spark *programs* access APIs that operate on and control special distributed in-memory datasets called Resilient Distributed Datasets (RDDs). Spark divides an RDD into *partitions*, which are stored in memory on individual servers. Since RDDs reside in volatile memory, a server failure results in the loss of any RDD partitions stored on it. To handle such failures, Spark automatically recomputes lost partitions from the set of operations that created it. To facilitate efficient recomputation, Spark restricts the set of operations, called *transformations*, that create RDDs, and explicitly records these operations. In particular, each RDD is an immutable read-only data structure created either from data in stable storage, or through a transformation on an existing RDD.

Spark records RDD transformations in a *lineage graph*, which is a directed acyclic graph (DAG) where each vertex is an RDD partition and each incoming edge is the transformation that created the RDD. Importantly, transformations are coarse-grained in that they apply the same operation to each of an RDD's partitions in parallel.



(a) No checkpointing (b) RDD-a checkpointed

Figure 4.1: The loss of RDD-b's partition #2 results in recomputation using lineage information. Partitions may be computed in parallel on different nodes.

Thus, Spark may use the lineage graph to recompute any individual RDD partition lost due to a server failure from its youngest ancestor resident in memory, or, in the worst case, from its origin data on disk. In addition, Spark allows programmers to save, i.e., checkpoint, RDDs, including all of their partitions, to disk, e.g., in a distributed file system, such as HDFS [188]. In this case, rather than recompute a lost RDD partition from its origin data (or its youngest ancestor resident in memory), as depicted in Figure 4.1a, Spark may recompute it from its youngest saved ancestor, as depicted in Figure 4.1b. Importantly, Spark provides no policies for checkpointing, and leaves checkpointing decisions to the programmer. Flint provides an automated checkpointing policy that we discuss in the subsequent sections.

Spark's RDD abstraction is versatile and has been used for long-running "big-data" batch jobs, as well as interactive data processing. For example, users can used a Read-Eval-Print-Loop (REPL) for interactive and exploratory analysis. As another example, Spark can be employed as a database engine with SQL queries executed via a translation layer such as Spark-SQL. Both examples require the Spark cluster to remain available for long periods of time: an exploratory REPL analysis may take several hours, and a database engine must be continuously available. Hence, if transient servers are used as cluster nodes, there is a risk of losing in-memory state, requiring significant overhead to regenerate and thus severely degrading interactivity.

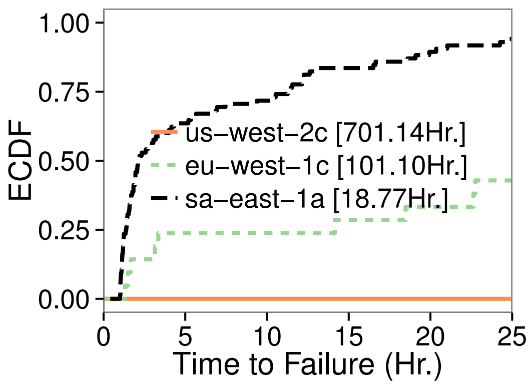
Flint includes automated policies to mitigate and respond to resources losses due to transient server revocations.

4.3 Flint Overview

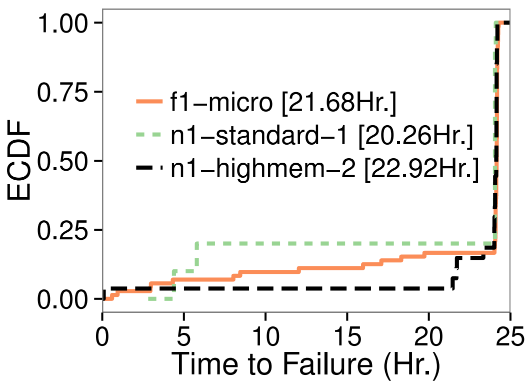
Flint is an application-aware framework for executing BIDI jobs on transient cloud servers. Flint's current design supports Spark-based BIDI jobs, implements application-aware, i.e., Spark-aware, policies for selecting and provisioning which transient servers to run on (based on their price and revocation rate characteristics), and determines when and how frequently to checkpoint application state, e.g., RDDs, based on the expected transient server revocation rates. To ensure transparency to end-users, Flint runs unmodified Spark programs. While Spark exposes a checkpointing interface for RDDs (via the `rdd.checkpoint()` operation), it requires the programmer to explicitly use it in Spark programs. Flint automates the use of Spark's RDD checkpointing mechanism by intelligently determining what RDDs to checkpoint and how often to do so.

Flint selects transient servers to minimize the overall cost of running a BIDI job that takes into account the average per-hour price of each transient server and the overhead of recomputing lost work based on the revocation rate. Since Flint's objective is to achieve performance near that of on-demand servers, on a revocation, it always requests and provisions a new transient server to maintain the original cluster size.

As noted earlier, Spark exposes an interface to checkpoint RDD state to disk, but leaves it to the programmer to determine what RDDs to checkpoint, when, and how frequently. Flint exploits this flexibility to implement an intelligent, automated checkpointing strategy tailored for transient servers. While EC2 provides a two minute revocation warning, it is not sufficient to complete Spark checkpoints of arbitrary size and restarting from incomplete checkpoints is not safe. Google provides an even



(a) EC2 spot instances



(b) Google preemptible VMs

Figure 4.2: Empirically obtained availability CDFs and MTTFs of transient servers on Amazon EC2 and Google GCE.

smaller warning of only 30 seconds. Thus, Flint does periodic checkpointing in advance so there is always some checkpoint of previous RDDs.

In general, the overhead of recomputing lost work due to a transient server revocation poses a challenging problem, since it requires Flint to balance the overhead of checkpointing RDDs with the time required to recompute them. At low revocation rates, checkpointing too frequently increases running time by introducing unnecessary checkpointing overhead, while similarly, at high revocation rates, checkpointing too rarely increases running time by causing significant recomputation. In addition, for *interactive* BIDI jobs, Flint must consider not only the overall cost of running a program to completion, but also the *latency* of completing each action within the program. For example, an interactive program might trade a small increase in overall cost (and running time) for a more consistent latency per action. Because of the performance and cost requirements of batch and interactive applications differ, we propose separate policies for batch and for interactive Spark applications in the subsequent sections.

4.4 Flint Design

Flint has two main components: a checkpointing policy implemented as part of the Spark runtime, and a per-job server selection policy. Flint's transient server selection policy runs as a separate node manager that monitors transient server characteristics, such as the recent spot price history for different instance types on EC2, to initially select transient servers for the cluster and to replace revoked transient servers while the program is running. We first discuss Flint's checkpointing and server selection policy for batch applications, and then extend it to support interactive applications.

4.4.1 Batch Applications

For batch BIDI jobs, Flint's goal is to execute batch-oriented Spark programs with near the performance of on-demand servers, but at a cost near that of transient servers. In this case, Flint provisions a *homogeneous* cluster of transient servers for each user. Since all transient servers in the cluster are of the same type, and the same bid price is used to provision them, it follows that when the market-driven spot price rises above this bid price, *all* servers in the cluster will be simultaneously revoked. Flint's checkpointing policy, discussed below, is derived from this insight and is specifically designed to handle the case where all servers of a cluster are concurrently revoked. We then outline the optimal server selection policy for batch applications, which leverages our assumption above that all transient servers are homogeneous.

4.4.1.1 Checkpointing Policy

Running a batch-oriented Spark program on a homogeneous cluster of transient servers, where a revocation causes the entire cluster to be lost simultaneously, is analogous to a single-node batch job that experiences a node failure—in both cases the job loses all of its compute resources. We adapt a well-known result from the fault tolerance literature [82] for deriving the optimal checkpointing interval for single node batch jobs for a given Mean-Time-to-Failure (MTTF). This optimal checkpointing interval minimizes the running time of a batch application when considering the rate of failures (or revocations), the overhead of checkpointing, and overhead of recomputation. Note that minimizing a batch application's running time also minimizes its cost on cloud platforms, since cost is structured as a price per unit time of use.

For a single-node batch job, running on a server with a given MTTF and a time to checkpoint δ, a first-order approximation of the optimal checkpointing interval is $\tau_{\text{opt}} \sim \sqrt{2 \cdot \delta \cdot \text{MTTF}}$ [82]. This approximation assumes the time to write the checkpoint is constant at all intervals and $\delta \ll MTTF$; if the MTTF is smaller than δ then there is no guarantee the job will finish, as it will continue to fail before completing each checkpoint and not make forward progress. In Flint's case, the $\delta \ll MTTF$ constraint holds, since δ is on the order of minutes (to write RDD partitions of varying sizes to remote disk) and the MTTF for transient servers in EC2 and GCE is on the order of hours (see Figure 4.2). We also assume that the failures occur according to a poisson process (inter-arrival times between revocations are exponentially distributed).

Note that in the single-node case, the optimal checkpointing interval depends only on the MTTF and the checkpointing time δ, and not the running time of the job. In Flint's case, we can derive the expected MTTF of each type of transient servers on both EC2 and GCE. Since Amazon EC2 revokes spot instances whenever their spot price rises above a user's bid price, we can use historical prices for each instance type to estimate their MTTF for a given bid price. Amazon provides three months of price history for each spot market, and longer traces are available from third-party repositories [119]. While GCE does not expose a similar type of indirect information about revocation rates, we know that GCE always revokes a server within 24 hours of launching it. In addition, users may estimate the MTTF for a small cost by issuing requests for different server types and recording their time to revocation. We performed such measurements, and found that currently in GCE the MTTF is near 24 hours (see Figure 4.2). In contrast, the MTTF varies much more widely between server types in EC2 due its dynamic pricing. For example, with a bid price equal to the on-demand price for the equivalent server, the MTTF ranges from 18–700 hours.

In addition to deriving the MTTF, Flint must also determine what in-memory state to checkpoint during each interval τ, which dictates the checkpointing time δ. Flint's checkpoint policy for batch applications is as follows.

Policy 1: *Every τ time units, checkpoint RDDs that are at the current frontier of the program's lineage graph.*

Thus, rather than checkpoint all state in the RDD cache on each server, which spans both memory and disk, every τ time units, Flint only checkpoints each new RDD at the *frontier* of the lineage graph every interval. The frontier of the lineage graph contains the most recent RDDs for which all partitions have been computed, and whose dependencies have not been fully generated.

Thus, in the lineage graph, the frontier includes all RDDs that have no descendants, i.e., the current set of sink nodes in the graph. Note that although the complete lineage graph is not known *a priori* since it generates new RDDs and evolves dynamically as the program executes, the lineage graph's frontier is always well-known. Specifically, Flint signals that a checkpoint is due every interval τ. After signaling, each new RDD generated at the frontier of its lineage graph is marked for checkpointing. Note that RDDs that are already (or are in the process of) being computed have no guarantee of being in memory, and may require recomputation. Spark maintains a cache of RDD partitions on each server that swaps RDD partitions to and from disk based on their usage, and may delete RDD partitions if the cache becomes full.

Once each RDD at the frontier of each lineage graph has been checkpointed, Flint will not checkpoint any subsequent RDDs that are derived from them in the lineage graph until the next interval τ. We assume here that the computation time for RDDs does not exceed the checkpointing interval τ. Since most RDD transformations, such as map and filter, have narrow dependencies, the computation time for any single

RDD is brief. However, we treat shuffle actions with wide dependencies as a special case, since each RDD partition that results from a shuffle depends on all partitions in the dependent RDD, resulting in a longer computation time. Because shuffles involve a larger amount of recomputation due to their wide dependencies, we checkpoint shuffle RDDs more frequently at an interval of τ divided by the number of RDD partitions that are being shuffled *from*. If server revocations occur *during* a shuffle operation (which act as barriers), then it is possible that the other nodes might end up waiting until the shuffle data is recomputed, as in a bulk synchronous parallel system. In all other cases, the recomputation operation does not cause waiting.

Finally, unlike in the optimal formulation above, the number of RDDs and the time required to write them to disk, i.e., the checkpointing time, is not static, but dictated by each program. Thus, Flint maintains a current estimate of the checkpointing time δ based on the time it takes to write all RDD partitions, which have a well-known size, in parallel to the distributed file system. As δ changes, Flint dynamically updates the checkpointing interval τ as the application executes. Although an accurate δ estimate improves the accuracy of the checkpoint interval τ, we note that τ is proportional to the square root of δ, which reduces estimation errors. More importantly, since we only checkpoint when RDDs are generated and not at arbitrary times, the system is not particularly sensitive to an accurate estimation of τ.

Flint supports general Spark programs with arbitrary characteristics. Thus, Flint's dynamic checkpointing interval automatically adapts to the characteristics of the program, checkpointing more or less frequently as the overhead due to checkpointing falls and rises, respectively. Note that, since RDDs are read-only data structures, the checkpoint operation in Spark is asynchronous and does not strictly block the execution of other tasks. However, checkpointing tasks consume CPU and I/O resources that proportionally degrade the performance of other tasks run as part of a Spark program.

4.4.1.2 Server Selection Policy

Based on the checkpointing policy above, we can compute the expected percentage increase in running time for a Spark program when running on transient servers with different price and revocation rate characteristics. Our goal is to choose a single type of transient server that has the least increase in running time (and thus, the least cost) to provision a homogenous cluster to execute the program. Specifically, in the case of spot instances, for a market k with an $MTTF_k$ based on the revocation rate at a certain bid price over the recent spot price history, the overall expected running time $E[T_k]$ for the program with a running time of T without any revocations is as follows.

$$E[T_k] = T + \frac{T}{\tau} * \delta + \frac{T}{MTTF_k}\left(\frac{\tau}{2} + r_d\right) \tag{4.1}$$

The first term is the running time of the program without any revocations, the second term is the additional overhead over the running time of the program due to checkpointing, and the third term is the additional overhead over the running time T due to provisioning new replacement servers (r_d) and recomputing the lost work, assuming that revocations are uniformly distributed over the checkpointing interval τ. The delay r_d for replacing a server is a constant—for EC2, it is typically two minutes. Factoring out T yields, $E[T_k] = T(1 + \frac{1}{\tau} * \delta + \frac{1}{MTTF_k}(\frac{\tau}{2} + r_d))$. Thus, we only need to know δ in addition to $MTTF_k$ to compute the percentage increase in running time on a market k. We conservatively estimate an initial δ by assuming our Spark cluster is properly sized for the application, and derive δ assuming that all memory is in use by active RDD partitions that must be checkpointed. We record the computation time for each RDD partition, and assume that the recomputation time for a partition will be the same as its initial computation time, given the same resources available. The immutable nature of RDDs, the lack of external state dependencies, and the static RDD dependency graph means that we can safely make this assumption.

Given $E[T_k]$ above, if the average per-unit price of each market k is p_k, we can derive the expected cost simply as :

$$E[C_k] = E[T_k] * p_k = T(1 + \frac{1}{\tau} * \delta + \frac{1}{MTTF_k} * \frac{\tau}{2}) * p_k \tag{4.2}$$

Since T is a constant, minimizing $E[C_k]$ requires choosing the market where the product of p_k and $(1 + \frac{1}{\tau} * \delta + \frac{1}{MTTF_k} * \frac{\tau}{2})$ is minimized. To do so, Flint simply evaluates this product across all potential spot markets, and provisions all servers in the cluster from the market that yields the minimum overall cost. Of course, for each market, a different bid price yields a different $MTTF_k$ and p_k. By default, Flint bids the equivalent on-demand price for all spot instances, as spot instances are cheaper if the spot price is less than the on-demand price. Note that we include on-demand instances as a spot market with an infinite MTTF (where checkpointing is not required). If the average price p_k exceeds the on-demand price, Flint transitions to using on-demand instances.

Note that, selecting servers from any other market than the one that yields the minimum overall cost will increase the overall cost for executing a batch application. Since batch applications are delay tolerant and are concerned with overall running time and cost, they can tolerate simultaneous revocations of all servers—the job can resume from a prior checkpoint. This insight enables us to model parallel Spark programs similarly to single-node batch applications. The performance characteristics under different number of simultaneous failures are shown in Section 4.6.3.

Restoration Policy. Whenever a revocation event occurs, Flint uses the same process as above to immediately select a new market to resume execution. When selecting a new market, Flint does not consider the market that experienced the revocation event, since the instantaneous price of that market has risen and the servers are not available. In addition, while Flint bases its selection on the average market price over a recent window, it does not consider markets with an instantaneous price

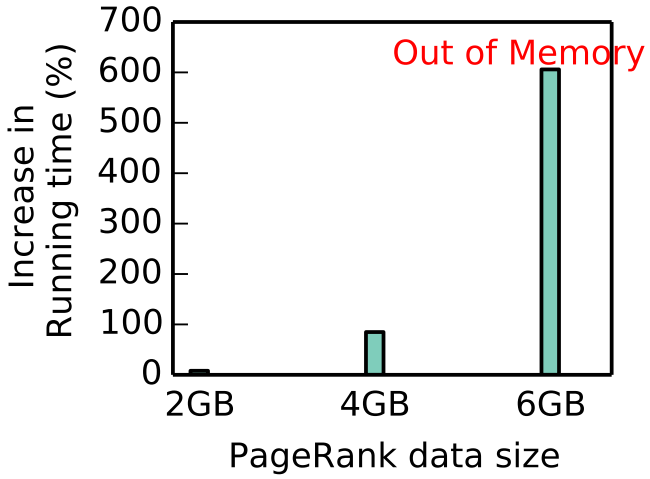


Figure 4.3: Simultaneous server revocations substantially increase running time if Spark runs out of available memory.

that is not within a threshold percentage, e.g., 10%, of the average market price. In the worst case, where prices across all markets are high, Flint resumes execution on on-demand servers, which are non-revocable.

4.4.2 Interactive Applications

While large simultaneous revocations do not degrade the running time of batch applications (relative to the same number of individual revocations), they do degrade the response latency of interactive BIDI jobs. Large simultaneous server revocations result in the need to concurrently recompute many RDD partitions, creating contention for resources on the surviving servers, which must multiplex their current work with recomputing lost RDD partitions. In the worst case, if the RDD working set is larger than the available memory on the remaining servers, Spark must swap RDD partitions between memory and disk as necessary to execute each task. Figure 4.3 shows the impact on performance under memory pressure due to large revocations. Such swapping increases the latency for interactive BIDI jobs.

Thus, applying the above policies designed for batch BIDI jobs to interactive ones will result in highly variable response latencies. Hence, rather than minimizing the expected cost and running time on transient servers, interactive BIDI jobs also value minimizing the variance between the maximum latency and the average latency of actions to provide more consistent performance, as opposed to excessively long latencies for some actions and short latencies for others. We can reduce this variance in latency by constructing a *heterogeneous* cluster for each interactive BIDI job—by mixing together different types of transient servers, e.g., from different spot markets, in the same cluster. Assuming that the demand and supply dynamics, and hence spot prices, for different transient server types are uncorrelated, a price spike in one market is independent of others. Hence, the revocation of one type of transient server due to a price increase in its market is independent of others and the cluster will only lose a fraction of the servers on each revocation event. In this case, the Spark program will continue execution on the remaining nodes, albeit more slowly, and can resume normal execution after Flint restores the revoked nodes to the cluster. We describe this policy as follows.

Policy 2: *Reduce risk through diversification—choose transient servers of different types with uncorrelated prices to reduce the risk of simultaneous revocations.*

Figure 4.4 shows that revocations on many (but not all) spot markets in EC2 are in fact independent and uncorrelated. However, as discussed above, selecting any market other than the one that minimizes the overall cost will increase the overall cost and running time of the application. Thus, even for interactive BIDI jobs, it is important to intelligently mix transient servers from different markets to reduce the variance in latency without significantly increasing the overall cost and running time of the application.

4.4.2.1 Checkpointing Policy

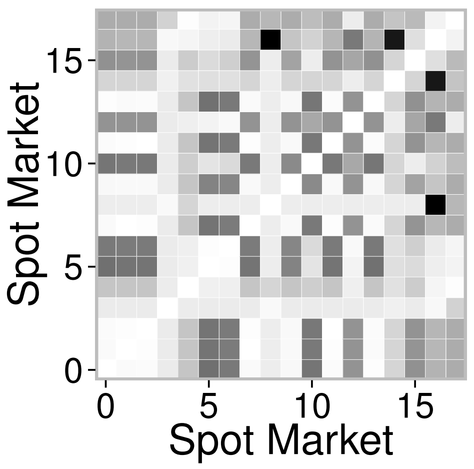
Before discussing our selection policy for interactive applications, we must first determine the appropriate checkpointing policy. Of course, we could also reduce our variance in latency by checkpointing more frequently, and thus enabling recovery from each large revocation event by simply reading in the checkpointed state. However, checkpointing too frequently degrades the average case performance. Instead, we extend the same checkpointing policy as above, assuming that we equally divide our cluster of size N across transient servers selected from m markets. In this case, we must estimate the aggregate MTTF for the heterogeneous cluster by computing the harmonic mean of $MTTF_1 \cdots MTTF_m$ of each individual server type within the cluster.

$$MTTF = \frac{1}{\frac{1}{MTTF_1} + \cdots + \frac{1}{MTTF_m}} \tag{4.3}$$

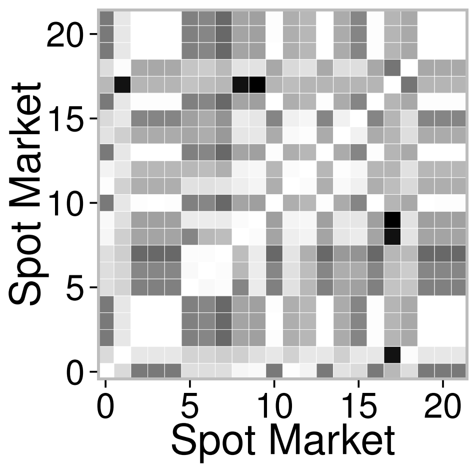
Note that the aggregate MTTF will be smaller than the MTTF for each individual market, in that there will be more revocation events, but each one will only result in the revocation of N/m servers. Thus, our checkpointing interval above will decrease, causing more frequent checkpoints. However, the size of each revocation event will also decrease, compared to using a single market. If we assume the overhead of recomputation for a Spark program is linear in the number of revoked servers, then when using more markets, the overhead of recomputation due revocation events decreases, while the number of revocation events increases. This decrease in the recomputation overhead for each event tends to balance out the increased number of revocation events due to the lower MTTF, although we leave a formal proof of this property to future work.

4.4.2.2 Server Selection

To intelligently provision a heterogeneous cluster, we first construct a subset L of mutually uncorrelated markets among all the possible markets. We construct L for two reasons. First, published price histories show that revocations usually do not happen simultaneously in different spot markets (in Figure 4.4, darker squares indicate



(a) *us-east-1a*



(b) *m2.2xlarge*

Figure 4.4: Publicly available EC2 spot price traces show that prices (and hence revocations) are pairwise uncorrelated (shown by darker squares) for most pairs of markets.

less correlated failures between any two markets). This observation confirms the feasibility of diversifying across markets to reduce concurrent revocation risk. Second, since there are potentially many markets to choose from—over 1000 markets in EC2's US-east region alone—constructing a smaller set of L markets prunes the search space. We greedily construct L by adding the most pairwise uncorrelated markets to L.

As before, we do not consider markets where the instantaneous risk of revocation is high, i.e., the spot price is not within some threshold of the average spot price. We then sort these candidate markets in order of their expected cost using the same approach as above for batch applications. After sorting the candidate markets, we greedily add markets to our set of selected markets S in order, as follows.

We first select the market that yields the minimum expected cost and then compute the expected variance in its running time based on the market's revocation rate. We compute the variance in the expected running time $\sigma^2 = E[T(S) - E[T(S)]]^2 = E[T(S)^2] - E[T(S)]^2$ for a set of markets S, where servers are equally distributed among the markets in S. In Equation 4.1 we have shown the scenario for a single market k; extending it to $m = |S|$ markets yields:

$$E[T(S)] = T + \frac{T}{\tau} * \delta + \frac{T}{MTTF(S)} \cdot \frac{1}{m} \cdot \left(\frac{\tau}{2} + r_d\right) \tag{4.4}$$

With multiple independent markets, when one market fails, only $1/m$ fraction of the servers are lost (because they are equally distributed among all markets). Note that the MTTF for multiple markets above is given by Equation 4.3.

We then select the market with the next lowest cost, and equally divide our servers between the two markets. For the mixed cluster, we again compute the expected variance in running time. If the expected variance is higher than the single market, we stop and do not add the second market; if the expected variance is lower, we evaluate the expected variance from dividing the servers among the three lowest cost markets. We continue adding markets until adding an additional market does not decrease the variance in running time. We also stop if the expected cost ever rises above the expected cost of running the application on on-demand servers. As shown in our evaluation, the result of this server selection algorithm is a mix of servers from different markets that decrease the variance in running time, providing consistent response latency, without significantly increasing the cost relative to the optimal cost in the batch case.

Restoration Policy. In addition to determining the initial mix of servers from different markets for the cluster, Flint must also replace a set of revoked instances from a particular market with instances from another market. To do so, Flint simply replaces these revoked instances with instances from the lowest-cost unused market in set L. As above, Flint does not consider markets with instantaneous prices that are significantly higher than their average price.

Bidding Policy. Flint uses a simple bidding policy to place bids for each spot server—we bid the on-demand price. Bidding in the current EC2 spot market has only a negligible effect on the average cost and the MTTF—these metrics stay the same for a very large range of bids. In Figure 4.11, which shows the expected costs for three instance types, we can see that the range of bids for which the cost remains unaffected is quite large. For example, bidding anywhere from 0.5 to 2× the on-demand price for the `m2.2xlarge` instance type yields the same cost. For all three types shown in the figure, the on-demand price is inside the wide range that yields the minimum price. Our simple bidding policy is thus motivated by the insensitivity of expected cost and MTTF on the bid [179], as well as a focus on systems mechanisms to handle spot revocations which can work even in environments where no bidding mechanisms apply, as in the case of GCE Preemptible Instances, which have a fixed price.

If market characteristics were to change, a modification to the simple bidding strategy might be necessary. Since Amazon provides up to three months of price history, the empirical relation between bids and the average price and MTTFs can be used to select bids that will minimize the expected cost using Equation 4.2. A similar approach can be found in [182, 179]. Lastly, we bid the same price for all the instances in a given market. However, a more sophisticated policy could stratify the bids within a market such that instances with different bid prices fail at different times. However, stratifying bids is not currently effective, as price spikes in the current spot markets are large and cause servers with a wide range of bids to all fail simultaneously [182].

Arbitrage. Flint reduces costs by using low-cost spot instances and spreads revocation risks by exploiting the uncorrelated prices across different spot instance markets. A concern is that neither of these characteristics of the spot markets might hold if systems such as Flint gain in popularity and the demand for spot instances increases. This increased demand might drive up market prices and volatility, causing the cost savings of spot servers to vanish. However, we believe that as infrastructure clouds continue to grow and add capacity, the surplus capacity (which is sold as spot servers) would also grow, such that the increase in demand would be matched by an increase in supply. Furthermore, systems like Flint only represent a small portion of users of the spot market: other users and systems utilize spot instances in different ways and

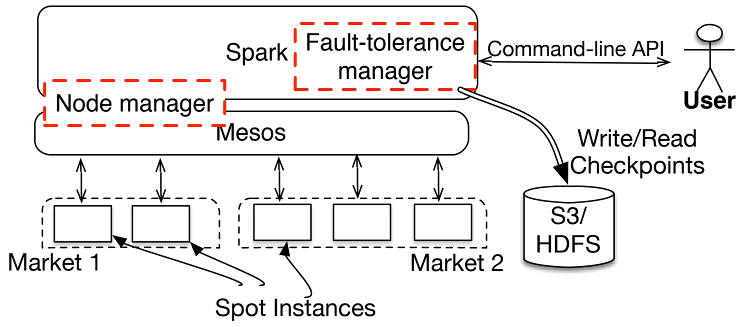


Figure 4.5: Flint architecture and system components.

have different spot instance demand characteristics. A more detailed analysis of the second-order market effects and other "game-theoretic" analysis is outside the scope of this paper, and we assume that the large numbers and sizes of the spot market will absorb the effects of the arbitrage opportunities Flint exploits.

4.5 Flint Implementation

We implemented a Flint prototype based on Spark 1.3.1 in about 3000 lines of Scala and Python. The prototype includes the policies for batch and interactive Spark applications from the previous section. Users interact with Flint via the command-line to submit, monitor, and interact with their Spark programs. Flint's implementation is split into two main components: a node manager that interfaces with Mesos and EC2, and implements the server selection policy, as well as a fault-tolerance manager embedded in Spark that implements the checkpointing policy (Figure 4.5). Our implementation primarily integrates with EC2 and supports spot instances. However, our approach is compatible with GCE, which offers transient servers at a fixed price per unit time.

To implement the server selection policy, the node manager accepts user requests for Spark clusters of size N to run their application, and selects the specific spot market(s) to provision the servers. To implement the batch and interactive server selection policy, the node manager monitors the real-time spot price in each EC2 spot market and maintains each market's historical average spot price and revocation rate (and MTTF) over a recent time window, e.g., the past week. The node manager acquires one or more servers in a particular market by placing a bid for them in the spot market at the on-demand price via EC2's REST API. Each Spark cluster in Flint runs in its own Virtual Private Cloud (VPC) that is isolated from other users. Flint launches the instances with its own customized disk image, e.g., an Amazon Machine Image (AMI), which contains a pre-configured version of a Spark master and worker. After the initial setup, Flint provides users with a web interface, as well as SSH connectivity, to the master and worker nodes to monitor job progress and use Spark interactively via the Spark shell.

Flint's fault-tolerance manager is written as a core Spark component so it may interact with Spark's internal APIs for scheduling, RDD creation, and checkpointing. The fault-tolerance manager monitors the set of RDDs at the frontier of the lineage chain, checkpoints them to stable storage every interval τ, and updates δ and τ as new RDDs are created. To compute τ, the fault-tolerance manager must interact with the node manager to retrieve the MTTF of each server in the cluster. To implement the checkpointing interval, the fault-tolerance manager maintains an internal timer for τ, and marks the first RDD in the queue from each active stage after the timer expires for checkpointing. If Flint marks an RDD for checkpointing, it checkpoints each individual partition of that RDD. To support automated checkpointing, we modify Spark's existing checkpointing implementation to enable fine-grained partition-level checkpointing. In Flint, once a task finishes computing its partition, it notifies Spark's DAG scheduler, which then invokes the fault-tolerance manager to check if it has marked the corresponding RDD for checkpointing. If so, it creates a new checkpointing task, which handles the asynchronous checkpoint write operation.

Checkpoint Garbage Collection. We have also implemented a garbage collector for checkpointed RDDs to reduce storage requirements. Checkpointing an RDD terminates its lineage graph and its ancestor RDDs are no longer “reachable.” Checkpoints for these unreachable RDDs are redundant and thus periodically removed. Lastly, to mitigate the impact of spot instance revocations, the node manager monitors the 120 second spot termination warning provided by EC2’s `/spot/termination-time` API. If Flint detects a warning on any worker, it immediately triggers the market selection on the node manager which selects and requests replacement instances. As part of this notification, the fault-tolerance manager informs the node manager of the most current values for δ and τ based on the collective size of the RDDs at the frontier of the lineage chain. The node manager needs these values to execute its server selection policy to replace revoked servers.

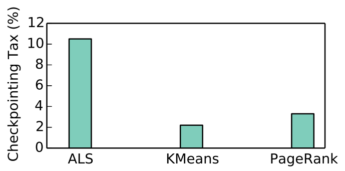
Checkpoint Storage. Flint stores all partition checkpoints that belong to a single RDD inside the same directory on HDFS. On recovery, we first check if the partition exists in the corresponding directory before any starting any RDD (re)computation. We use Elastic Block Store (EBS) volumes and treat them as durable storage. Using EBS instead of local, on-node storage has several advantages. Data on local disks is lost upon revocation, and a revocation event can cause the data loss such that HDFS cannot recover even using 3-way replication. In addition, the amount of local storage in EC2 is limited, e.g., 10GB on most nodes. However, unlike local storage which is free, there is an extra cost for EBS volumes which depends on their size and I/O rate. Because we use EBS to only store checkpoints, which we frequently garbage collect, the EBS disk size required is small. In addition, the I/O rate is limited as Flint judiciously regulates checkpointing frequency. We use the two minute revocation warning to pause all nodes, flush data, and cleanly unmount all EBS volumes. After revocation, we first let HDFS recover the missing chunks. If that fails for any reason, since the data on EBS volumes persists even after revocations, we copy all the data from the EBS volumes to the newly launched instances [31].

The SSD EBS volumes that Flint uses are currently charged \$0.10 per GB per month by Amazon. Because Flint provides Spark as a managed service, these EBS volumes are reused among jobs, and the EBS costs are thus amortized. EBS volumes required for storing Flint checkpoints cost about 1-2% of on-demand instances, adding an overhead of about 10-20% to the final cost using spot instances. A more detailed cost analysis and breakdown of storage is presented in the next section. We are not constrained in the choice of checkpoint storage and there are other options that are feasible as well. For example, Amazon's S3 object store is about 20 times cheaper than EBS, and is a viable option for reducing storage costs, albeit at worse read/write performance. Amazon's Elastic MapReduce File System (EMRFS [18]) uses a combination of S3 and DynamoDB database for low-cost storage for Spark. A similar storage configuration can be used for storing Flint checkpoints at low cost.

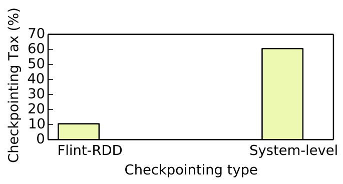
4.6 Flint Evaluation

We conducted our experiments by running popular Spark programs on Amazon EC2 to quantify Flint's performance and cost benefits for both batch and interactive BIDI workloads. We run all experiments on a Spark cluster of 10 `r3.large` instances in EC2's US-East region. Each `r3.large` instance has 2 VCPUs, 15GB memory, and 32GB of local SSD storage. We use persistent network-attached disk volumes from Amazon's Elastic Block Store (EBS) to set up the HDFS filesystem (with a replication factor of three) and use it to store RDD checkpoints.

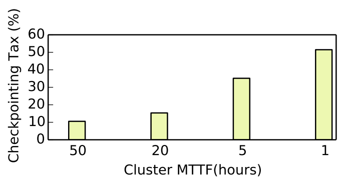
Our evaluation includes systems experiments using our Flint prototype to evaluate the effect of recomputation and checkpointing on real Spark applications, as well as simulation experiments to examine the cost and performance characteristics of Flint over long periods under realistic market conditions. We use a range of batch and interactive workloads in our evaluation, as described below. The input data sizes for each workload listed below were carefully chosen to max out the total cluster memory



(a) Performance overhead due to Flint's RDD checkpointing.



(b) System level distributed checkpointing overhead is large compared to RDD checkpointing.



(c) Checkpointing overhead for the ALS workload increases with increasing market volatility.

Figure 4.6: Performance overhead of system- and application-level checkpointing.

used by intermediate RDDs and to ensure stable Spark behaviour even under node revocations.

4.6.1 Workloads

PageRank. PageRank is a graph-processing workload that computes the rank of each page in a web graph iteratively based on the rank of the pages that link to it. PageRank is a good candidate for evaluating our checkpointing policy, since it creates a large number of RDDs—proportional to the number of vertices in the graph—and involves a large number of shuffle operations. We use the optimized PageRank implementation from Spark's graphx library. For our experiments, we use the Live Journal [25] dataset of size 2GB.

KMeans Clustering. KMeans is a clustering algorithm that partitions data points into k clusters with the nearest mean. We use KMeans clustering as an example of a compute-intensive application: it consists of applying a series of narrow dependencies to an RDD and then a large shuffle operation per iteration. We use the optimized implementation from Spark's `mllib.DenseKMeans` library with a randomly generated dataset of size 16GB. KMeans is a prototypical example of a iterative machine learning technique.

Alternating Least Squares. Alternating Least Squares (ALS) is a linear regression model that fits a set of data points to a function with the minimum sum of squared errors between the model and the data points. ALS's RDD lineage graph is similar in structure to KMeans. However, ALS is more shuffle-intensive where each transformation takes more time than with KMeans. We use Spark's `mllib.MovieLensALS` implementation on a 10GB dataset.

TPC-H. We use Spark as an in-memory database server that services clients issuing SQL queries from the TPC-H database benchmark with a data size of 10GB [28]. Since TPC-H queries are data-intensive, to accelerate query execution, Flint de-serializes and re-partitions the raw files stored on disk first and then persists them in memory as RDDs. Each time a new query arrives, Flint executes it using in-memory data rather than loading the data from disk again. TPC-H is an interactive workload where the query response latency is the primary metric, rather than the running time. The workload is shuffle- and join-intensive, as many SQL queries translate to shuffle and join operations on RDDs.

4.6.2 Quantifying the Checkpointing Overhead

We first verify and quantify the overhead due to checkpointing RDDs in Flint and compare it with both the performance of running on on-demand servers without checkpointing, and with a systems-level checkpointing approach. The overhead due to checkpointing dictates how close Flint's performance on transient servers comes to the performance of on-demand servers. For these experiments, we use a relatively low MTTF of 50 hours to highlight the checkpointing overhead—the MTTFs in current EC2 spot markets range from 20 to 2000 hours.

We first measure the checkpointing overhead for three batch workloads when the MTTF is equal to 50 hours using Flint's intelligent checkpointing algorithm. As Figure 4.6a shows, the performance overhead due to checkpointing, as percentage increase in running time, for all three batch applications is small, ranging from 2% to 10%. Of these three applications, ALS has the largest collective set of RDDs and hence also has the highest checkpointing overhead. Due to the larger data sizes and

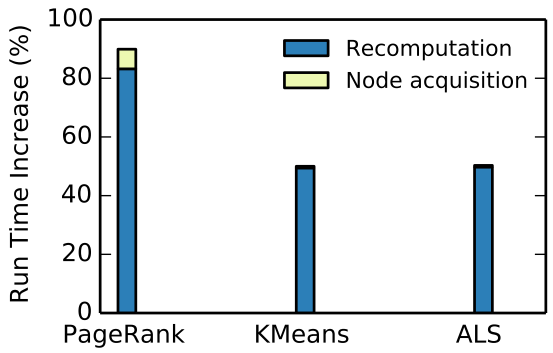
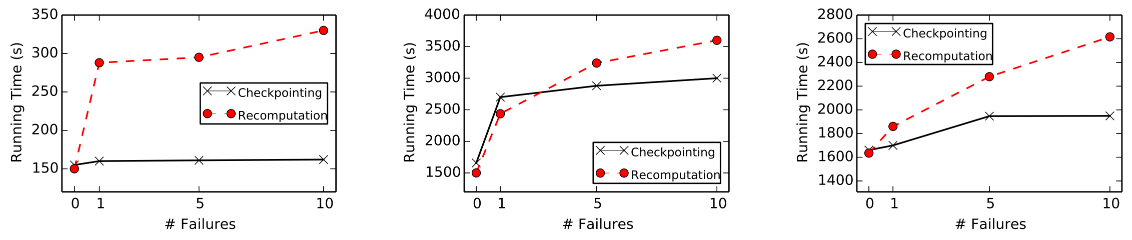


Figure 4.7: Recomputation of lost RDD partitions due to a single revocation causes a 50-90% increase in running time.

higher network utilization (the most constrained and bottlenecked resource for Flint), the checkpointing overhead for ALS is also the highest.

Next, we compare the performance overhead of Flint's intelligent application-level checkpointing with a systems-level approach using the same checkpointing frequency. A systems-level distributed checkpointing approach must checkpoint the entire memory state of each Spark worker, including all active RDDs, cached RDDs, shuffle buffers etc. In contrast, by checkpointing only the frontier of the RDD lineage graph from within the application, Flint can avoid checkpointing stale application state or unnecessary system state. Figure 4.6b compares the performance overhead of system-level checkpointing with FLint's RDD checkpointing for the ALS workload. The system-level approach increases the running time by 50% compared to our application-level approach that only checkpoints selective RDDs. The result demonstrates the benefit of leveraging fault-tolerance mechanisms that are already embedded into data-parallel frameworks for high failure-rate environments like transient servers.

Last, we measure the change in checkpointing overhead when running the ALS workload on transient servers with varying volatility. Figure 4.6c shows that, as expected, the checkpointing overhead increases as the transient servers become more volatile (with a higher revocation rate and a lower MTTF). With a highly volatile market, where the MTTF is 1 hour, Flint's checkpointing overhead increases from 10%



(a) PageRank (b) ALS (c) KMeans

Figure 4.8: Application running times under various failure scenarios with and without checkpointing. Applications are running on a cluster of size ten. Zero indicates the baseline case of no failures.

to 50% of the application's typical running time. This result represents an upper bound on Flint's checkpointing overhead, since any further increase in the checkpointing overhead will exceed the RDD recomputation time.

Spreading application nodes across multiple availability zones also does not seem to hurt application performance significantly. We found no noticeable decrease in the performance of KMeans, and only a 7% degradation for the ALS workload. While the inter-availability zone network latencies are certainly much higher compared to within a zone, we conjecture that large checkpoint writes are bandwidth-sensitive and not latency-sensitive, and multiple availability zones can thus be used without a large performance penalty.

Result: *Flint's checkpointing overhead is low, increasing application running time between 2 and 10% even with relatively low MTTF values. In addition, even for extremely volatile markets, Flint's checkpointing overhead increases running time by less than 50%. Further, Flint's application-level approach significantly reduces the overhead relative to systems-level checkpointing (from a 50% increase in running time to a 10% increase in running time for ALS).*

4.6.3 Impact of Server Revocations

We now consider various transient server revocation scenarios and system configurations to demonstrate their impact on running time. We are interested in evaluating the overhead of recomputation triggered by server revocations. In all experiments, revoked servers are replaced by new transient spot servers, such that Flint maintains a cluster size of ten.

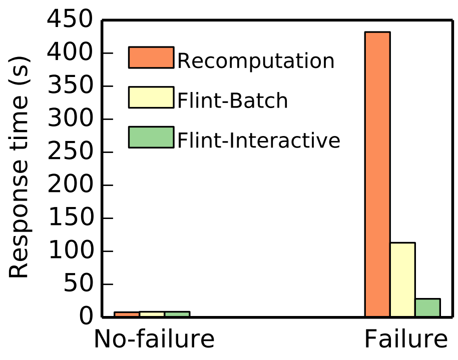
Figure 4.7 shows the performance impact of a single server revocation out of a cluster of size ten without Flint's intelligent checkpointing policy. The figure illustrates that a single revocation can cause running time to increase sharply, up to 90% in the case of PageRank. Since Flint immediately replaces any revoked server, the increase in running time is due to two factors: i) recomputing RDD partitions lost due to the revocation and ii) the time to acquire replacement servers. For PageRank, the time to acquire a new server contributes 5% of the increase in running time with the rest of the increase coming from recomputing RDDs. For the other two applications, which have longer running times, the time to acquire replacement servers is negligible, and all of the increase is due to recomputing lost RDDs.

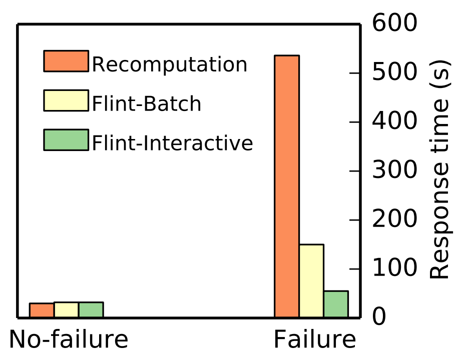
We also evaluate the impact of the number of concurrent revocations on performance. Figure 4.8 shows the total running time for the three batch applications when varying the number of concurrent server revocations without checkpointing. Here, a value of zero represents the baseline case with no failures. Figure 4.8 shows that application running time increases as the number of concurrent revocations increases, by up to 100%. The large overhead is due to the recomputation of lost RDD partitions, as well as their recursive dependencies. The graph also shows that running time is not strictly a linear function of the number of concurrent revocations: the impact on performance decreases with each additional revocation. Thus, for batch jobs, Flint's approach of using only a single market where all transient servers are concurrently revoked incurs less overhead than spreading servers across multiple markets with more frequent, but smaller, revocation events.

Result: *Without checkpointing, recomputation due to revocation of even a few servers, causes a significant increase in running time and cost. The impact on running time for batch applications tends to decrease as the size of the revocation event increases, which supports Flint's batch checkpointing policy for that selects spot instances from the same market.*

Figure 4.8 also compares the running time with and without Flint's checkpointing policy as the size of the revocation events increase. Since checkpointing bounds the amount of recomputation, the running time with checkpointing is significantly smaller than the recomputation-only configuration for all the three workloads. For PageRank(Figure 4.8a), checkpointing is particularly beneficial—periodically saving the shuffle output drastically reduces and bounds the recomputation required on a revocation. Similarly, checkpointing the RDDs in KMeans (Figure 4.8c) bounds the performance degradation when moving from 5 to 10 simultaneous failures. Further, the sublinear relationship between the size of the revocation event and the running time is even stronger when using checkpointing. That is, as the size of the revocation event increases, with checkpointing, the increase in running time flattens out, reflecting the bound on performance degradation due to checkpointing. Of course, with no revocation events, applying the checkpointing policy slightly increases running time due to the overhead of checkpointing, although this increase is not significant.

So far we have evaluated Flint's performance on a cluster with ten machines. As cluster size grows, system scalability is governed by the scalability of the underlying Spark engine, as well as performance of the checkpoint storage backend (HDFS in our case). Flint's policies for market selection are applicable when an application starts and after revocation events and thus incur little run-time overhead. Both Spark and HDFS have been known to scale well to cluster sizes in the hundreds of nodes [50].





(a) Short query (b) Medium-length query

Figure 4.9: Flint's interactive mode results in 10-20× improvement in TPC-H response times during failures.

However, we leave a more detailed analysis of Flint on larger cluster sizes to future work.

Result: *Flint's checkpointing policy significantly reduces the increase in running time due to revocations, by 15-100% for our three representative batch applications.*

4.6.4 Support for Interactive Workloads

Checkpointing is even more essential for interactive applications. Figure 4.9 shows the response time of two queries—query three and query one of TPC-H—with and without revocations. In this case, our revocation scenario is either all ten servers are concurrently revoked (when using either recomputation without checkpointing or Flint's batch checkpointing policy), or a single server is revoked ten times (when using Flint's interactive policies).

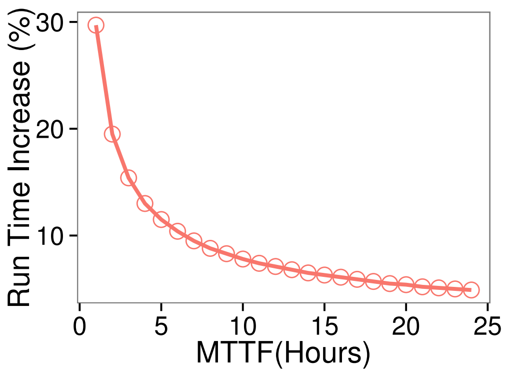
Without revocations, the checkpointing overhead for Flint's batch and interactive modes is low (~10%). The response time without revocations is low for all three of our policies: recomputation without checkpointing, Flint's batch checkpointing policy, and Flint's interactive checkpointing policy. For a small query, the latency is a few seconds, and for a larger query, the latency remains less than ten seconds. However, with revocations, the response time rises substantially to 400-500 seconds for both query types without any checkpointing. The rise occurs because recomputing the RDDs lost due to revocation requires re-fetching the input data from Amazon's S3 storage service, and then again re-partitioning and de-serializing the data.

Using Flint's batch checkpointing policy, the response time reduces by a factor of 4×. In addition, using Flint's interactive checkpointing policy, which is explicitly designed to trade-off cost for interactive performance, reduces the response time even further: from 100-150 seconds with the batch checkpointing policy to 28-55 seconds with the interactive checkpointing policy. This additional reduction (3×) in the response time is due to the interactive checkpointing policy and the market selection that mixes different types of servers in the same cluster. Flint's batch policies select markets to minimize the expected cost, while Flint's interactive policy also considers the variance in response time when selecting markets. This experiment demonstrates the benefit of considering response time.

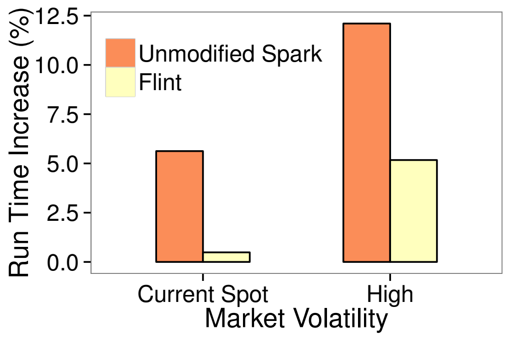
Result: *Flint's checkpointing and server selection policies decrease the response time of interactive workloads by an order of magnitude (∼10×). Flint's interactive policy results in lower response times than its batch policy, since it spreads risk by mixing transient servers from different markets.*

4.6.5 Cost-Performance Tradeoff

To quantify the impact of Flint on cost and performance, we use traces of EC2 spot prices from January to June 2015. We also use empirically collected availability statistics for over 100 GCE Preemptible Instances that we requested and were revoked over a one month period in August 2015. In addition to examining Flint's cost and performance on real data, we also present results on synthetic data with lower MTTFs to demonstrate Flint's performance under extreme conditions, i.e., with high market volatility. For these experiments, we simulate the performance of a canonical program that checkpoints 4GB RDD partitions every interval.



(a) Performance vs. MTTF



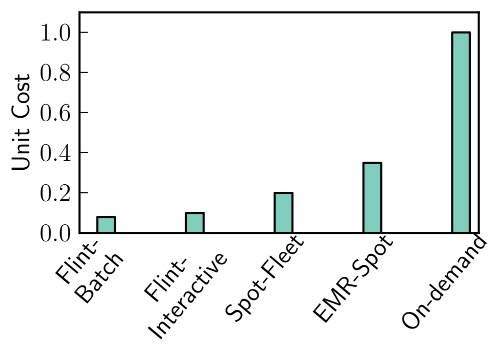
(b) Flint vs. Unmodified Spark

Figure 4.10: Flint's increase in running time compared to using on-demand servers is small in today's spot market, and is low even for highly volatile markets equivalent to GCE.

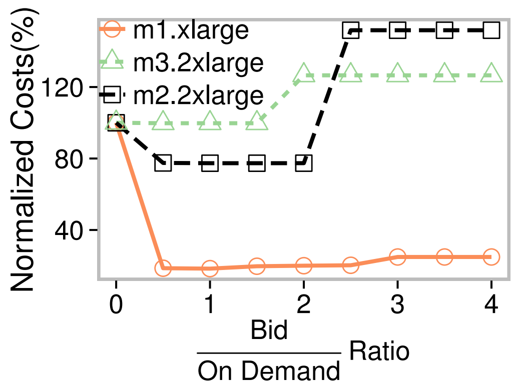
We first demonstrate the decrease in running time as the MTTF of the transient servers increases. As shown in Figure 4.10a, once the MTTF extends beyond twenty hours, Flint's increase in running time is less than 10% compared to using on-demand servers. Since MTTFs of twenty hours are on the low end for EC2 spot markets (assuming a bid equal to the on-demand price), Flint's performance on transient servers will be on par with on-demand servers. Figure 4.10b quantifies this performance by showing the increase in running time when using Flint on spot instances compared to using on-demand servers. The graph shows that in the current spot market there is little increase (<1%) in running time when using Flint versus using on-demand servers. By contrast, when running unmodified Spark on spot instances (while still employing Flint's server selection policies), the increase in running time is over 5%.

Of course, the existing spot market in EC2 is under-utilized and not particularly volatile. Thus, we also show results for a higher volatility market based on our GCE data with an MTTF close to 20 hours. In this case, running unmodified Spark on spot instances increased running time by 12%—Flint's increase is <5%.

Result: *Flint causes a small increase in running time (1%-7%) compared to on-demand servers for transient servers with both high and low volatilities, represented by EC2 spot markets and GCE preemptible instances, respectively.*



(a) Flint cost saving.



(b) Cost as a function of the bid price.

Figure 4.11: Flint determines the bid for each market based on our expected cost model. Flint is able to run both batch and interactive applications at 10% of the on-demand cost.

Lastly, we quantify Flint's cost savings for batch and interactive workloads compared to running on equivalent on-demand instances. We compare Flint's server selection policies from Section 3 with multiple existing approaches for running Spark on EC2 spot instances. In particular, we compare against EC2's Elastic MapReduce (EMR) managed service to execute unmodified Spark programs on spot instances. Note that Spark-EMR on EC2 incurs an additional flat fee of 25% of the on-demand price per hour in addition to the cost of the spot instances. We also examine using SpotFleets in EC2, since this is an application-agnostic service that EC2 provides to automatically replace revoked spot instances with a spot instance from another market. Interestingly, this EC2 service, like Flint, automatically bids the on-demand price for spot instances on behalf of users. SpotFleets enable users to set a policy that automatically selects an instance from either the cheapest market or the least volatile market (without considering the impact of revocations on performance). Thus, comparing Flint with SpotFleet represents the benefit of embedding the server selection and replacement policy into Flint and making these policy decisions based on the application characteristics.

For this experiment, we configure SpotFleets to use two `r3` instance types in the fleet. Flint's cost-aware server selection (for both batch and interactive jobs) results in 90% cost savings compared to executing on on-demand servers. Combined with our previous result that showed the overhead of Flint compared to using on-demand servers in the current spot market, this demonstrates that Flint achieves its goal of executing BIDI workloads at a performance level near that of on-demand servers, but at a price near that of transient servers. In addition, Flint's batch and interactive policies also lower costs relative to using Spark-EMR on spot instances by 66%, and lower costs relative to using SpotFleets by 50%. These results are important in that they demonstrate Flint's cost savings are not simply due to the fact that spot instances are significantly cheaper than on-demand servers. Since Spark-EMR and SpotFleets also use spot instances, the savings stem solely from Flint's intelligent application-aware checkpointing and server selection policies.

At current spot prices, improving on the cost of using on-demand servers is not challenging—even simple strategies for using spot instances are capable of improving on on-demand costs. In contrast, by comparing with Spark-EMR and SpotFleets, we show that Flint not only results in lower costs than using on-demand servers, but also lower costs than using spot instances when using unmodified Spark and application-agnostic bidding strategies, respectively.

Flint uses EBS for checkpoint storage, which incurs an extra cost. Due to the low space requirements of periodic checkpointing and garbage collection, these storage costs are also low. EBS volumes cost $0.1 per GB *per month*, and because Flint provides Spark "as a service," these volumes can be reused among different jobs, and thus their monthly cost is amortized. The `r3.large` servers we use have 15GB of main memory, and we conservatively provision twice that memory for storing checkpoints. Note that Spark only uses 40% of RAM for storing the RDD data—the rest is used as an RDD cache—thus we effectively over-provision by more than a factor of four, and can always add more EBS volumes if storage space is running low by dynamically extending HDFS. The hourly cost for EBS volumes has an overhead of $0.1 * 30/(24 * 30) = 0.004$. This extra cost is $\sim$2% of the on-demand cost and 20% of the average spot instance costs. We account for these storage costs in our cost analysis.

Finally, Figure 4.11b shows the cost of using different spot instance types on EC2 as a function Flint's bid price. This figure demonstrates that in the current EC2 spot market, Flint's default policy of bidding the on-demand price results in the lowest cost. As the figure shows, there is a wide range of bid prices for each market, ranging from roughly half the on-demand price to 1.5× the on-demand price that yield the lowest cost. This behavior results from the spot prices in EC2 being "peaky" where they frequently spike from very low to very high, and then return to a low level. As a result, placing a bid price somewhere above the low steady state, but below the average peak, results in the same cost. Thus, unlike prior work that focuses on optimizing bidding strategies for EC2 spot instances, we find that in practice simply bidding the on-demand price is optimal, and that there is actually a wide range of bid prices that result in this optimal cost.

Result: *Flint executes applications at near the performance of on-demand servers (within 2-10%) but at a cost near that of spot servers, which is 90% less than using on-demand servers and 50-66% less than using existing managed services such as SpotFleets and Spark-EMR.*

4.7 Related Work

Our work builds upon a large amount of prior work on spot instances, as well as fault tolerance mechanisms.

Spot Markets. Since servers in the spot market are significantly cheaper than the equivalent on-demand servers, they are attractive for running delay-tolerant batch jobs [197, 118, 14]. Checkpointing is a common fault-tolerance mechanism for mitigating the impact of revocations on batch jobs in the spot market [203, 125, 225]. However, Flint employs fine-grained application-level checkpointing, rather than systems-level checkpointing, as in previous work. In addition, Flint focuses on distributed data-parallel jobs and not simple single-node batch jobs, as in recent work [197].

Prior work has also used spot instances for data-parallel tasks. For example, EC2's EMR service that we compare against [18] allows Hadoop and Spark jobs to run on spot instances, and may be combined with SpotFleets to define an automated policy to replace revoked spot instances. However, these services are application-agnostic and, as we show, by not considering the application characteristics they may make non-optimal decisions. In addition, prior work has explored running Hadoop jobs on spot instances [136, 75]. However, Hadoop lacks the built-in checkpointing and recomputation mechanisms that Flint leverages in Spark. Prior work has also explored running a distributed database on spot instance [65, 168]. This work addresses the problem of deciding serialization points for database operations, which differs from Flint's focus on defining checkpointing and server selection policies. Finally, Flint also supports interactive workloads. Prior work demonstrates that single-node interactive applications can be run on spot instances using continuous system-level checkpointing and nested virtualization [182]. However, Flint is a *distributed* data-parallel system for running BIDI workloads on transient servers.

Fault-tolerance Mechanisms. The performance effects of server failures has been well studied for Hadoop [89, 95]. Similarly, our work models the impact of server failures and revocations in Spark. Flint's intelligent checkpointing approach to minimize running time is based on the optimal approach for single-node batch jobs [82]. Other checkpointing mechanisms and policies have been developed for other types of applications. For example, Zorro uses checkpointing and other optimizations to recover from failures in distributed graph processing frameworks [166]. Similarly, Naiad also includes a policy for automatically checkpointing vertices and recovering from server failures [154]. Spark Streaming [229] incorporates automated periodic checkpointing of RDDs to enable real-time data processing, but does not take into account recomputation overhead and cluster volatility. These systems' policies may also benefit BIDI workloads running on transient servers, and are future work.

Bidding Policies. Spot market prices are determined by a second price auction and have been modeled in prior work [61]. Numerous bidding strategies for individual spot markets to optimize the cost/performance of batch jobs exist [240, 210, 231, 234, 194, 198]. However, as we show, a simple bidding strategy of bidding the on-demand price is optimal for Flint. By focusing on the checkpointing and server selection, Flint is applicable to transient servers that do not permit bidding, such as GCE's Preemptible Instances that offer transient servers at a fixed price.

4.8 Flint Summary

The low price of transient servers is attractive for recent cluster-based in-memory data-parallel processing frameworks, since these frameworks need to cache large datasets in memory across a large number of servers. However, transient server revocations degrade the performance and increase the cost of these frameworks. In this chapter, we designed Flint, which includes intelligent, application-specific checkpointing and server selection policies to optimize the use of transient servers for data-parallel processing. In particular, Flint's policies support BIDI workloads that may be either batch or interactive. Our results show Flint enables a 90% cost saving compared to using on-demand instances and a slight decrease in performance of 2%.

CHAPTER 5

PORTFOLIO-DRIVEN RESOURCE MANAGEMENT FOR TRANSIENT SERVERS

The previous two chapters presented transiency-specific fault-tolerance techniques for running interactive and batch-interactive applications. The risk of transient server revocations can also be mitigated through smartly selecting transient servers.

This chapter presents *server portfolios* for selecting a heterogenous collection of transient servers with different costs and availabilities. Server portfolios enable construction of an "optimal" mix of severs to meet an application's sensitivity to cost and revocation risk. We implement model-driven portfolios in a system called ExoSphere, and show how applications can implement custom policies for handling transiency.

5.1 Motivation and Overview

Transient servers typically incur a fraction of the cost of their regular ("on-demand") server counterparts, making them a popular choice for running large-scale data-intensive jobs involving tens or hundreds of servers due to their low cost. However, revocations of some, or all, of an application's transient servers can seriously disrupt its performance or cause it to fail entirely.

Despite the low cost of transient servers, effectively using them remains challenging. On some cloud platforms, such as Amazon EC2, transient servers have dynamically varying prices that fluctuate continuously based on supply and demand. In addition, the availability of transient servers (in terms of their mean time to revocation), can also vary significantly across server configurations and based on changing market conditions. Thus, it is challenging for a cloud application to judiciously select the most appropriate server configuration based on historical pricing or availability data to satisfy its needs. The problem is compounded by the large number of transient server configurations available to applications—there are over 2,500 distinct types of transient servers in EC2 and over 300 in Google's cloud platform.

The preemptible nature of transient servers also imposes new requirements on cloud applications. Specifically, applications must determine whether and how to save their computation's intermediate state to gracefully handle server revocations, which are akin to server failures. Further, they must also define recovery policies to determine how to re-acquire new transient servers upon revocation, and how to restore state and resume their computation on these new servers. Different applications, such as Spark, MapReduce, and MPI, also have different tolerances to revocations, and require different application-specific mechanisms to handle revocations and their subsequent recovery. However, prior research has largely focused on separately designing custom modifications to support transiency for each narrow class of application [176, 221, 142].

To address this problem, we introduce a model-driven framework called *server portfolios*. Portfolios represent a virtual cloud cluster composed of a mix of transient server types with a configurable cost and availability depending on the application's tolerance to revocation risk and price sensitivity. Our portfolio model derives from Modern Portfolio Theory in financial economics [148, 143], which enables investors to methodically construct a financial portfolio from a large number of underlying assets with various risks and rewards.

The flexibility and explicit risk-awareness that portfolios offer is not provided by prior work on transient server selection. A majority of prior work [91, 197, 73, 226] on transient servers solves the problem of choosing *one* server type (among the hundreds that cloud providers offer). Choosing *multiple* server types has received relatively little attention, and mostly relies on application-specific, ad-hoc approaches to optimize either cost or revocation-risk [176]. In contrast, portfolios are a *general* technique that allow server selection for a wide range of risk tolerances and application preferences. This diversification of servers is crucial because it reduces revocation risk. If the server markets are not correlated, then a revocation in one market may not necessarily affect the other—thus allowing distributed applications to continue running on the available markets.

We use portfolio modeling as part of the design of an application-independent framework for supporting transiency, called ExoSphere. ExoSphere uses portfolio modeling to expose virtual clusters of transient servers of different types to different applications. Along with portfolio modeling, ExoSphere also supports custom application-specific policies for handling transiency. In particular, ExoSphere adopts an Exokernel approach [93] by exposing a set of basic mechanisms that are common to all transient server environments. These mechanisms can be used by applications to design policies for handling revocations, saving state, and performing recovery. Thus, ExoSphere's mechanisms simplify modifying distributed applications to effectively run on transient servers.

5.2 Server Portfolios

A key factor in making effective use of transient servers is judiciously choosing the most appropriate server configuration for each application. Due to their preemptible nature and variable pricing, picking the "correct" server configuration is surprisingly complex in today's cloud platforms due to the following reasons:

Large number of potential choices. A typical cloud platform offers a large number of transient server markets. Amazon's EC2 cloud offers 2500 distinct markets, while Google Cloud Platform offers more than 300 markets for predefined machine types alone. Assuming an application imposes a certain base requirement on the desired per-server compute and memory capacity, it must still choose from a large number of feasible configurations.

Pricing idiosyncrasies. Cloud operators such as Amazon EC2 use demand-supply driven pricing to price their spot servers [41, 61]. Each server type has different demand-supply characteristics, and this can lead to some interesting idiosyncrasies, which can be seen in Figure 2.2. In this example, the `m3.medium` in availability zone `a` has the most stable prices, `g2.2xlarge` in the same availability zone has a lower average price but high variance, and the `m3.medium` in availability zone `b` has higher price than in zone `a`. The `g2.2xlarge` price spikes are not correlated with the other two servers. The example shows that smaller servers may occasionally be more heavily discounted than smaller servers, and that identical servers in two availability zones may also be discounted differently.

Importantly, choosing a server configuration based on price alone may yield sub-optimal results. For instance, server configurations with cheap prices may also see higher customer demand and consequently higher volatility and frequent revocations. Frequent revocations add substantial overheads to an application in terms of increasing checkpointing costs and adding recovery overheads. Instead, sometimes the choice of a slightly more expensive server configuration that sees a lower revocation rate may be a better choice and yield lower overall costs.

Revocation rates may also not be related to average prices—neither the willingness to pay higher prices (by using a higher bid) nor choosing higher priced configurations necessarily yield lower revocation rates [178]. It is not practical to expect applications to analyze detailed price histories and volatilities across hundreds of transient servers when choosing a server type.

Due to the challenges above, cloud providers such as Amazon have begun offering server selection tools. Amazon SpotFleet [22] automatically replaces revoked servers. However, SpotFleet provides a limited choice in terms of the combinations of server configurations it offers, and does not alleviate many of the challenges above. While it enables applications to specify their combination of server configurations, it is up to applications to choose their specific server configuration. While tools, such as Amazon Spot Bid Advisor [20], may help users in selecting servers based on price, they expose only coarse volatility information, e.g., low, medium, or high.

5.2.1 Reducing Risk through Diversification

We use two key insights for reducing server revocation risk for a distributed batch-oriented application. The *first insight* is to choose servers whose mean time between revocations (MTTR) significantly exceeds the expected job length. For example, if a batch job has an expected length of 5 hours, then it will have a higher probability of completion if it runs on a server with MTTR of 100 hours, when compared to running it on a server with MTTR of 10 hours. Thus, choosing server configurations where the MTTR is much greater than the job length also increases the chances of a job completing without any revocations.

Each transient server configuration in a cloud platform represents a *market* with its own supply and demand conditions. If a parallel batch-oriented job chooses homogeneous servers from a single cloud market, then any revocation event will cause *all* servers to be lost simultaneously (in Amazon's EC2 spot market, if the spot price rises above bid price, then all the servers with that bid-price are revoked.). Our *second insight* for reducing the impact of concurrent revocations is to choose a heterogeneous mix of transient servers drawn from multiple markets.

Empirical analysis indicates that price fluctuations across markets are largely uncorrelated with each other (Figure 2.3). Thus, revocation events in one market may not cause revocations in certain other markets, since surging demand and revocations in one market will not impact available capacity in other independent markets. As a result, use of a heterogeneous mix of transient servers drawn from independent or weakly correlated markets can mitigate the impact of revocations—since revocations now only impact a fraction of an application's servers. This enables jobs to make forward, albeit degraded, progress on the remaining transient servers.

However, constructing such a heterogeneous mix of servers from multiple markets is not trivial. It involves selecting transient servers that are "cheap" and yield high savings compared to their on-demand counterparts, yet at the same time we must minimize the risk of simultaneous revocations—if all markets fail simultaneously, there is little value in diversification. Thus we must satisfy two objectives: pick markets to minimize cost *and* minimize their failure correlation. The large number of possible markets (>2,500 spot markets on Amazon EC2), means that achieving this dual objective is intractable with ad-hoc techniques [182] and heuristics [176] that past work on multiple transient server selection has used. We describe our solution to this multiple server selection problem using portfolio theory next.

5.2.2 Server Portfolios

Intelligent server selection is key to minimizing the frequency and magnitude of disruptions seen by applications running on transient servers. To address this problem, we present *server portfolios*, a new model-driven framework to create virtual clusters composed of a mix of transient server types which offer flexible costs and availability.

Portfolios enable ExoSphere to construct a mix of cloud servers tailored to application needs. Server portfolios draw inspiration from finance [148, 69, 143]. Intuitively, a financial portfolio involves creating a suitable mix of financial investments for an investor that are drawn from an underlying mix of assets such as stocks, bonds, etc. The goal is to construct a mix that matches the investor's tolerance for risk and reward. The risk tolerance dictates whether the portfolio contains a more risky mix of high-reward assets, or a mix of lower-reward but lower-risk assets.

Similarly, server portfolios comprise a mix of transient servers that are drawn from an underlying mix of all transient server markets. Like financial assets, transient server markets exhibit different price and revocation characteristics. Some markets may have low prices but higher revocation rates, while others have higher, more stable, prices with infrequent revocations. Consequently, depending on the risk tolerance of an application, server portfolio construction involves maximizing the risk-adjusted returns by designing an appropriate mix of server markets.

ExoSphere instantiates the model-driven portfolio mechanism to create virtual clusters for applications. At startup time, applications specify their aggregate resource requirements (CPU-cores and memory) in the form of a resource vector $\mathbf{r} = [r_{\text{cpu}}, r_{\text{mem}}]$, and their risk tolerance[1]. It then uses portfolio creation models and algorithms that are rooted in Modern Portfolio Theory [148, 143] to construct a mix of servers for the application, as discussed next.

5.2.3 Model-driven Portfolio Construction

We now present ExoSphere's portfolio model, which is based on Modern Portfolio Theory[2]from financial economics [69, 148, 143]. The goal in ExoSphere is to maximize *risk-adjusted returns* for each application, where the returns are the *cost savings* from using transient servers (over the on-demand prices), while risk is the application's tolerance to server revocation events.

Formally, ExoSphere finds a suitable mix of transient servers that maximize the risk-adjusted expected return given by:

[1]If available, the estimated job length can be provided, and only markets with MTTR >> job-length are considered.

[1]Modern Portfolio Theory was first proposed in 1952 [143] and remains the foundational basis for much of portfolio optimization in finance even today [148].

$$E[\text{Return}] - \alpha \cdot \text{Risk} \tag{5.1}$$

where $E[\text{Return}]$ is the difference between the cost of an on-demand server and the expected cost of the transient server. To formally define $E[\text{Return}]$, assume that the cloud platform offers servers in n distinct markets. Let D_i denote the on-demand price, and let $E[S_i]$ denote the mean of the transient server price. Then,

$$E[\text{Return}_i] = 1 - \frac{E[S_i]}{D_i} \tag{5.2}$$

Let $\mathbf{c}$ denote the vector representing the returns for all n markets, where $\mathbf{c} = [\text{Return}_1, \ldots, \text{Return}_n]$. Let x_i denote the fraction of servers from market i chosen in our portfolio ($0 \leq x_i \leq 1$). Then $\mathbf{x} = [x_1, \cdots x_n]$ denotes the portfolio allocation vector, and $\mathbf{x}^T$ is its transpose. The effective expected return of a portfolio is then:

$$E[\text{Return}] = \mathbf{c}\mathbf{x}^T \tag{5.3}$$

The parameter α (in Equation 5.1) denotes the *risk-averseness* of the application or user. A low value of α indicates that the application places lower emphasis on avoiding server revocation risk. Conversely, a high value of α indicates that an application is highly risk-averse, and is willing to incur an extra cost for this. We also use the term risk tolerance to mean the inverse of risk-averseness.

To capture risk, we draw an analogy with financial portfolio selection, where investments are chosen such that their prices are not correlated. The rationale is that if one asset (say, a particular stock) sees a decline in price, then the other assets (e.g., a bond) are unlikely to see a concurrent decline. This way, we avoid large declines in the overall portfolio value.

In our case, we wish to select server markets with independent revocation events—thus if there is a revocation in one market, others will not see a concurrent revocation. This reduces the total number of allocated servers that are revoked. To do so, we define a covariance matrix $\mathbf{V}$ that captures pairwise correlations between all pairs of markets. V_{ij} is the correlation between markets i, j, and captures their simultaneous revocations. Higher values indicate that the two markets are highly correlated in their revocations, and the chances of closely spaced revocations are greater. We use this formulation to define the revocation risk of a portfolio as:

$$\text{Risk} = \mathbf{x}\mathbf{V}\mathbf{x}^T \tag{5.4}$$

Our portfolio construction problem can then be formulated as the following optimization problem:

$$\begin{aligned} \text{Maximize:} \quad & \mathbf{c}\mathbf{x}^T - \alpha\mathbf{x}\mathbf{V}\mathbf{x}^T \\ \text{Subject to:} \quad & \sum_1^n x_i = 1 \\ & \mathbf{x} \geq 0 \end{aligned} \tag{5.5}$$

We can solve Equation 5.5 for a wide range of risk-aversion parameters (α) to compute the lowest-cost portfolios for any given risk. The expected returns and revocation risks of these portfolios are shown in Figure 5.1, which shows the expected cost savings for a range of revocation risks. As the revocation risk is reduced, so is the cost savings. We also see from Figure 5.1 that expanding the candidate-set from r3 servers in the US-east-1 region to *all* the servers in the US-east-1 region results in a 1% increase in savings, and a 20-50% reduction in revocation risk. This occurs because a larger set of candidate markets both allows more freedom in choosing markets, and increases the number of markets with low correlations.

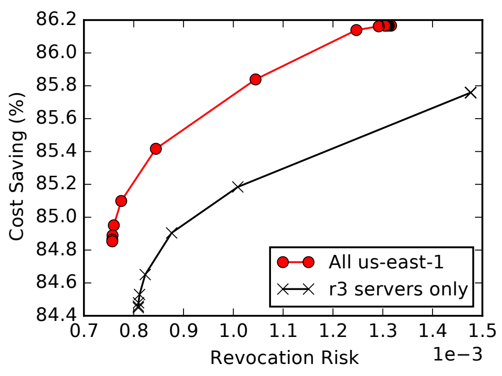


Figure 5.1: Cost savings and revocation risks of portfolios with different risk-averseness. Choosing a portfolio from a larger collection of servers (all US-east-1 vs. only r3-type servers) results in higher returns at lower risk.

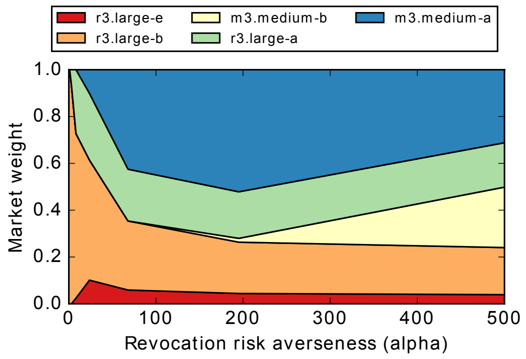


Figure 5.2: The effect of risk-averseness in portfolio diversity. A single market (`r3-large-b`) dominates the portfolio when $\alpha = 0$, but the portfolio's diversity increases with increasing risk-averseness.

The effectiveness of the risk-averseness parameter can also be seen in Figure 5.2, which shows the distribution of servers in portfolios with different risk-averseness parameters. We can see that portfolios become more diversified as the risk-averseness increases.

Constructing the covariance matrix. The covariance matrix $\mathbf{V}$ captures the pairwise correlation between markets. Our formulation allows multiple types of correlation to be used. The different correlation functions (and their corresponding $\mathbf{V}$ matrices) allows ExoSphere to adjust the portfolios to the users' perceptions of risk.

The first and most basic form of correlation is simply the correlation between spot prices. In the case of Amazon EC2, we can use price histories of spot servers, which are publicly available, to compute the mean returns and the covariance matrix. That is, we compute the pairwise covariances by using spot prices to capture revocation events and using the standard covariance formulation. Let X_t, Y_t denote the spot price of markets X, Y respectively at time t. Then the standard definition of covariance applies:

$$V_{XY}^{price} = \frac{1}{T}\sum_{t=1}^{T}(X(t) - E[X])(Y(t) - E[Y]) \tag{5.6}$$

This captures the correlation between the prices in different markets, and is useful metric for price sensitive users, since they may not want prices of all markets to increase simultaneously.

In transient server environments, simultaneous market revocations can lead to disruption of application availability or performance. To capture simultaneous revocation risk between two markets, we use the likelihood of simultaneous revocation. We again use the spot price traces to find simultaneous revocations between markets—we say that two markets have a simultaneous revocation if servers in those markets get revoked within a small time window (5 minutes). This allows us to define the entries in the similarity matrix using the probability of simultaneous revocations.

$$V_{XY}^{revoc} = \text{Probability of simultaneous revocation of X,Y} \tag{5.7}$$

Lastly, we also provide a hybrid risk formulation that captures both simultaneous revocations and changes in prices. We first transform the spot prices to capture revocation events, and then compute the covariance of these transformed prices. Since a spot server is revoked if its price increases above the bid price, we capture the revocation and unavailability by setting the price to the maximum spot price. For a given market, if we are given a trace of the spot prices S and the bid price B, we define the transformed prices as:

$$\begin{aligned} S'(t) &= S(t) && \text{if } S(t) < B \\ &= \text{Maximum spot price} = 10 * \text{On-demand price} && \text{otherwise} \end{aligned}$$

This ensures that we impose a very high uniform penalty when there are revocations. Because we set the prices to the maximum spot price during a revocation, this results in a high correlation if two markets fail at near the same time. The final step is to compute the covariances between pairwise markets (after applying the above price transformation) by using the standard covariance formulation : $V_{ij}^{hybrid} = \frac{1}{T}\sum_{t=1}^{T}(S_i'(t) - E[S_i'])(S_j'(t) - E[S_j'])$.

Our portfolio model (Equation 5.5) formulation closely mirrors portfolio construction found in Modern Portfolio Theory. Infact, it is a quadratic convex optimization problem.

First, we note that the covariance matrix $\mathbf{V}$ is positive semidefinite. $\mathbf{V}$ is a matrix of covariances that are always non-negative. To establish semidefiniteness, it is enough to show that for any vector $\mathbf{a}$, $\mathbf{a^T V a} \geq 0$. By using the definition of co-variance, we get:

$$\begin{aligned}\mathbf{a^T V a} &= \mathbf{a^T} E[(x-\mu)(x-\mu)]\mathbf{a} \\ &= E[(\mathbf{a^T} \cdot (x-\mu))((x-\mu)\cdot \mathbf{a})] = E[((x-\mu)\mathbf{a})^2] \geq 0\end{aligned}$$

Since the covariance matrix is positive semidefinite, $\mathbf{x^T V x}$ is strictly convex, and thus the problem formulation in Equation 5.5 is a quadratic convex optimization problem [68, 148]. The formulation can be solved by an off-the-shelf convex solver, such as `cvxopt` [36]. This allows us to exactly solve the portfolio modeling problem and get portfolios that maximize the risk adjusted returns, without having to rely on heuristics or approximation. For Amazon EC2 spot instance portfolios, we use the publicly available time series of spot prices for each spot market. We can then compute the average spot price for each market and can get the returns vector $\mathbf{c}$, as well as the covariance matrix $\mathbf{V}$.

5.2.4 Server Allocation using Portfolios

ExoSphere considers the risk-averseness requirements of the application along with the computing resource requirements. Based on these requirements, ExoSphere first constructs a portfolio of resources on cloud servers, and then allocates the resources to the applications in the form of containers on these servers.

Applications submit CPU and memory resource requests in the form a resource-vector $\mathbf{r} = (r_{\text{cpu}}, r_{\text{mem}})$, and their placement constraints. The placement constraints comprise primarily of the risk-averseness factor $\alpha \in [0, \inf)$, and any server preferences they might have (gpu-enabled servers only, no small servers, etc).

We then construct portfolios based on these requirements, which gives us the weights for each market in the form of a weight-vector $\mathbf{x}$. These weights represent the fraction of resources that must be allocated in a market. For each market i, we compute the CPU and memory resources that must be allocated in that market by multiplying the portfolio-weight of that market (x_i) by the resource-vector ($\mathbf{r}$). ExoSphere then determines the actual number of servers to allocate in market n_i based on the CPU and memory capacities of the servers in that market (CPU_i, MEM_i) as follows:

$$n_i = \max\left\{\frac{x_i r_{\text{cpu}}}{\text{CPU}_i}, \frac{x_i r_{\text{mem}}}{\text{MEM}_i}\right\} \tag{5.8}$$

We take the maximum of the servers required to satisfy both the CPU and memory requirements so that the application's resource allocation meets or exceeds the requirements in all resource dimensions. This approach can be extended to other resource types (disk/network bandwidth, etc.). Upon deciding the number of servers that an application needs in each market, ExoSphere then requests new servers (with bid price set as the on-demand price) from the cloud operator. ExoSphere also allows applications to dynamically adjust their resource requirements, which is useful for auto-scaling. Applications can adjust their CPU and memory requirements ($\mathbf{r}$) at any time, and ExoSphere adds or removes servers from each market.

5.2.5 Statistical Multiplexing of Servers

In the above described server allocation policy, it may be possible for an application's resource requirements to be smaller than the resources offered by the server portfolio. This can occur because of two reasons. The first reason is that ExoSphere maximizes the (risk adjusted) cost-savings relative to the on-demand price, which may require selection of larger servers. Such price inversions are common in EC2 spot markets, and can occur if smaller transient servers have a larger demand compared to their larger counterparts. The second reason for surplus resources in a portfolio is that ExoSphere's allocation ensures that sufficient servers are available to meet the demands across all resource types i.e., both CPU and memory. For example, an application requesting 2 CPUs and 10 GB memory may be allocated a portfolio of 2 `m3.large` servers each having 2 CPUs and 7.5 GB memory, resulting in 2 free CPUs and 5 GB of free memory across both the servers.

ExoSphere reduces the surplus unused resources in a portfolio by relying on statistical multiplexing. The key idea is that transient servers can be multiplexed across multiple portfolios. This allows multiple applications to share the servers in their virtual clusters such that the free and unused resources of a server can be used by other applications. In addition to increasing server utilization, this also reduces costs, since the cost of transient servers is also proportionally shared between the applications sharing a server.

ExoSphere's statistical multiplexing, also referred to as the shared-cluster policy, works as follows. We use the portfolio modeling and creation process described earlier. This gives us the portfolio weights vector $\mathbf{x}$, indicating the weights of each market in the portfolio. The application's actual resource requirements ($\mathbf{r}$) are first met by trying to use as many surplus resources as possible across all the servers in a given market. That is, for each market in the application's portfolio, we first find surplus resources on existing servers in that market, and then request the cloud servers required to meet the unmet resource demand in that market instead of all n_i servers (Equation 5.8). Finding surplus resources involves finding servers such that their allocated-resource vector is less than the available resources. ExoSphere uses the "best-fit" policy: it sorts the servers in each market in descending order of their free resource availability, and then proceeds to allocate resources (as containers) from these servers until either all free resources in the market are allocated or if the application's resource requirements in that market are satisfied.

Finally, we note that this multiplexing of servers is only effective if there exist multiple applications to exploit the free resources, and if there is a steady stream of applications leaving and entering a system. We evaluate the cost effectiveness of this multiplexing scheme in Section 5.5.3. In the next section, we describe how applications can use the API provided by ExoSphere to design and implement their own transiency-specific policies.

5.3 ExoSphere Design and API

In addition to supporting the portfolio abstraction, ExoSphere provides a number of key mechanisms to support the execution of batch-oriented applications on transient servers. ExoSphere's design is based on the Exokernel philosophy, where it provides a small set of mechanisms to make an application transiency-aware, and leaves the design of transiency-specific policies to the application.

Unlike much of prior work on running applications on transient servers, ExoSphere gives applications the ability to define their own policies for handling revocations. This allows applications to define policies to suit their fault tolerance requirements, and also allows more efficient fault tolerance. For example, using application-level fault tolerance such as application-level checkpointing [176] may significantly reduce the overhead of checkpointing compared to application-agnostic system-level checkpointing.

ExoSphere uses a two-level architecture (Figure 5.3), where ExoSphere provides the portfolio abstraction and transiency-specific "up-calls" to the applications, which may use them to implement their own policies. Associated with each application is a job-manager, which communicates with ExoSphere to implement these policies.

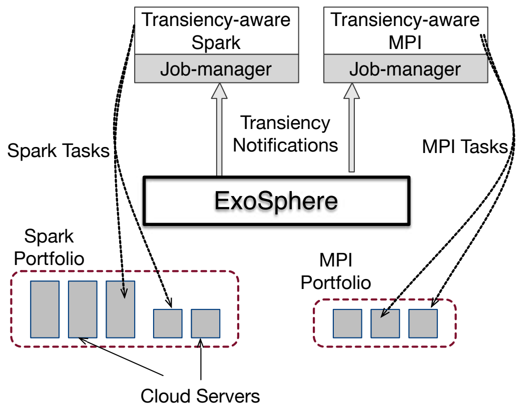


Figure 5.3: ExoSphere's design architecture. The job managers for each application implement the resource allocation requests and the fault tolerance policies.

Down-Calls
`portfolio = reqResources(cpu, mem, alpha, job-len)`

Up-Calls
`mttrList = portfolioMTTR(portfolio)`
`Notify(`**`event`**`, serverList)` **`event`** `= {hardRevocation, softRevocation, priceThreshold}`

Figure 5.4: ExoSphere API. Applications allocate portfolios by using down-calls, and receive transiency-specific notifications using the up-calls provided by ExoSphere.

Given any vanilla batch-oriented application, converting it into a transiency-aware variant of that application involves defining three policies: a (i) *portfolio policy*, which specifies its resource needs and risk tolerance, (ii) *fault-tolerance policy*, which specifies whether and how the application state is saved to deal with potential server revocation, and (iii) *recovery policy*, which specifies the policy to replenish servers upon a revocation event and to resume the application after recovering saved state.

The portfolio policy is implemented using ExoSphere's portfolio abstraction described in the previous section. To implement a broad range of fault-tolerance and recovery policies, ExoSphere supports three key mechanisms via the up-call API described in Figure 5.4:

Exposing the Portfolio MTTRs: Since cloud platforms only expose transient server prices but not revocation statistics, ExoSphere provides MTTR information immediately after portfolio creation and periodically (every 5 minutes) via the `portfolioMTTR` upcall. ExoSphere provides the mean MTTR of an application's portfolio, as well as the specific MTTRs of the individual transient servers within the portfolio. An application can use this knowledge of how frequently a portfolio server is likely to be revoked to tune how frequently to save its state.

Hard revocation signals: ExoSphere tracks a cloud platform's termination warning for one or more servers and signals the application about imminent revocation of these servers via the `Notify(hardRevocation)` upcall.

Soft signals: Soft signals are provided to signal specific conditions to the application. ExoSphere currently supports two types of soft-signals: (i) price threshold signals and (ii) soft revocation. Price threshold signals are used by price sensitive applications to track when the price exceeds a specified threshold, by using the `Notify(priceThreshold)` upcall, which it can use to relinquish servers and restart computation later to avoid going over budget. Soft revocation signals are upcalls from ExoSphere when it detects early signs of revocation—they serve as an early warning (but not a guarantee) that revocation may occur in the near future (e.g., when the signature of a price spike is detected). The soft-revocation notification provides more time for applications to take action (e.g. checkpoint, migrate, etc).

Next, we describe how these mechanisms can be used to create transiency-aware versions of three common batch-oriented applications with modest effort.

5.3.1 Data-parallel Application: Spark

The low-cost of transient servers makes them very appealing for running data-parallel data-intensive frameworks like Hadoop, Spark, Naiad, etc. Such frameworks run two broad classes of jobs. Traditionally, they run data intensive batch jobs that perform computation over large amounts of data in parallel. These frameworks also support batch-interactive [176] jobs such as SQL queries, interactive machine-learning, or streaming analytics, which have lower-latency requirements.

We use Spark [228] as a representative data-processing framework to build a transiency-aware application using ExoSphere. Spark is a popular data-parallel framework and supports both batch and batch-interactive computation. Spark performs data transformations on in-memory distributed datasets called Resilient Distributed

Datasets (RDDs [228]). The loss of servers leads to the loss of the in-memory RDD partitions, which can lead to recursive recomputation. While batch workloads can tolerate the delay due to recomputation, such recomputation significantly increases the latency for batch-interactive workloads, such as SQL queries or REPL-based environments.

Portfolio Policy. The portfolio policy for a Spark cluster depends on the Spark workload characteristics. Purely batch workloads are more disruption tolerant and may choose to optimize for lower cloud costs. Thus, when instantiating a Spark cluster for a batch workload, a low risk-averseness portfolio (low α) can be requested. Doing so will skew the portfolio towards lower costs. In contrast, batch-interactive and streaming workloads are highly risk-averse, and thus request highly diversified portfolios with a high α to reduce the performance impact of revocations (but at potentially higher cost).

Fault-tolerance Policy. Spark includes a built-in RDD checkpointing mechanism, which serializes RDDs to stable storage. However, Spark leaves it to the application to decide which RDD to checkpoint. A checkpoint operation imposes significant overhead, since it causes a substantial amount of in-memory data to be written to disk.

Designing a fault-tolerance policy for our transiency-aware version of Spark is straightforward using this checkpoint operation—we periodically checkpoint recent RDDs. Due to the overhead imposed by checkpointing, the checkpoint interval must be carefully chosen. Since ExoSphere exposes the MTTR of the portfolio, we can use it to set the checkpointing interval to $\tau = \sqrt{2 \cdot \delta \cdot \text{MTTR}}$, where δ is the time it takes to write a checkpoint to disk, and the MTTR is Mean Time To Revocation of the portfolio. This expression follows directly from a classic result in fault-tolerance [82] and has been used in other Spark-based systems such as Flint [176]. To implement this policy, we modify the Spark job-manager to periodically checkpoint RDDs, and use saved checkpoints when resuming after a revocation. The pseudo-code for the Spark periodic checkpointing is below:

```
while(true):
  mttr = portfolioMTTR(portfolio).get()
  tau = math.sqrt(2*mttr*delta)
  sleep(tau)
  for rdd in job.rdds.SortBy(``age'')[0] :
      rdd.checkpoint()
```

Recovery Policy. The recovery policy comprises of two parts: how to recover the application upon a revocation event, and how to resize the cluster to handle lost servers. Upon receiving a hard revocation signal from ExoSphere, the job-manager in our transiency-aware Spark triggers recomputation from the last saved RDD checkpoint. The decision on whether to replenish lost servers depends on the job progress and workload characteristics. Due to Spark's in-built fault-tolerance mechanisms, jobs are able to continue execution on remaining servers. However, continuing in this degraded mode increases job completion times (even when resuming from a saved checkpoint), due to the potential of spilling RDDs to disk, or reduction in the size of the RDD cache.

For pure batch jobs, we can use job progress (by comparing against estimated job-length), and intelligently decide whether to replenish (e.g., replenish if job-progress $< 70\%$). For batch-interactive or streaming workloads, an immediate replenishment policy is always preferred due to the latency requirements.

Comparison with other Spark-based systems. Flint [176] and TR-Spark [221] are two recently proposed transiency-aware versions of Spark. Both systems use an application-level fault-tolerance and require significant complex modifications in Spark to embed new mechanisms and policies. Our version uses ExoSphere abstractions and mechanisms to implement similar policies. We model our version on Flint's design.

However, while Flint requires 3000 lines of code changes [176] to Spark, ExoSphere requires adding only 400 lines, and benefits from separating issues such as portfolio construction out of the application. ExoSphere also allows a richer server selection policy, since portfolios can be tailored to the workload's risk tolerance.

TR-Spark [221] is another attempt to make Spark transiency-friendly, and changes task-scheduling in Spark to avoid scheduling jobs to nodes that face imminent revocation. These changes can also be supported by ExoSphere, since TR-Spark also uses MTTR information. Mostly, ExoSphere's Spark benefits from separation of concerns and requires less changes to the application (Spark) in order to run on transient servers.

5.3.2 Parallel HPC Application: MPI

Message Passing Interface (MPI) is the predominant framework for scientific and high-performance computing. MPI jobs tend to be parallel compute-intensive tasks and their large degree of parallelism can benefit from running on low-cost transient cloud servers [142]. However, unlike Spark, MPI's message-passing model is highly intolerant to revocations. In particular, revocation of a single server can cause the entire MPI job to fail.

Portfolio policy. Since even a single server revocation requires the entire job to be restarted (from the beginning or from a checkpoint), a policy that attempts to limit failures to a fraction of the servers is not adequate—any revocation, whether it is one server or all servers, has the same impact. Thus, stability is more important than server diversity, i.e., choosing servers with MTTR >> the job length, which reduces the probability of revocation, is more important than portfolio diversity. Thus, MPI's job-manager requests portfolios by specifying the expected job-length, and specifies a *low* risk-averseness parameter to ensure selecting high-MTTR servers.

Fault-tolerance policy. Many MPI platforms, such as OpenMPI [35], support checkpointing. In such cases, the MPI job can periodically checkpoint its state similar to ExoSphere's Spark. If checkpointing is not supported or is undesirable, then no fault-tolerance policy is necessary and the job is simply restarted from the beginning.
Recovery policy. Due to the inability of MPI jobs to continue computation after partial failures, the immediate replenishment policy must be used to restore the cluster to its original size upon a failure of one or more servers. Once replenished, the job is restarted from the most recent checkpoint or the beginning. The pseudo-code of the revocation-handling policy for MPI is shown below:

```
def Notify(hardRevocation, servers) :
  Kill_MPI_Job()
  portfolio = reqResources(cpu, mem, alpha=0)
  Start_MPI_Job(portfolio)
```

The ExoSphere MPI version required a modest effort of 50 lines of code for the portfolio and recovery policy.

5.3.3 Delay Tolerant Application: BOINC

Volunteer computing frameworks such as BOINC [49] are an example of "embarrassingly parallel" workloads that are delay-tolerant and do not have strict deadlines.
Portfolio policy. Since reducing cost is more important than mitigating failures, a low-to-moderate risk-averseness parameter (α) can be specified when constructing a portfolio for BOINC. For highly price-sensitive workloads, a low value may be used, but it risks losing a large fraction (or all) servers. Use of a moderate value provides some diversification, which allows progress to be made when part of the portfolio is revoked.
Fault-tolerance policy. Typically no fault-tolerance mechanisms are needed, since if a server is lost in a volunteer-computing scenario, the task is restarted. In some cases

with long-running tasks, a lazy-checkpointing policy can be used, which checkpoints the task after receiving a soft or hard revocation warning. Soft warnings increase the chances of completing the checkpoint, since a lazy-checkpoint may not complete within the hard-warning duration (2 minutes on EC2, 30 seconds on GCP).

Due to its price sensitive nature, BOINC can use soft signals to set a price threshold and upon receiving a notification of rising prices, can voluntarily relinquish servers and wait for the price to reduce to maintain a budget.

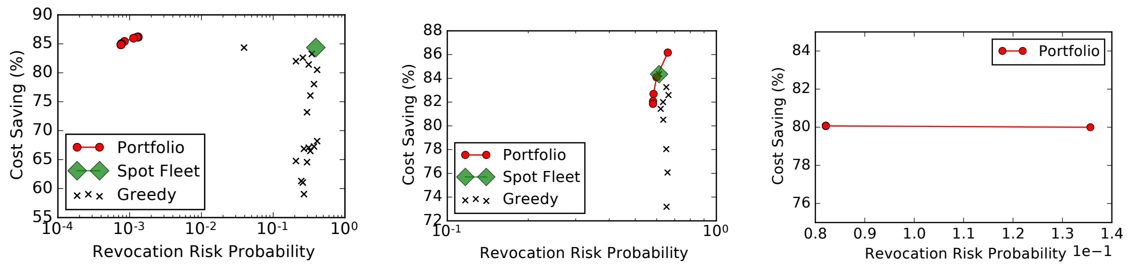
Recovery policy. The price sensitive nature implies that immediate replenishment of lost servers is not strictly necessary. The BOINC job-manager can monitor the price of portfolios offered by ExoSphere to wait until prices drop. Tasks that were affected due to server revocations are simply queued on other remaining nodes and are restarted (from the beginning or from the last checkpoint).

The transiency-aware BOINC required about 200 lines of additional code—most of which pertain to the implementation of lazy checkpointing and recovery.

5.4 ExoSphere Implementation

While ExoSphere's design and its portfolio mechanism are general, we implement ExoSphere using Mesos [113]. The choice of an existing cluster manager for implementing ExoSphere is motivated by two factors. First, Mesos employs an Exokernel [93] like philosophy of letting higher-level applications implement their own specific resource allocation policies. Thus, ExoSphere's abstraction and interfaces are a natural fit into the architecture of such cluster managers. Second, enhancing a popular cluster manager such as Mesos to support transient cloud servers yields a transiency-aware cluster-manager that can find broad use and adoption in today's cloud platforms.

ExoSphere is built using Mesos v0.27 and cloud native APIs. Our prototype has two key components, the ExoSphere master and the application job-managers. The ExoSphere master is implemented in 5000 lines of C++ code by extending the



(a) Today's EC2 (b) Synthetic highly-correlated EC2 markets (c) Google preemptible instances

Figure 5.5: *Portfolios for various market scenarios*

Mesos master. The master implements two key components: portfolio-based resource allocation and the application-facing API shown in Figure 5.4. Our prototype currently supports EC2 spot instances, which offer a rich collection of servers and publicly available price history. We also have proof-of-concept support for Google's preemptible instances (where inferring availability information is more challenging).

ExoSphere requires applications to implement a job-manager using ExoSphere's API to design their own portfolio-creation, fault-tolerance, and recovery policies. Requiring applications to implement their own job-manager is increasingly common in modern cluster managers—our job-manager is equivalent to Application-Schedulers in Mesos and Application-Masters in Yarn. The master communicates with the job-managers using the existing Mesos RPC and HTTP APIs. This allows existing application-schedulers (written for Mesos) to be used and augmented with the transiency-specific functions provided by ExoSphere. The only requirement for running existing Mesos applications on ExoSphere is that they handle the revocation hard-warning notification, which can be implemented in either C++, Java, or Python.

Applications (via their job-managers) make requests for resources and their portfolio requirements via the existing Mesos `requestResources` RPC, which the ExoSphere master intercepts and handles. The resource allocation involves portfolio construction, server-packing, and creating and keeping track of the cloud servers.

Portfolio-based allocation. In order to construct portfolios, we use historical price traces. Amazon publishes the past three months of spot price traces (available using the EC2 `describe-spot-price-history` API). We periodically collect the price history for all markets, and compute the mean spot price, as well as the covariance matrices for various risk functions. Once these have been computed, ExoSphere solves the quadratic convex optimization problem using the Python `cvxopt` solver, which takes under 1 second for the 250 us-east-1 markets and under 25 seconds for the 2500 global markets. The portfolios for various risk-averseness factors are precomputed and cached, and this reduces the computational overhead of portfolio construction even further. In the absence of any server-type or job-length constraints, portfolio construction usually only involves a simple look-up/search in the portfolio cache.

ExoSphere does explicit, fixed resource allocation, and does not use Mesos's Dominant Resource Fairness allocator. Once the application terminates or voluntarily relinquishes its resources, its servers are placed on a free-list of servers for a short duration ($2\times$ allocation latency), instead of immediately terminating them. Similar to anticipatory scheduling, holding on to recently relinquished servers in the free-list speeds up the allocation of servers for future applications, since launching transient servers takes a few (~5) minutes.

New cloud servers are requested using the standard EC2 APIs, and are started with either the application provided disk-image (containing the required application dependencies), or a default image (AMI) which has a few common applications installed. We assume that most applications will use S3 or EBS for storing data, since the content of local disks is lost upon server revocation. The resources on cloud servers are offered to the applications using the Mesos abstraction of resource offers.

ExoSphere's portfolio-based policy may over-allocate resources, which can lead to idle resources on some servers. For example, an application requesting 2 CPUs is allocated a cloud server with 4 CPUs results in 2 surplus CPUs. To increase cluster utilization and reduce costs, ExoSphere also implements a server packing policy as an optimization, which first tries to meet resource demands of the application (in each market) from the idle resources on the servers in that market. For this, we use a simple first-fit approach to allocate resources. Note that applications run inside containers (e.g., Mesos executors), which provide security and performance isolation. Nevertheless, applications which do not wish to face the potential interference because of other co-resident applications can still request private cloud servers not shared with other applications.

ExoSphere Upcalls. The ExoSphere master also interacts with the servers and the cloud provider in order to issue transiency-specific notifications. Revocation hard-warnings are first detected by the servers, which then inform the master, which relays them to the applications via the Mesos `inverseOffers` API, which includes a list of affected servers/containers and the remaining time until termination. Soft revocation warnings are provided by monitoring the state of each server, and notifying the application if it reaches the `marked-for-termination` state. Additionally, the master can also bid much higher than the on-demand price and monitor for price increases to increase the soft-warning duration. Price notifications are used by applications to know if the price of their portfolio has increased above a threshold. The ExoSphere master uses the `describe-spot-price-history` EC2 API to continuously monitor prices of all the active markets and delivers the notification if the price crosses the threshold.

5.5 ExoSphere Evaluation

Our experimental evaluation focuses on answering two key questions: i) What is the effectiveness of the portfolio abstraction in reducing cost and revocation risk? and ii) What is the impact of different policies for handling revocations and with different risk tolerances? We evaluate ExoSphere on EC2, and also show results for Google

Cloud Platform (GCP). We use spot price traces over a six month period (Apr-Sept 2015) for evaluating portfolios, and restrict ourselves to a single region (us-east-1), since many applications have geographic locality constraints that prevent using servers from multiple regions. We evaluate the performance of transiency-aware variants of Spark, MPI, and BOINC on ExoSphere.

Spark. We use the Spark version modified to work with ExoSphere, and use Amazon S3 for storing input/output data and RDD checkpoints. We use a combination of batch and low-latency workloads for Spark. We use KMeans, an iterative machine-learning algorithm with ∼16GB of input data as a batch workload. For the low-latency, interactive scenario, we use TPC-H database queries served by a Spark application. Spark supports SQL queries by translating them into equivalent RDD operations, with each query akin to a short running job.

MPI. We use MPICH [34] v2.7, which supports Mesos. We use MPI as an example of a "rigid", transiency-agnostic application, which responds to revocations by requesting new servers and restarting its job. We use the NAS parallel benchmark [53] as an MPI workload.

BOINC. We use BOINC as a delay-tolerant bag-of-tasks distributed application. We configure BOINC to run fixed-length CPU-intensive embarrassingly parallel tasks.

5.5.1 Portfolio Construction

We first examine the risk-return tradeoff in portfolio construction, and compare it to other server selection approaches. A baseline approach to server selection in multiple markets is the greedy strategy which picks server-types offering the lowest average spot price. To select multiple (k) markets, the greedy approach picks the top-k cheapest server-types for use by the application.

Figure 5.5a shows the expected cost savings (compared to using on-demand servers) and the expected revocation risk for both the greedy approach and our portfolio

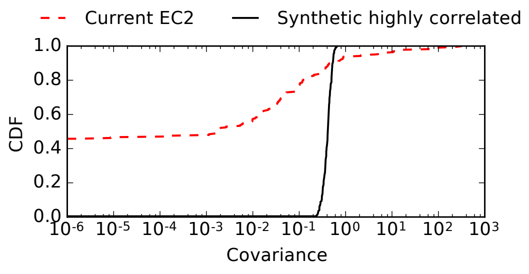


Figure 5.6: CDFs of covariances.

	Checkpointing	Restart
Eager-Replenish	+7%	+32%
No Replenishment	+11%	+54%

Table 5.1: Increase in running time of Spark (KMeans workload) due to a revocation event.

abstraction. Since there is no explicit way to specify risk-averseness in the greedy approach, we use a cruder "number of markets" as the diversification criterion, and divide servers among all markets equally. Each point in Figure 5.5a for the greedy scenario refers to a different number of markets picked. The *best* greedy portfolio approaches the cost-savings offered by the portfolio approach, but yields a much greater revocation risk (by 50×). Compared to portfolios, other greedily constructed selections offer upto 40% less saving at 100× more revocation risk.

We also compare against Amazon Spot Fleet [22], which provides a risk-agnostic mix of servers. Given a set of spot server types, Spot Fleets are constructed by either of two policies: the lowest-cost policy picks the server with the lowest spot price; and the diversified policy equally distributes servers among all chosen markets. The diversified policy is thus equivalent to our greedy policy when all servers in a region are considered. The result for the lowest-cost Spot Fleets is also shown in Figure 5.5a. The Spot Fleet has similar cost to our portfolio approach, but has almost 100× higher revocation risk for the single-market lowest-cost policy.

In practice, a system involving greedy server selection would have to iterate over all market sizes and pick the best performing greedy selection [176], and would have no explicit way to control the revocation risk of the selection.

So far we have seen the expected behavior of portfolios in EC2's spot market. We now explore their behavior in other scenarios. Figure 5.6 shows the CDF of the covariances of the EC2 spot markets, which shows that there is a large number of extremely low-correlation markets with a few highly-correlated markets. If usage of spot instances and the portfolio-approach were to increase, it is possible that the increased demand for the servers in the "uncorrelated" markets would increase prices and revocations. To evaluate this scenario, we construct a synthetic highly correlated covariance matrix which has no uncorrelated markets (shown in Figure 5.6). For our experiment, we use these synthetic covariances and EC2 prices. Figure 5.5b shows that in highly correlated markets, the portfolio approach outperforms the greedy approach by 10% in both savings and risks, and is comparable to Spot Fleets. This is because the correlated markets reduce the possibility of exploiting market independence for portfolio diversification. We emphasize that this is a worst-case scenario. Today's markets provide ample opportunity to diversify across independent markets.

Google preemptible instances. Unlike spot instances, Google's preemptible instances have fixed discounts. The lack of variable pricing means that we cannot use it to estimate the correlations between different server types. Given the lack of any availability data, we assume that ExoSphere would require active probing [159] to infer availability. For our experiments, we use the covariance matrix from the EC2 markets. The resulting cost savings and revocation risks are shown in Figure 5.5c. Because the preemptible instances have discounts in a very narrow range (80-80.2%), the portfolios also have nearly constant cost savings. Importantly, increasing diversification reduces the revocation risks by 75%.

Result: *Portfolio construction outperforms greedy selection in uncorrelated markets, and its diversified portfolios reduce revocation risk even when markets are correlated.*

5.5.2 Application Performance

We now examine application performance when running on portfolios with different risks and transiency-specific policies. For ease of exposition, we group portfolio risk-averseness into three types: low, medium, and high. Low risk-averseness typically

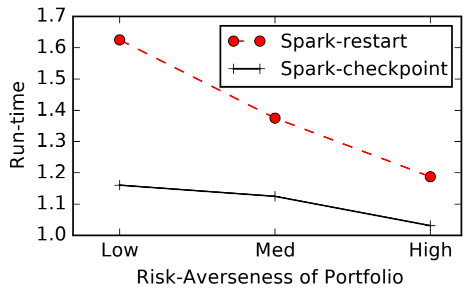


Figure 5.7: Spark job run-time (normalized to on-demand servers) when one market fails.

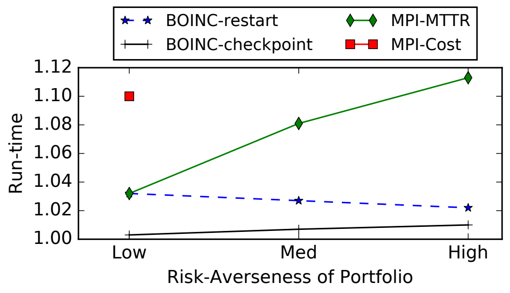


Figure 5.8: BOINC and MPI and risk averseness.

corresponds to single-market mode of operation, while high risk-averseness corresponds to using as many markets as possible. Note that in practice, the portfolio weights for many markets are quite small (< 0.001), thus these markets will not be used for applications requesting < 1000 servers. As a result, the high risk-averseness scenario corresponds to about 10-15 server-types for most applications.

Spark. To see the impact of using application-specific policies, we run the Spark KMeans workload with the "medium" risk-averseness portfolio. We vary the fault-tolerance and replenishment policy, and show the increase in running time compared to on-demand servers when there is a market failure (corresponding to about 1/5th of the servers lost) in Table 5.1. The periodic checkpointing with eager replenishment increases job length by 7% compared to on-demand servers, while the no-checkpointing, no-replenishment policy increases job length by 54%. Overall, these policies yield 60-80% cost savings compared to on-demand servers, while increasing completion times by 1.07-1.5×.

Next, we observe Spark performance as a function of portfolio diversity and fault-tolerance policy, and use eager replenishment. Figure 5.7 shows the impact of a single revocation event on the running time (compared to on-demand servers). Since checkpointing reduces RDD recomputation in Spark, the increase in job completion times is only 5% in highly diversified portfolios and 16% in the low risk averseness portfolio. Even without checkpointing, using portfolios yields 62%, 38%, and 18% increases in running time with low, medium, and high diversification, while still yielding 86% cost savings.

As the portfolio diversity increases, the impact of revocation of servers in one market decreases, because a smaller fraction of the state is lost. The recovery of the lost state is the primary contributor to the increase in running time. Thus, Spark workloads benefit from portfolio diversification, and can reduce the increase in running times to 5% using checkpointing, and 86% cost savings compared to on-demand servers.

MPI. For MPI, we examine the *expected* running time of jobs compared to on-demand servers in Figure 5.8. We consider two cases. In the first case, we assume that the job-lengths are known in advance, and only request markets with MTTRs greater than two times the job-length. When only a single-market portfolio is chosen ("MPI-MTTR" in the figure), the running time increases by only 3% compared to on-demand servers. In the second case, we use the Spot Fleet strategy of picking the cheapest single market. In this case ("MPI-Cost" in the figure), the running time increases by 10%. Thus, picking stable markets with MTTR >> job-length significantly reduces the running time for MPI, when compared to the lowest-cost strategy. The cost-savings are ~80% in all cases.

Note that since a single server revocation stops the entire MPI job, MPI only cares about minimizing server revocations, and *not* simultaneous revocations. Portfolio diversification reduces the number of simultaneous revocations and not the total number of revocations. The effect of increasing portfolio diversity can also be seen with the "MPI-MTTR" strategy in Figure 5.8, and we see that for highly diversified portfolios, the increase in running-time is close to 10%. Thus, MPI does not benefit from portfolio-diversification, and stable, single-portfolios represent the best portfolio.

BOINC. Unlike MPI, revocation in BOINC only affect the tasks that are running on the servers. We show BOINC performance compared to on-demand servers with the lazy-checkpointing and the restart fault-tolerance policies in Figure 5.8. With the highly diversified portfolio, the increase in running time compared to on-demand servers is less than 1% with the lazy-checkpointing and about 2% without checkpointing. Thus, ExoSphere can run embarrassingly parallel tasks at 1.01-1.05× overhead, but at more than 80% cost savings.

Result: *The impact of portfolio diversification is highly application dependent. Application performance is governed by the combination of portfolio composition, application characteristics, fault-tolerance policy, and recovery policy. Cost depends on application performance and a combination of portfolio composition and application policies. Portfolios with low risk averseness (i.e. high risk portfolios) yield lower costs, albeit at higher revocation risk.*

5.5.3 Multiple Applications

ExoSphere is designed to run multiple applications simultaneously, and we now evaluate its multiplexing capabilities. We use the Eucalyptus cloud traces [23] for realistic application arrivals and resource requests. We evaluate the costs of running such a trace by using a simulator which replays the spot price traces. The purpose of our simulator is to observe cluster utilization and costs for different application usage scenarios and cloud prices.

Sharing of servers among multiple applications can reduce total costs. We evaluate the extent of these savings compared to not sharing servers (e.g., a private mode). We assign the risk-averseness to the jobs in the Eucalyptus trace using four different distributions: all jobs requiring low risk-averseness portfolios; all high risk-averseness portfolios; equally distributed among low-medium-high; and distributed in a 1:2:1 ratio. Figure 5.9 shows the total cost incurred with the private and the cluster sharing

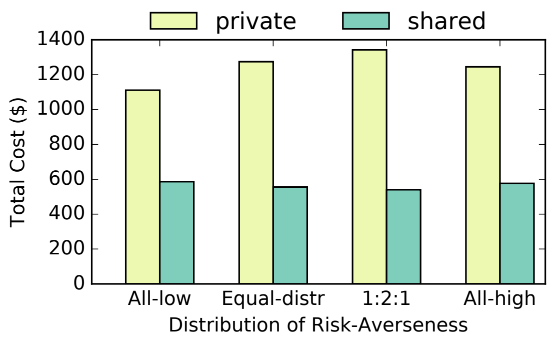


Figure 5.9: Packing policy comparison.

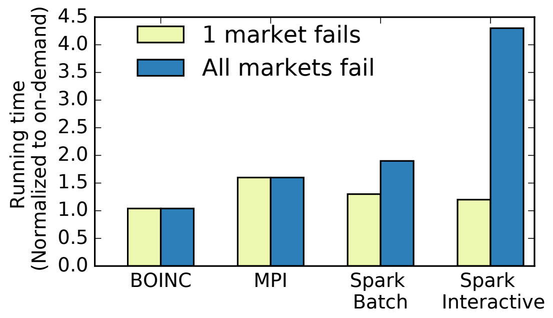


Figure 5.10: Performance during the worst-case "black swan" failure events.

policy, which uses first-fit bin-packing to find idle resources in servers in each market to satisfy portfolio requirements. By sharing servers among multiple applications, this packing policy lowers costs by 50%.

5.5.4 Black Swans: Multiple market failures

Finally, it is important to note that while the portfolio construction technique gives the best expected portfolios, it assumes that the historical price trends will continue to hold. In extreme situations, it is possible that even when selecting servers with low risk of concurrent revocations, all (or a large majority) of markets might be revoked. These events are akin to stock market crashes and are the black-swan events that have a high impact and are hard to predict. We show the performance implications of such extreme events in Figure 5.10, which shows the relative performance of applications running on their ideal portfolios and using the "best" fault-tolerance policy. We compare the application performance in the expected case of a single-market failure versus the worst case when all markets fail. We see that the impact on different applications is varied. BOINC and MPI see no difference in their expected and worst-case, since their preferred portfolios have only a single market. For a batch workload in Spark, the increase is significant (50%), and for the interactive Spark workload, the increase is more then $4x$. We note that these black-swan events only cause a one-time performance-hit, and don't affect expected cost savings.

Discussion: The real-world success of any portfolio-based technique relies on the ability to model the returns and risks of the underlying markets. However, there are many events that cannot be modeled using historical price traces alone. Black-swan and other rare events are hard to model, since they may have never occurred in the past. We also note that transient instances can be unilaterally revoked by the cloud provider, and cannot be modeled by price-traces alone. While price-based modeling has led to great gains in financial markets, spot markets are different from classic financial markets in a number of ways. Spot prices show higher volatility—prices can increase 10X in a single jump. The high volatility makes spot markets harder to model and also means that existing financial models that assume low volatility cannot be applied directly.

5.6 Related Work

Our work leverages prior work on transient servers, cluster management, and portfolio theory.

Systems for Transient Servers. Recent work has looked at developing systems and middleware for transient servers like EC2 spot instances. SpotCheck [182] introduced the notion of a *derivative cloud* which combines spot and on-demand instances to run arbitrary applications on top of spot instances with high availability. SpotCheck relies on nested virtualization and continuous memory checkpointing to live-migrate to on-demand instances upon revocation. For batch jobs, SpotOn [197] performs spot market selection by considering the market cost and availability, and showed that the fault tolerance mechanism has an important influence on the server selection. OptiSpot [91] uses a combination of queueing-based application performance models and a markov chain based spot price models to select the right server type and bid price for a given application.

Transient server selection. ExoSphere's portfolio based server selection differs from prior work in regards to its flexibility and generality. The risk tolerance knob introduced in ExoSphere allows easy and explicit characterization of portfolio risk. Transient server selection policies in earlier systems [182, 197, 176, 221, 142, 76] do not have explicit support for managing revocation risk. This is because these systems have mostly targeted a single class of applications, and have server selection policies suited to that. For example, Flint [176] runs Spark [230] applications on transient cloud servers, and selects markets with the lowest effective cost for batch Spark jobs, and uses a greedy multi-market strategy for batch-interactive jobs.

Transiency-aware Applications. Prior work on making applications transiency-aware has involved application-level application-specific approaches. For example, Flint [176] and TR-Spark [221] modify Spark to better support transiency, e.g., via checkpointing, while related work focuses on optimizing MPI jobs for spot servers [142]. Similar work has modified Hadoop and other batch applications for transient servers as well [226, 212]. Checkpointing and scheduling policies for data processing and machine learning workloads on transient resources have been developed more recently in [223, 107].

Portfolios. In contrast to prior work which focused on supporting narrow classes of applications on transient servers, ExoSphere's goal is to provide a common platform for a wide range of applications. ExoSphere distills common abstractions based on the experience of past work to enable easy modification of current and future applications to support transiency. Our portfolio abstraction is inspired from financial economics, where investment portfolios are created for diversification and to reduce risk [148, 94, 92, 143] . In transient server markets, diversification reduces the probability of simultaneous revocations, and thus plays a crucial role in server selection. The idea of risk-reduction using diversification formalized in Modern Portfolio Theory in the 1950's remain the basis for other popular portfolio creation techniques such as Black-Litterman [170]. Exploring other portfolio construction techniques for server selection remains part of our future work.

A significant amount of prior work has gone into optimizing multiple objectives in the context of server selection. Server selection to optimize for performance and cost of on-demand servers (but without transiency considerations) is discussed in [97, 211], and in [202] which uses genetic algorithms to find spot/on-demand pareto-efficient frontier. CherryPick [47] uses bayesian optimization to select cost-optimal on-demand servers. Utility-based selection of servers is shown in [73], which selects homogeneous cloud servers for different applications. In contrast, ExoSphere selects a heterogeneous mix. Our use of portfolios is not to be confused with portfolios of policies/algorithms—wherein a portfolio of multiple policies and algorithms are run to find the most efficient algorithm. This approach is commonly used in SAT solvers [116], and has also been applied to cloud scheduling [193].

Cluster Management. There has been a significant amount of work on designing cluster resource managers [113, 200, 201, 172, 67] and resource management policies for running multiple applications in data center [123, 84, 99, 140] and cloud environments [86, 85]. In particular, ExoSphere builds on Mesos [113] and can be viewed as a transiency-aware cluster manager. To our knowledge, current cluster managers do not support transiency and variable pricing as first-class primitives.

Resource Allocation in Data Centers. There has also been a significant amount of work in allocation of surplus resources and risk-driven resource allocation in data centers. [71] allows idle resources to be reclaimed and uses resource usage traces to predict resource availability for long running services. Services can run uninterrupted with a high probability by maintaining slack between the resource allocation and usage. Risk-aware overbooking of resources by using admission-control policies is discussed in [199]. [173] considers job task placement to mitigate correlated failures in the data center, where a failure in a power component can affect multiple machines and hence multiple tasks. Using surplus resources in computational clusters has a long history—Spawn [206] introduced a market based system for selling idle resources to applications, similar to what public cloud operators are doing now. ExoSphere's portfolio-driven allocation policies can work in data center environments where the resource allocation involves optimizing two different and possibly competing objectives. For example, ExoSphere can be used to minimize performance interference adjusted costs, where instead of revocation risk, there is risk of performance interference due to co-located applications. We note that the transiency specific API that we have developed for applications can be used "as-is" in data center environments, where the resources of low priority jobs are revoked in favour of higher priority applications.

5.7 ExoSphere Summary

The effective use of transient servers is predicated on their careful selection. In this paper, we introduced portfolio modeling for transient server resource management. Unlike prior resource management schemes, portfolios allow the easy creation of virtual clusters with different revocation risk tolerances. Existing convex optimization techniques can compute portfolios efficiently—computing portfolios for 2,500 spot markets takes well under one minute.

We have prototyped and implemented a portfolio-driven cluster manager, ExoSphere, that exposes a narrow, uniform interface and allows multiple applications to develop and use their own transiency-aware policies for handling revocations. We have shown that existing applications such as MPI, Spark, etc., can use this interface to design their own policies and significantly increase their performance and cost saving on transient servers. Our experience with portfolios has shown that they are a powerful and promising resource management primitive, and can be especially useful in situations where multiple resource management objectives (such as cost and revocation risk) have to be minimized.

CHAPTER 6

RESOURCE DEFLATION: A NEW TECHNIQUE FOR TRANSIENT RESOURCE RECLAMATION

Data centers and clouds are increasingly offering low-cost computational resources in the form of transient virtual machines. Whenever demand for computational resources exceeds their availability, transient resources can reclaimed by *preempting* the transient VMs. Conventionally, these transient VMs are used by low-priority applications that can tolerate the disruption caused by preemptions, which are akin to fail-stop failures.

This chapter asks the question: "Can resource providers still offer low-priority transient resources by using different resource reclamation mechanisms?" We propose an alternative technique for reclaiming resources, called *resource deflation.* Resource deflation allows VMs to dynamically shrink (and re-expand) based on resource pressure, instead of being preempted outright. Deflatable VMs allow applications to continue running even under resource pressure, and increase the applicability and performance of low-priority transient resources. We develop mechanisms, policies, cluster-management techniques, and specialized applications for deflation, that allow VM resources to be dynamically reduced while minimizing performance degradation. When deflatable VMs are deployed on a cluster, our policies allow up to 2.3$\times$ overcommitment without the risk of preemption.

6.1 Motivation

Transient computing resources are becoming commonplace in data centers and in cloud computing platforms. A transient computing resource, such as a server, is one that can be unilaterally revoked by the provider for use elsewhere. In enterprise data centers, servers running low-priority batch applications can be reclaimed by terminating their virtual machines upon resource pressure from high priority applications [201]. In cloud context, all three major cloud providers, Amazon [17], Azure [40], and Google [24], offer *preemptible instances* that can be unilaterally revoked during periods of high server demand.

The primary benefit of transient computing is that it enables data center operators and cloud providers to significantly increase utilization of servers. Idling servers can be allocated to lower priority disruption tolerant jobs or sold at a discount to price sensitive customers. In both cases, the resource provider has the ability to reclaim these resources when there is increased demand from higher priority or higher paying applications. Preemptible cloud servers have become popular in recent years due to their discounted prices, which can be 7-10x cheaper than standard, conventional non-revocable servers. A common use case is to run data-intensive processing tasks on hundreds of inexpensive preemptible servers to achieve significant cost savings.

Despite the many benefits, the preemptible nature of transient computing resources remains a key hurdle. From an application standpoint, server revocations are essentially fail-stop failures, leading to disruption. Systems for using transient resources have received significant attention. Prior work has explored mitigating the impact of preemptions by developing transiency-specific fault-tolerance mechanisms and policies that work either at a system-level [182, 197] or are tailored to different applications [176, 180, 108, 224, 142, 209, 145]. In enterprise data centers, using transient resources to increase utilization and minimize the performance impact of preemptions remains an important problem [221, 238, 153, 201]. Even with these proposed solutions, the preemptible nature of transient resources presents a significant burden for many applications as they require changes to the application (legacy) code in many cases.

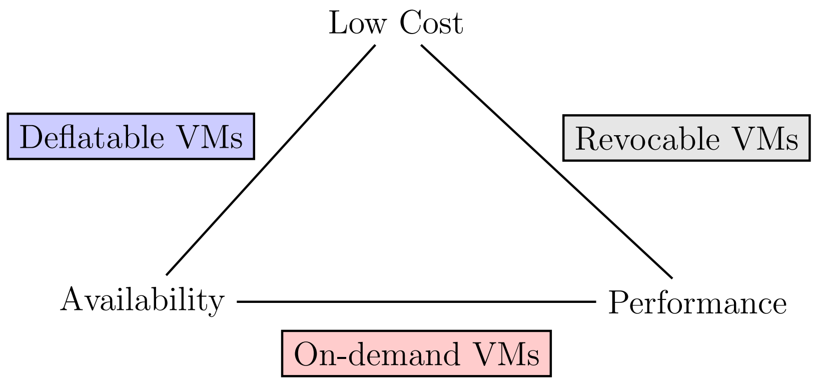


Figure 6.1: The three-way tradeoff between cost, availability, and performance. Deflation provides low-cost computation, but without the hassles of sudden preemptions present in current cloud transient servers.

In this chapter, we present *resource deflation* as a new abstraction for implementing and managing transient computing resources in data centers and cloud platforms. We argue that resource preemption is only one approach, and an extreme one, for reclaiming erstwhile surplus resources from low-priority applications. In resource deflation, transient computing resources allocated to a VM can be dynamically reduced to reclaim them. In this case, resource preemption becomes a *special case* of deflation where resources are deflated to zero. In general, the amount of deflation is determined by the magnitude of resource pressure from higher priority applications.

The primary benefit of resource deflation comes from a key observation—for many applications, the relationship between amount of resources allocated and application performance is sublinear or at most linear. This is shown in Figure 6.2 where the performance of different applications is shown for different resource deflation levels. We see that there is a significant operating region in which many applications can be deflated without paying the proportional performance penalty—when resources are reduced by 50%, the performance drops by $< 30\%$ for memcached, kernel-compile, and SPECjbb.

Resource deflation is more attractive than preemption for low-priority applications since they can continue to run, albeit more slowly, under resource pressure, rather than

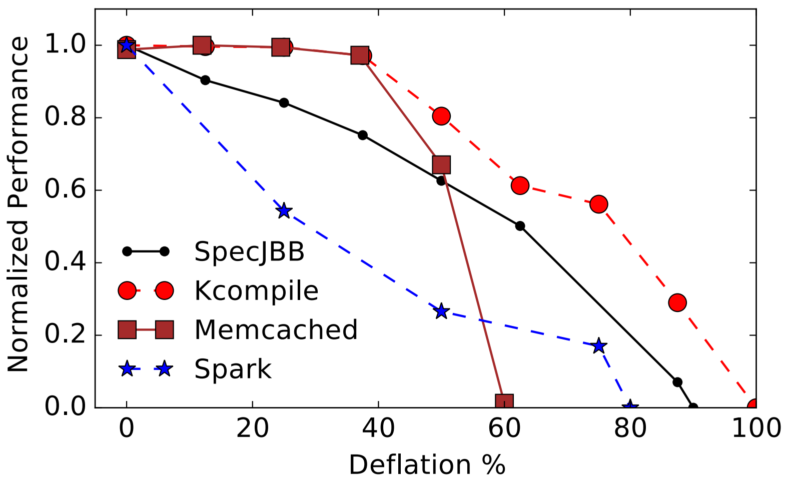


Figure 6.2: Application performance when all resources (CPU, memory, I/O) are reduced in the same proportion. In many cases, applications exhibit linear and sub-linear performance degradation due to deflation.

being terminated. In such a system, preemptions become rare events, and may even be eliminated under certain scenarios—reducing or eliminating the need for additional fault-tolerance mechanisms in applications. Deflation thus trades off performance for availability of transient resources. By reducing application resource allocation, deflation exposes them to potential performance degradation (Figure 6.1). However we show that this compares favorably to the performance degradation caused by fault-tolerance techniques required to deal with preemptions.

We design and develop a system that uses deflation for low-priority applications; and show that resource deflation is a suitable technique for increasing the utility of low-priority resources for a wide range of applications. In doing so, we make the following contributions:

1. We develop mechanisms for resource deflation by combining existing hypervisor overcommitment mechanisms. We show how our novel overcommitment approach outperforms existing mechanisms, reclaiming more resources while minimizing performance degradation.

2. We develop a black-box application-agnostic technique based on CPU performance counters to infer the minimum resource allocation required by an application to function with acceptable performance and not crash due to deflation.

3. We design and implement cluster management policies that can completely remove the risk of preemption, at cluster overcommitment levels as high as 2.3×.

4. While deflation is designed to be application agnostic, we also develop deflation-aware variants of four popular applications and show that the performance impact of deflation can be minimized by up to 6×.

6.2 Deflation Background

In this section we provide background on transient servers and motivate the need for VM deflation as an alternative.

6.2.1 Transient Computing

Our work assumes a virtualized data center where applications run in either VMs or containers multiplexed on physical machines. Such a virtualized architecture is now commonplace in both enterprise and cloud data centers. Since data center capacity is provisioned for peak demand, the average utilization tends to be low [201], in the range of 20-30%. Data center operators can increase the overall system utilization or maximize revenues in case of the cloud, by offering unused server capacity transiently to low-priority applications or at a discounted cost.

Thus the data center is assumed to host two classes of applications—high and low priority workloads. Low priority applications are scheduled whenever there is enough surplus server capacity in the data center; however, resources allocated to VMs of low priority applications are assumed to be transient. Some or all of these resources may be reclaimed at short notice when server demand from high priority applications starts increasing.

Current systems implement resource reclamation in the form of server revocations, a form of preemption from low-priority applications. Cloud offerings such as Amazon Spot instances [17], Google Preemptible VMs [24], and Azure batch VMs [40] are examples of such low-cost but preemptible VMs. Enterprise data centers similarly preempt low-priority jobs when high priority jobs arrive [221, 238, 201].

In this work we assume that low priority applications run inside a special type of VM called deflatable VMs. Deflatable VMs support fractional resource reclamation by allowing the cluster manager to dynamically reduce ("deflate") the CPU, memory, and I/O resources allocated to the application. Deflation can be done progressively in stages—whenever more resources are to be reclaimed, the VM's resources can be shrunk to meet the increased demand.

6.2.2 Elastic Scaling versus Resource Deflation

Modern cloud platforms and virtualized data centers support vertical elastic scaling to handle dynamic application workloads [138]. Vertical scaling allows the server capacity allocated to a VM to be dynamically changed to match the application workload dynamics. In other words, elastic scaling changes resource allocation based on the workload while resource deflation forces an application to scale *down* its resource usage—in response to resource pressure from elsewhere. While elastic scaling mechanisms are well studied in literature [121, 100, 187], such mechanisms cannot simply be applied "in reverse" to implement deflation.

There are several important differences between elastic scaling and deflation. First, elastic scaling approaches, in general, endeavor to always give an application *adequate* resources based on its current needs. Thus if an application's demands rise, it is given more resources, and if the demands fall, the surplus resources are reclaimed while still ensuring it has adequate resources for the reduced demand. In contrast, resource deflation can (and often will) allocate resources that are *inadequate* for its needs. Of course, if an application is underutilizing resources, then they can be easily reclaimed during resource pressure. However, if resource pressure persists or increases, allocated resources can be reduced further causing the allocation to be substantially below application needs. In normal circumstances, elastic scaling does not reduce allocation below the demand.[1]

Second, elastic scaling techniques assume that the user or application specifies explicit performance goals in the form of service level objective (SLO) and scales resources up or down based on the application's specified performance objectives. In contrast, transient computing emphasizes use of idle resources by allocating them to low-priority applications. Performance or SLOs of low priority jobs is not the primary consideration when reclaiming resources for high-priority jobs. Despite the absence of user-specified SLOs in transient computing, deflation algorithms need to carefully consider, and minimize, the impact of resource reclamation on the resulting performance degradation.

6.2.3 VM Overcommitment versus Resource Deflation

Our VM deflation approach makes use of virtual machine resource overcommitment mechanisms to dynamically adjust the resource allocation. These mechanisms are also used in virtualized cluster managers (such as VMWare DRS [103] and OpenStack) to increase server consolidation by packing more VMs on to a smaller set of physical servers. At surface-level, this is similar to the use of our proposed VM deflation system, which also seeks to increase cluster-wide utilization through overcommitment.

[1]In extreme overloads, even elastic scaling may unable to meet the application demands, due to lack of cluster resources, and may degrade performance. We treat extreme overloads as different from deflation.

However, compared to conventional VM overcommitment and cluster managers, we argue that there are key differences in both our approach and higher-level objectives:

Magnitude of Overcommitment. Conventional virtualized cluster managers use VM overcommitment to a much smaller degree. Since the physical clusters are sized for relatively static and predictable enterprise workloads, the level of overcommitment, if any, is under 20%. In contrast, we propose overcommitting resources by around 50–80%.

Hybrid overcommitment. To achieve high overcommitment, we develop the use of hotplug based mechanisms, and show how they can be combined with hypervisor-based mechanisms to reclaim a large amount of resources quickly. The effectiveness of hotplug based mechanisms has only received scant attention.

Performance-counter guided resource allocation. Conventional virtualized cluster managers such as DRS rely on proportional overcommitment and disregard application tolerance to overcommitment. In contrast, we use CPU performance counters to infer the maximum deflation magnitudes for an application, and use that to guide both deflation *and* placement of new VMs.

Finally, the existence and prevalence of VM overcommitment mechanisms only increases the feasibility and viability of our proposed deflation approach.

6.2.4 Transient Server Applications

Traditional transient computing has assumed reclamation via server revocation. Consequently, interactive applications such as web services or transaction processing are assumed to be unsuitable for transient servers—since they cannot tolerate down times caused by server revocation.

Batch-oriented applications, on the other hand, are well suited for transient computing. Such applications tend to be both delay and disruption tolerant and can handle longer completion times. In the event of a preemption, they can simply be restarted from the beginning or restarted from a checkpoint if the application is amenable to periodic checkpointing.

Under deflation, all classes of applications become more amenable for transient computing. Performance degradation, rather than outright termination (and downtime), maybe acceptable even to many interactive applications except the most mission-critical ones. Temporarily deflating a batch application may be a better alternative that avoids wasteful restarts. Certain deflation policies that guarantee no preemption are useful for applications that have no checkpointing support or incur substantial checkpointing overheads.

6.3 Deflatable Virtual Machines

In this section we describe how resource deflation can be instantiated for virtual machines through deflatable VMs.

6.3.1 VM Deflation Overview

Resource deflation for a VM requires the ability to dynamically shrink (and grow) the resources allocated to the VM. The virtual machine monitor (also called the hypervisor) typically exposes an interface to determine the resource allocation of a VM and to allow dynamic modification to the allocation. A cluster manager or cloud management framework uses such APIs for initial placement of VMs and subsequent changes to the VM's allocation (Figure 6.3).

In our case, we can adapt these mechanisms for VM deflation. We design two types of VM deflation mechanisms—transparent deflation mechanisms, which transparently shrink the VM's resource allocation, and explicit deflation mechanisms, where the deflation is performed in a manner that is visible to the guest OS, (and by extension, to the applications and the application cluster manager). In the former case, the guest

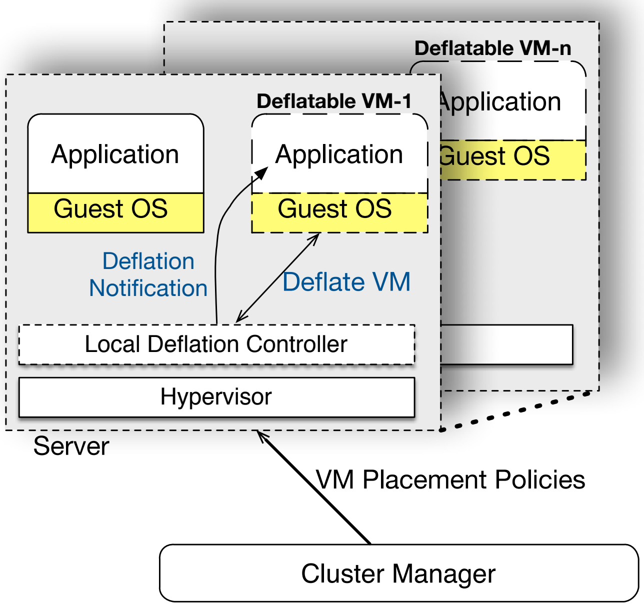


Figure 6.3: Overview of our deflation system

OS and applications are unaware of the deflation and the VM simply runs "slower" than prior to deflation. In the latter case, since deflation is visible to the guest OS and/or applications, they can take explicit measures, if wanted, to deal with deflation. We describe each mechanism and a hybrid approach that exploits the key benefits of both approaches.

6.3.2 Transparent VM Deflation

Since hypervisors virtualize resources and offer them to virtual machines, they can also *overcommit* these resources by multiplexing virtual resources onto physical ones. Transparent deflation exploits such multiplexing mechanisms to deflate resources by overcommitting them. For example, virtual CPUs (vCPUs) of the VM may be mapped to a dedicated physical CPU cores. Such an allocation can be deflated by remapping the vCPUs onto a smaller number of physical cores, and sharing the capacity of these cores using the hypervisor's built-in scheduling mechanism. Thus the guest OS and applications inside the VM still see the same number of vCPUs, but these vCPUs run slower.

In case of memory, hypervisors allocate an amount of physical memory to a VM and multiplexes the VM's virtualized memory address-space onto physical memory, via two-dimensional paging. Memory deflation thus involves dynamically reducing the physical memory allocated to a VM.

In the case of network, one or more logical network interfaces of a VM are mapped onto one or more physical NICs and a certain bandwidth of the physical NICs is allocated to each vNIC by the hypervisor. Network deflation involves reducing the physical NIC bandwidth allocated to the VM. Finally in the case of local disks, the I/O bandwidth allocated to each VM can be throttled.

In all of the above scenarios, the VM itself has no knowledge of the deflation, which is done at the hypervisor level "outside" the VM. The VM may get scheduled at a lower frequency or have less physical memory, etc. Our deflation framework has been implemented in KVM and Linux using Linux's cgroups facility. By running KVM VMs inside of cgroups, we can control the physical resources available for the VM to use. For deflating CPUs, we use CPU bandwidth control by setting the CPU shares of the deflatable VM. The memory footprint of a deflatable VM is controlled by restricting the VM's physical memory allocation by setting the memory limit in the memory cgroup. Similarly for disk and network I/O, we use the respective I/O cgroups to set bandwidth limits.

6.3.3 Explicit Deflation via Hotplug

Explicit deflation mechanisms use the notion of resource hotplug to change the VM's allocation in a manner that is visible to the guest OS and the applications. Modern operating systems and hypervisors now support the ability to hot plug (and unplug) resources. By unplugging virtual resources, the VM's resource allocation can be controlled. In the case of CPU, if a VM has n vCPUs allocated to it, its CPU resources are reclaimed by unplugging k out of n vCPUs. Hot plugging and unplugging requires guest OS support, since it must reschedule/rebalance processes and threads to a smaller or larger number of cores. Thus the deflation is visible to the guest OS and applications. In case of memory, we use memory unplugging to explicitly reduce the memory seen by the guest OS. We don't use hot unplug for NICs and disks since this is generally unsafe.

Hot unplugging has a safety threshold—unplugging too many resources (e.g., too much memory) beyond this safety threshold can cause OS or application failures. Furthermore, hot unplug can only be done in coarse-grained units. For example, it is not possible to unplug 1.5 vCPUs.

6.3.4 Hybrid Deflation Mechanisms

Both transparent and explicit deflation have advantages and disadvantages. Explicit deflation—by virtue of being visible, allows the OS and applications to gracefully handle resource deflation. However, deflation can only be done in coarse-grained units and has a safety threshold. Transparent deflation can be done in more fine-grained steps and has a much broader deflation range than explicit deflation. It does not require any guest OS support but can impose higher performance penalty since the OS and applications do not know that they are deflated.

Our hybrid deflation technique combines both mechanisms to exploit the advantages of each. Initially, a VM is deflated using explicit deflation until its safety threshold is reached for each resource. From this point, transparent deflation is used for further resource reclamation to extract the maximum possible resources from the VM under high resource pressure. Figure 6.4 presents the high-level pseudo-code of our hybrid deflation approach. The key challenge is to determine the hot unplug safety threshold so as to switch over from explicit to transparent deflation.

```
1 def deflate_hybrid(target):
2   hotplug_val = max(get_hp_threshold(), round_up(target))
3   deflate_hotplug(hotplug_val)
4   deflate_multiplexing(target)
```

Figure 6.4: Pseudo-code for hybrid resource deflation.

For deflating CPUs, our hybrid approach first sets the hotplug target by rounding up the target number of vCPUs (line 2 in Figure 6.4). Then the cgroups based CPU multiplexing deflation can deflate the VM the rest of the way. Note that the hotplug operation may not always succeed in removing all the CPUs requested—the guest OS may unplug the CPU only if it is safe to do so. If the number of reclaimed CPUs via hotplug is less than the number requested, then the multiplexing-based CPU deflation takes up the slack.

When deflating memory, we set the hotplug threshold by using the guest OS's resident set size (RSS)—since unplugging memory beyond the RSS results in guest swapping, and we presume that it is safe to unplug as long as the VM has memory greater than the current RSS value.

6.4 KneeFinder: Finding The Limits of Deflation

Deflating a VM to reclaim resources allocated to it causes a performance degradation for the resident application. The performance degradation faced by the VM depends on the magnitude of the deflation, the workload characteristics, and the application's resource usage model.

The performance of VMs under deflation is governed by the application's utility curves, as shown in Figure 6.5, which shows a representative curve of application performance vs. the magnitude of deflation. Since a VM may have a certain amount of surplus for each resource (due to overprovisioning), removing the unused surplus resources from the VM results in little performance loss. This is indicated by the

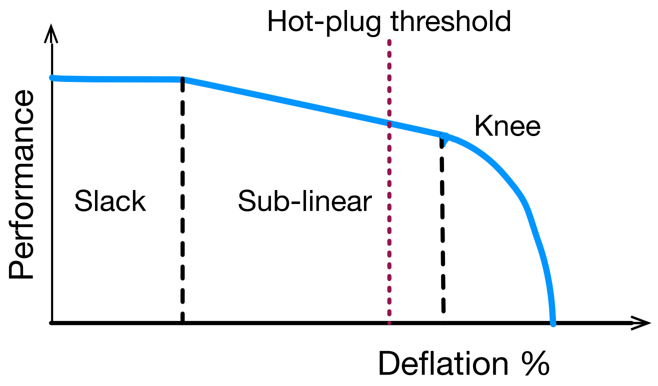


Figure 6.5: A representative deflation utility curve.

"slack" region in Figure 6.5. Deflation beyond this point causes a graceful performance degradation that manifests itself in a sub-linear or linear drop in performance. Sublinear decrease in performance can also result from the inherent resource-adaptivity of the underlying operating system or of the application itself. For example, most parallel applications only achieve sub-linear speedup due to Amadahl's law, and thus reducing the number of threads (vCPUs) that an application uses results only in sublinear performance degradation. This behavior continues until we reach a "knee" after which further deflation causes a precipitous drop in performance. Note that the hotplug safety threshold lies in the (sub) linear region of the curve. In general, deflating beyond the knee provides so little utility to the application that it is better to preempt the application rather than deflate it to this level. Utility curves of four different applications shown in Figure 6.2 also show these sub-linear and knee regions.

6.4.1 Black-box Knee Determination

Utility curves as shown in Figures 6.2 and 6.5 are useful for determining the performance after deflation. However, in general, the deflation utility curves are unknown to us and neither SLOs nor such curves are specified by the user. Finding application utilities requires access to application-level metrics such as throughput or response times. In environments such as public clouds, the cloud provider is distinct from the application provider and can't access these application metrics.

To overcome this limitation, we present a black-box approach for dynamically finding the knee of the (unknown) utility curve of applications at run-time. Knowledge of the knee can be used to design policies to intelligently deflate each application and increase the usefulness of transient computing.

Our approach is motivated by two observations. First, knowledge of the *entire* curve, while useful, is not strictly necessary and it is adequate to simply determine the knee, which is just one point on the curve. Thus determining the knee is less expensive than determining the full curve [241], making it feasible to do so in live production environments.

Our second observation is that it is possible to infer the performance knee by using the correlation between hardware performance counters and application performance. CPU performance counters are present in all modern CPU families and are accessible to the hypervisor.

In particular, we use instructions-retired as a proxy for application performance. Prior work has shown that the rate of instructions (instructions/second) is fairly closely correlated with application throughput [236]. We verify and build on this finding—Figure 6.6 shows the correlation between Memcached performance and the CPU performance counters, across various deflation levels. At 50% deflation, we observe a sharp drop in the instructions/second, corresponding to the application's performance knee. Table 6.1 shows the correlation between the performance counters and the application performance, and we can see that in general, the performance counters have a high correlation with application performance.

We assume that rate of instructions is a reliable indicator of throughput, and when resources are deflated (even memory and I/O), the decrease in performance eventually manifests itself in the form of degradation of throughput of CPU instructions. Using

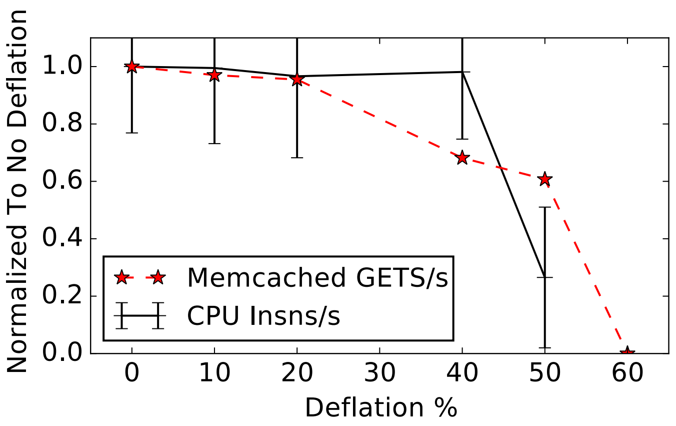


Figure 6.6: Memcached performance shows high correlation with CPU counters (pearson correlation=0.72). Drop in counters predicts the knee at 50% deflation.

Application	Correlation
Memcached	0.72
Kcompile	0.99
SpecJBB	0.99

Table 6.1: Correlation between application performance at different deflation levels and CPU performance counters.

multiple performance counters (such as cache-access/misses, cycles stalled due to memory loads, etc.) is a promising approach [157] and is a subject of future work.

Our black-box knee finder uses short deflation "probes" to search for the performance knee, and stops the search based on the values of the performance counters. We find the performance knee for each resource type (CPU, memory, etc.) by first establishing a baseline of performance indicators. This baseline is established by looking at the entire undeflated history of a VM. We then deflate the VM briefly (30 seconds) using the hybrid deflation mechanisms, and measure the effect on the performance counters (PMCs).

If we observe a precipitous drop in the instructions, then we have found the knee and stop the search. We find the knee through a simple exponential search (Figure 6.7). If M is a VMs maximum resource allocation, then we search for the knee (m) by progressively increasing the deflation by a factor of $\frac{1}{\sqrt{2}}$. Upon the completion of the search, we revert the VM to its original resource allocation. We repeat this process for each resource type (CPU, memory, disk, and network), and establish the knee for each resource type. Our approach finds the knee one resource at a time to avoid combinatorial search over multiple resources.

```
base_mean, base_std = get_baseline_pmc()

def knee_search():
  lower_limit = 0.1; x = 1.0; s = 1/sqrt(2)
  while x > lower_limit:
    x=x*s
    deflation_target = x*Max_alloc
    deflate(deflation_target)
    sleep(30)
    probe_perf = get_pmc()
    if (probe_perf-base_mean) > 3*std or \
        probe_perf < base_mean/3:
       return deflation_target
 return lower_limit
```

Figure 6.7: Pseudo-code for knee-finding

A change in the VM's workload phase can result in a different performance knee. To detect changes in the workload phase or the execution environment of the VM, we again use CPU performance counters. In particular, we rely on other metrics like instructions per unhalted clock cycle (IPC)[2].

We continuously record the IPC over the lifetime of a VM, and if we detect a significant change in the IPC in a moving window (3 standard deviations above the mean), then we signal a change in the workload phase, and we trigger the knee-finder to run again.

[2]For knee-finding, we use instructions/second, which is different from IPC.

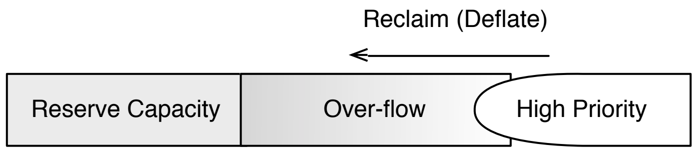


Figure 6.8: Resource Pools

6.5 Cluster-wide Deflation Policies

A data center or a cloud platform employs a cluster manager (or a cloud management framework) which is responsible for mapping low and high priority application VMs onto specific servers and for policies to reclaim resources from low-priority VMs under resource pressure. In our case, such a cluster manager employs deflation policies to determine which low-priority VMs to deflate, by how much, and when deflated VMs can be reinflated when the resource pressure subsides.

6.5.1 Cluster Resource Pools

Our cluster-wide deflation policies uses two key parameters to determine how aggressively to use unused server resources and how aggressively to deflate low-priority VMs when reclaiming these resources. The first parameter, reserve-capacity, specifies the total cluster capacity reserved for transient deflatable VMs. This reserved capacity is the minimum capacity guaranteed regardless of resource pressure.

The second parameter, overflow-factor, specified as a multiple of the reserve capacity, specifies what portion of the non-guaranteed cluster capacity to use for low-priority VMs. This parameter controls how aggressively to use unused server resources for low-priority VMs. Thus we effectively partition the cluster into multiple abstract virtual resource pools (Figure 6.8). A server in the cluster may be dedicated entirely to a single pool, or may have its resources dedicated to multiple pools.

Different combinations of reserve-capacity and overflow factor yield a range of policies. A reserve-capacity of zero implies that no resources are guaranteed for low-priority VMs and the entire cluster can be used by high priority VMs (after first deflating and preempting low-priority VMs). If overflow factor is set to a sufficiently high multiple, it implies that the entire cluster can be used by low-priority VMs if there are adequate resources. Ofcourse, this overflow capacity can be reclaimed up to reserve-capacity, at any time under resource pressure.

These resource limits and pools allow cluster and cloud operators more flexibility in controlling resource allocation of different principals. The reserve capacity can be set to the cumulative "knee" of the deflatable VMs. Since deflatable VMs are only preempted if deflated past the knee, this sets a bound on the max deflation, and guarantees no preemptions.

6.5.2 VM Deflation Policies

With VM deflation, servers can accommodate incoming VMs by potentially deflating existing deflatable VMs. In this subsection, we look at the policies which determine *How much to deflate each VM by?*

Our guiding principle is to deflate *multiple* VMs in response to resource pressure. We have seen that applications often have "slack" in their resource utility curves. Thus by deflating multiple VMs by small amounts, it is possible to minimize the performance degradation due to deflation.

The extent of VM deflation depends on two factors. First, the resource pressure on the server. This resource pressure is a consequence of a new incoming VM that is assigned to this server. The second factor that determines the extent of deflation, are the other colocated deflatable VMs. Thus the magnitude of VM deflation is entirely local to the server.

Now assume that r resources of a particular type (CPU, memory, I/O) must be reclaimed. Let there be n deflatable VMs, and let their current resource allocations be c_i, and their maximum allocations be M_i. Our task is to reclaim x_i from each VM such that $\sum x_i = r$. The *distribution* of these x_i's is controlled by the local deflation policies, which we describe below.

Proportional deflation. The idea is to deflate VMs such that all VMs lose the same fraction of resources. That is, $\frac{M_i - x_i}{M_i} = \frac{M_j - x_j}{M_j} = \alpha$ for all i, j. Thus for the proportional deflation policy, each VM is deflated by :

$$x_i = \alpha M_i = \frac{r}{\sum M_i} \cdot M_i \tag{6.1}$$

This is repeated for all resources (CPU, memory, I/O), and the VMs are deflated to their new targets $c_i - x_i$, and the VM is immediately deflated using the hybrid deflation mechanisms described earlier in Section 6.3.

Knee-aware Proportional. The above proportional policy can result in VMs deflated beyond their performance knees, and can result in the VMs entering the preemption zone. To mitigate this, the knee-aware proportional policy attempts to places a minimum limit on deflation. Using the knee-finder, we can obtain deflation lower limits m_i for each VM. We then try to reclaim resources x_i such that:

$$x_i = M_i - m_i + \alpha \cdot \left(\frac{M_i}{M_i - m_i} \right) \tag{6.2}$$

If the resource knee (m_i) is not known (as may be the case for newly launched VMs), the VM is run with full resources (0% deflation) until the knee-finder is run. Once the m_i value is known, this knee-aware proportional policy algorithm is re-run for all the $n+1$ deflatable VMs so that even the new deflatable VM is deflated.

Utiliity Maximization. While the proportional deflation policies described above can reclaim resources to meet a target, they are not explicitly maximizing VM performance. To maximize performance, we resort to utility-based maximization, that partitions a server's resources such that the aggreagate server performance is maximized. In other words, $\max \sum U(M_i - x_i)$.

The utility-maximization approach assumes the utility curves are available, which is rarely the case, and hence not a practical policy. We thus use the knee-aware proportional policy by default, and compare it with the utility-maximization approach in Section 6.7.

Preemption. While deflation aims at minimizing the number of VM preemptions, high resource pressure, or insufficient reserve capacity, can lead to VM preemptions. With the knee-aware proportional policy, we are forced to preempt if the sum of deflatability of all VMs is greater than the reclamation target: $r > \sum(c_i - m_i)$. Our preemption policy is to preempt VMs that are closest to their knee. That is, we sort VMs in descending order of $\frac{c_i - m_i}{M_i}$, and keep preempting VMs until we reach the reclamation target and the newly assigned VM can begin running.

Reinflation. We also use deflation as a purely reactive mechanism. That is, VMs are deflated only when a server is under resource pressure, and are not deflated if there are enough resources to run in non-deflated mode. The resource pressure on a server is continuously monitored, and the VM deflation levels are rebalanced upon each change. We use the same proportional policies for reinflation that we use for deflation. That is, instead of a resource deficit r, we find the resource allocations with a new resource surplus $-r$. Negative values of x_i in Equations 6.1, 6.2 indicate reinflation rather than deflation, and we reinflate each VM by $-x_i$.

6.5.3 Deflation-aware VM Placement

Assuming the above policies, the cluster manager handles each incoming VM by placing the VM onto a server.

6.5.3.1 Placement Without Limits

Without any pre-specified reserved-capacity or overflow-factor, a VM can be placed on any machine on the cluster—effectively a single large pool. Assigning VMs to servers is a mutli-dimensional bin-packing problem, and is well studied in the context of VM consolidation [152].

Conventionally, for the online VM placement, bin-packing policies such as first-fit, best-fit have been used [161]. These policies use resource availability (or “free space”) on each server to guide VM placement—first-fit picks the first server with sufficient free space, and best-fit picks the server with the most available free space. The resource availability on a server is easy to compute as

$$\mathbf{Available} = \mathbf{ServerCapacity} - \mathbf{Used} \tag{6.3}$$

Where the **Used** vector is computed by adding the resource allocations of all the VMs running on the server.

Deflatable VMs introduce an additional complexity to this bin-packing setup: since VMs can be deflated, the conventional resource availability formulation of Equation 6.3 does not hold. Instead, we also account for the deflatability of the VMs:

$$\mathbf{Available} = \mathbf{ServerCapacity} - \mathbf{Used} + \frac{\mathbf{Deflatable}}{\mathbf{Overcommitted}} \tag{6.4}$$

Here, the **Deflatable** vector is the maximum total amount of resources that can be reclaimed by deflation ($\sum c_i - m_i$). We scale the deflation vector by pairwise dividing it with the **Overcommitted** vector. The **Overcommitted** vector captures the extent of the total deflation already done, and is computed as follows:

$$\mathbf{Overcommitted} = \frac{\mathbf{Committed}}{\mathbf{Used}} = \frac{\sum M_i}{\sum c_i} \tag{6.5}$$

The idea behind scaling the deflatable vector is to prefer servers that are less overcommitted, so as to spread the load more evenly on the cluster.

We use this definition of availability (Equation 6.4) to compute the "fitness" of placing a new VM onto a server. As in [101], we use the cosine similarity between the demand vector and the availability vector to determine fitness: fitness$(\mathbf{D}, \mathbf{A_j}) = \frac{\mathbf{A_j} \cdot \mathbf{D}}{|\mathbf{A_j}||\mathbf{D}|}$. Here, $\mathbf{D}$ is the demand vector of the new VM, and $\mathbf{A_j}$ is the resource availability vector (Eqn 6.4) of server j.

The best-fit packing policy then simply selects the server that maximizes packing fitness. In addition to the first-fit and best-fit policy, we also implement a a "2-choices" policy that randomly picks two servers and assigns the VM on the server with the higher fitness function.

6.5.3.2 Placement With Usage Limits

When the reserve-capacity and overflow factors are specified, then we place VMs into the appropriate pool by using the deflation-aware VM placement policy (such as best-fit). Incorporating the reserve capacity and overflow factors into the VM placement results in hierarchical allocation as shown in Figure 6.9.

6.6 Implementation

We have implemented the hybrid deflation mechanisms, knee finder, deflation policies, and deflation-aware applications, as part of a deflation-aware cluster manager for VMs.

Our system is comprised of two main components. A centralized cluster manager implements and invokes the VM placement policies and generally controls the global-state of the system. In addition, we run local deflation controllers that run on each server. These local controllers control the deflation of VMs by responding to resource pressure, by implementing the proportional deflation policies described in Section 6.5. Both the centralized cluster manager and the local-controllers are implemented in about 4,000 lines of Python and communicate with each other via a REST API.

```
def Place(vm):
  if isHighPrio(vm):
    ok = PlaceInPool(vm, HighPriorityPool)
    if not ok:
      ok = ReclaimFromOverFlow(vm) #Deflate or Migrate
      if ok:
         PlaceInPool(vm, OverFlowPool)
      else:
         Reject(vm)
  elif isLowPrio(vm):
    ok = PlaceInPool(vm, ReserveCapPool)
    if not ok:
      ok = PlaceInPool(vm, OverFlowPool) #Can deflate
      if not ok:
         Reject(vm)

def PlaceInPool(vm, pool):
   server = GetFittestServer(vm) #best-fit, etc.
   if not server.CanFit(vm):
      return False
   return server
```

Figure 6.9: VM placement with multiple pools.

Deflation Mechanisms. Our prototype is based on the KVM hypervisor [127], and we use the libvirt API for managing VM lifecycles and for lower-level resource deflation primitives. Our hybrid resource deflation mechanisms presented in Section 6.3 are implemented by the per-server local controller. For hot-plugging (and unplugging) of CPU and memory, we rely on QEMU's agent-based hotplug. A QEMU hotplug agent runs inside the VMs as a user-space process, and listens for hotplug commands from the local deflation controller. The hotplug commands are passed to the VM kernel via this agent. This allows the hotplug to be "virtualization friendly". Unlike physical resource hotplug where unplug is a result of a fail-stop failure, the agent-based approach allows unplug operations to be executed in a best-effort manner by the guest OS kernel. This increases the safety of the unplugging operations. For example, if the guest kernel cannot safely unplug the requested amount of memory, the hot unplug operation is allowed to return unfinished. In this case, the memory reclaimed through hot plug will be lower, but the safety of the operation is increased.

Our hybrid deflation approach also uses hypervisor level multiplexing of resources. For this, we run KVM VMs inside cgroups containers, which allows us to multiplex resources. For CPU multiplexing, we adjust the cpu shares of the VM. For memory multiplexing, we limit the VM's physical memory usage by limiting the memory usage of the cgroup (mem.limit_in_bytes). Similarly, we throttle the disk and network bandwidth using the equivalent libvirt API's.

Knee Finder. We also record CPU performance counters on each server for inferring the performance utility-curve knee. We use the Linux perf tool to record the architectural counters (instructions_retired_any.p, cycles, and ref-cycles), and use the per-VM counting mode (:G). The knee-finder reads the counter values both for establishing a "baseline" performance fingerprint, and also after each deflation probe operation. We read and store the counter values once per second for each VM. Since each probe operation lasts about 30 seconds, this gives us 30 performance-counter samples to detect if they show a large deviation from the baseline.

6.6.1 Deflation-aware Applications

As noted in Section 6.3, hotplug-based deflation is explicit—change in resource allocation is visible to the guest OS and the applications. Many applications can adjust to changing resource availability and improve their performance by changing their scheduling policies, workload-intensity, and resource consumption. We develop deflation-aware versions of three popular applications that allow them to gracefully handle deflation, using simple policies and modest effort.

Memcached: Memcached is a popular user-space in-memory key-value store [7]. In conventional operation, the memcached server is started with a fixed, maximum cache size. Our deflation-aware memcached dynamically adjusts the maximum cache size based on the memory availability inside the VM. When shrinking the cache size, the memcached object eviction algorithm (LRU) is invoked. Our implementation is based on memcached v.1.3 and a previous dynamic memory-size version developed in [115], and comprises of about 500 lines of modifications to the memcached server. Shrinking the cache size may result in a lower object hit-rate, but avoids paging in memory pages from the slow swap disk. This modification allows memcached to serve more traffic even when the memory is deflated to below the cache size.

JVM: Garbage collected run-time environments such as the Java Virtual Machine can also react to deflation. A similar tradeoff exists between the performance degradation due to reducing the memory footprint vs. due to memory pressure. Reducing the heap size results in increased garbage collection overhead, but is nevertheless favorable to fetching pages from the swap disk. Prior work on JVM heap sizing have also explored this tradeoff [66, 222]. In particular, the JVM's object heap size can be adjusted based on the memory availability. We use IBM's J9 JVM [42] that has the ability to change the maximum heap size during run-time. We set the max heap size to the actual physical memory availability to avoid swapping. We implement this using the JMX API in an agent process in the guest in Java in about 30 lines of code. Our deflation-aware JVM allows the large class of JVM based applications to be made memory-deflation aware.

Mesos/Spark: Distributed data processing frameworks such as Spark [228] can also be made deflation-aware. These frameworks often run on top of cluster-management frameworks such as Mesos [113]. Mesos essentially provides resource offers in the form of executors to Spark, and Spark tasks run inside these executors. We develop a deflation-aware version of Mesos that affects Spark task placement and scheduling. Tasks running on deflated VMs can slow down the job, if other tasks on non-deflated VMs depend on them. These task-stragglers can hurt the overall performance of data-parallel applications like Spark [131]. Our deflation-aware Mesos, and by implication deflation-aware Spark, is designed to mitigate this task-straggler problem. In particular, we reduce the number of tasks scheduled on deflated VMs. We implement our changes as a small 20 line bash script which disables mesos executors on deflated VMs using the mesos master HTTP API.

6.7 Experimental Evaluation

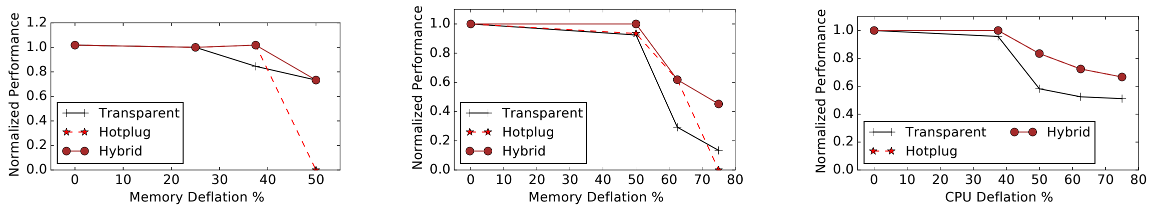
In this section, using testbed experiments and simulation, we show the performance of our deflation framework focusing on answering the following questions:

1. What is the performance impact of deflation mechanisms and deflation-aware applications?
2. What is the effectiveness of knee-aware deflation?
3. How do cluster policies impact cluster utilization and preemptions?

6.7.1 Environment and Workloads

We use the deflation-based cluster management system described previously in Section 6.6 to perform all our experiments. We run applications in KVM VMs running on Ubuntu 16.04 (Linux 4.10.3). Both the host and guest OS are x86-64. Our experiments are performed on a cluster of Dell R310 servers with 2-way Intel Xeon v4 E5-2620 CPUs, and a total of 16 cores per machine. We disable hyperthreading to reduce hardware interference. Each machine has 64 GB DDR4 memory, and a 1TB magnetic HDD, and we use 1GigE networking. The VMs are configured with 8 vCPUs and 8 GB memory, and run the same software configuration as the host OS. We run the following applications and workloads inside the VMs.

Memcached. We use memcached as a memory intensive workload. We drive memcached using YCSB's workload "B" configuration with 40 million records and operations, and a 95% read ratio [79]. The dataset is loaded into memory and uses



(a) Memcached memory deflation (b) Kernel-compile memory deflation (c) Kernel-compile CPU deflation

Figure 6.10: Hybrid deflation improves performance for both CPU and memory deflation.

around 4.5 GB in total. Since our VMs have 8GB allocated memory, the remaining 3.5 GB is used by the guest OS, disk, and network buffers.

Kernel Compile. We use the standard Linux kernel compilation benchmark as a CPU intensive workload, and as a workload that heavily relies on the OS buffer (page) cache for performance. We compile Linux 4.13 using four threads.

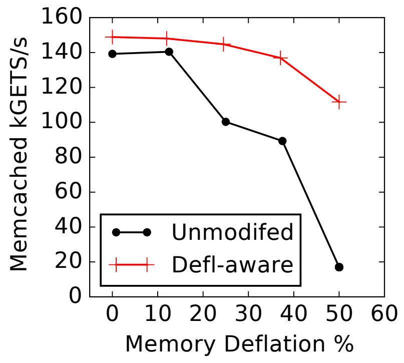
SPECjbb. We use SPECjbb 2015 with IBM's J9 JVM as a benchmark for Java server applications. We drive SPECjbb in the "fixed IR" mode, with a rate of 2000 requests per second. The IBM J9 JVM enables dynamic adjusting of JVM heap sizes. In this configuration, the jbb workload has a maximum heap size of 4.5 GB.

Spark. We use Spark v 1.6, running on Mesos 1.0. We use the Alternating Least Squares (ALS) machine learning workload on the MovieLens-large dataset.

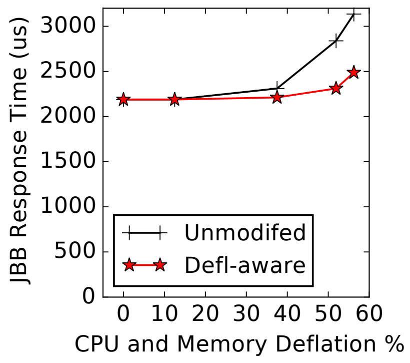
6.7.2 Deflation Mechanisms

The choice of deflation mechanisms developed in Section 6.3 can have an impact on application performance under deflation. We compare the application performance with the transparent, hotplug, and hybrid deflation mechanisms in Figure 6.10.

The effectiveness of hybrid deflation is highlighted in the case of memory deflation (Figures 6.10a, 6.10b). Memory hotplug allows the guest OS in the case of kernel-compile workload to shrink its memory usage by evicting items from its buffer cache—mitigating the effect of resource pressure, and resulting in a 2× improvement



(a) Memcached



(b) JVM (SpecJBB)

Figure 6.11: Deflation-aware application performance

in performance. While memory hotplug yields superior performance compared to host-swapping, unplugging beyond the safety threshold may result in applications getting terminated. We see this when kernel-compile is deflated by 75% in Figure 6.10b and memcached is deflated by 50% in Figure 6.10a. Since the hybrid deflation mechanism uses hotplug only below the safety threshold, it does not result in application termination.

For CPU deflation, Figure 6.10c shows the performance of the kernel compile workload. The hybrid approach uses CPU hotplug in this case, and outperforms transparent CPU multiplexing by 20%.

6.7.3 Deflation-aware applications

We now evaluate the effectiveness of making applications deflation-aware. In particular, we shall compare the performance under deflation of unmodified vs. deflation-aware variant of applications developed earlier in Section 6.6.1.

Memcached: Our deflation-aware memcached adjusts the size of the memcached memory usage based on the memory deflation levels. Figure 6.11a shows memcached performance (in GETS/s) for different values of memory deflation. With the cache-sizing modification, there is no significant decrease in performance up to 40% deflation, and at 50% deflation, it provides 6× the throughput of the unmodified version.

This shows the effectiveness of sizing an application-level cache like memcached to the memory availability. At high deflation levels, the unmodified version has to read some objects from swap, which is a slow operation bound by the disk-speed. Additionally, these slow GET requests (that hit swap), increase system load and decrease the overall throughput of the application. The deflation-aware memcached avoids this by sizing the cache to fit in the available memory, and sees a *higher* number of cache misses because it has evicted items that wouldn't fit in the memory available. But by doing so, it avoids swapping and obtains a much higher throughput, yielding a higher effective cache hit rate in terms of GETS/s.

JVM: Our deflation-aware JVM adjusts the size of the JVM memory heap, and reduces page swap-in rates when under memory pressure by increasing the frequency of garbage collection. Figure 6.11b shows the response times of the SPECjbb benchmark both with and without the deflation-aware JVM modifications. Deflation-aware JVM provides 20% better response times at 50% and 55% deflation.

6.7.4 Knee-aware Proportional Deflation

The performance of deflatable VMs is dictated by how much they are deflated by, which is in turn determined by the knee-aware proportional deflation policy. We now evaluate the effectiveness of this policy in terms of how application performance is impacted. In particular, we are interested in comparing against the utility-maximizing resource allocation policy, that determines the resource allocation based on utility curves, and provides the "optimal" resource partitioning that maximizes the overall performance of all the VMs. This utility-maximization approach needs full utility curves and is impractical in most settings. We also compare against a simple proportional deflation policy that is not knee-aware, and can inadvertently deflate VMs below their knee.

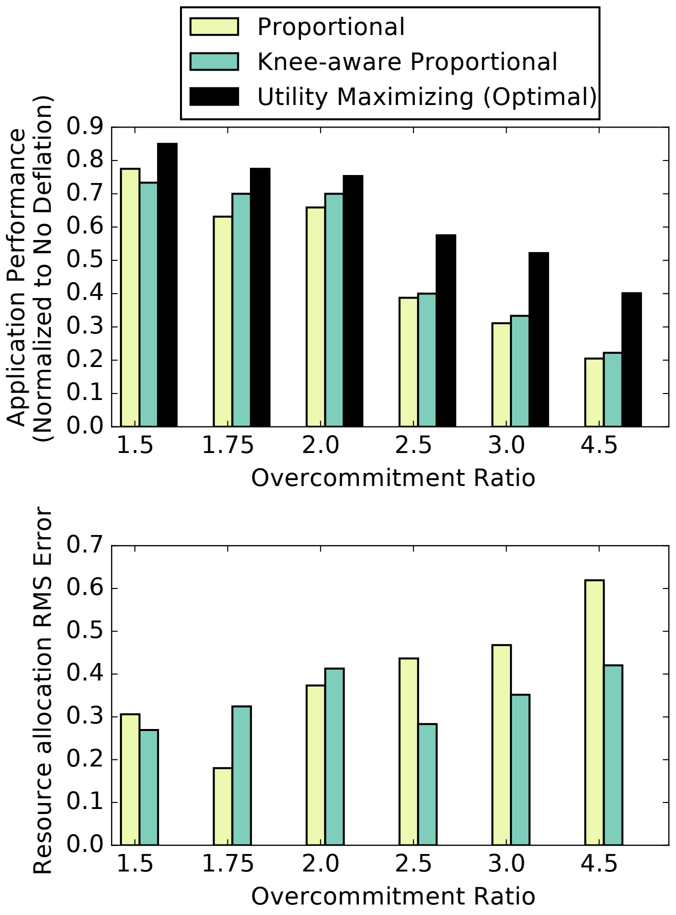


Figure 6.12: Compared to optimal utility maximization, performance of VMs with the knee-aware proportional deflation is within 10%-50% of the optimal.

Since the utility maximization approach requires the entire utility curve, for this experiment, we pre-generate utility curves for the memcached, kernel-compile, and SPECjbb applications. We evaluate the average performance of the applications (normalized to no deflation) in Figure 6.12 (top). We increase the resource pressure on the server by launching more deflatable VMs, which also increases the overcommitment ratio as VM deflation increases to accommodate more VMs. The memcached, kernel-compile, and SPECjbb applications are equally distributed among the VMs. The overcommitment ratio is the ratio of resources committed and resources available (server capacity), and an overcommitment ratio of 1x means that the server's resources are fully committed (i.e., no overcommitment).

Increasing overcommitment results in highly deflated VMs, which results in decreasing application performance, as we can see from Figure 6.12. Importantly, up to an overcommitment ratio of 2×, the knee-aware proportional deflation policy is within 18% of the optimal utility-maximizing approach. At higher overcommitment levels of 4.5×, knee-aware deflation is still within 50% of the optimal. The simple proportional deflation policy, which is not knee-aware, can result in VMs being deflated to below the knee (and even preempted). Thus the knee-aware policy shows higher utility than the simple proportional policy.

Since the difference in utility (performance) is due to the differences in resource allocation, in Figure 6.12 (bottom), we show the difference in resource allocation of the proportional and knee-aware proportional policy compared to the optimal utility-maximizing approach. For each VM, we compute the difference between allocation vectors produced by the proportional deflation policies ($\mathbf{r}$) and the utility-maximizing allocation ($\mathbf{r}_{\text{OPT}}$) as: $\text{RMS}(\mathbf{r}_{\text{OPT}}, \mathbf{r})$. We see that the root mean square error is generally *high*—up to 50% for the knee-aware proportional policy. However we note that even though the differences in resource allocations may be high, the difference in performance is low (20%). This is because although two resource allocations may have a large difference, their difference in performance is minimal due to sublinear nature of utility curves.

Result: *Even though knee-aware proportional deflation is utility agnostic, reduction in performance compared to the optimal is under 25% even at* 2.5× *overcommitment.*

6.7.5 Cluster Policies

To measure the impact of VM overcommitment on cluster-wide performance, we run a mix of VMs on an 8-node cluster running SpecJBB, kernel-compile, and memcached. We increase the number of VMs to increase the overcommitment, and observe the effect on VM performance. We define the goodput of a VM as is its performance normalized to no deflation, and so the cluster goodput is the aggregate across all VMs. Increasing the number of VMs to an increase in the overall cluster goodput, shown in Figure 6.14a. We see that the cluster overcommitment results in an increase in cluster goodput up to about 2× overbooking, where the goodput peaks at about 1.4× the non overcommitted cluster. This implies that we are able to get 40% more work out of the cluster. However we reach a tipping point, as the goodput drops at higher overcommitment levels, and the individual performance of VMs decreases sharply near the knee.

At a cluster-level, incoming VMs are placed onto servers using the VM placement policies described in Section 6.5.3. To evaluate the impact of different VM placement policies on large-scale clusters, we also implement a trace-based cluster simulator that allows us to exercise different policy parameters. We use the Eucalyptus cloud traces [23] to obtain VM arrivals, lifetimes, and VM sizes. We assign some fraction of VMs as low-priority VMs that are either deflated or preempted. We also use real application utility curves in these simulations, and use the knees obtained using the knee-finder. Thus in addition to simulating cluster-wide policies, this allows us to also study the performance effects of deflation at a cluster-wide level.

The placement of VMs onto servers affects the server overcommitment, the deflation levels of VMs, and in extreme cases of deflation, also the number of VM preemptions. We examine the server overcommitment and preemptions with the different placement policies. Figure 6.13 shows the results when using the "DS1" Eucalyptus cloud trace with 9,000 VMs on a simulated 100 node cluster. We assume that 50% of all VMs are low-priority and can be deflated, while the rest are high-priority non-deflatable VMs.

On the left, we see the overcommitment levels of different servers in the cluster. With deflation, our goal is to maximize the overcommitment of servers, while at the same time reducing the preemptions. All policies yield similar levels of server

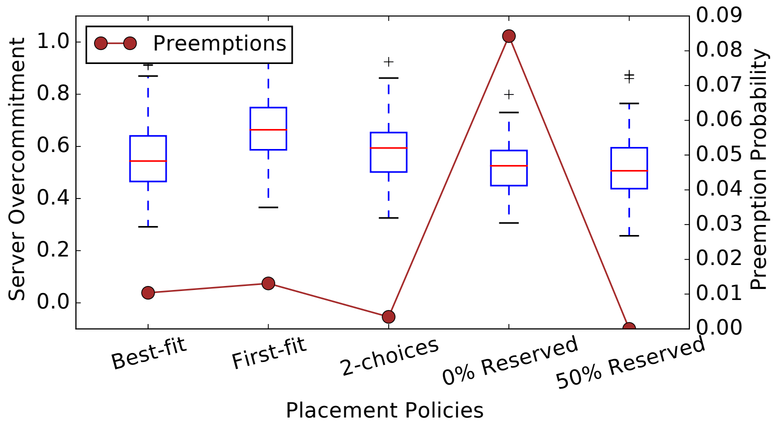


Figure 6.13: Server overcommitment and preemption probabilities (right axis) with different VM placement policies.

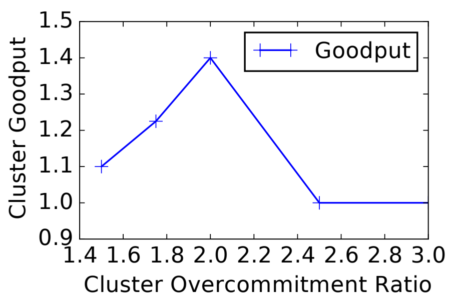
overcommitment. The differences in the placement algorithms are masked by the use of deflatable VMs, since “mistakes” in VM placement can be “fixed” by deflation.

Figure 6.13 also shows the effect of cluster management policies on the preemption probability. Without reserved capacity and without any limits on the high-priority pool, the preemption probability with the best-fit, first-fit, and 2-choices placement policies is under 0.01.

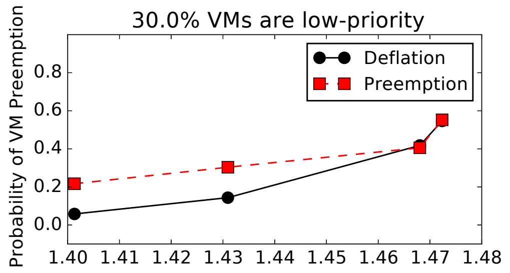
Since the distribution of low:high priority VMs is known (50:50), we can also control deflation/preemption using the reserved capacity and high-priority pool-capacity knobs. Specifically, we set the high-priority pool to be 50% of the cluster capacity. With 0% reserved capacity, deflatable VMs are under a risk of preemption, and VMs are preempted with a probability close to 0.08. Setting the reserved capacity to 50% removes the risk of preemption, since it relies on admission control of VMs.

Deflation vs. Preemption. We have proposed deflation as an alternative to preemption, and we now compare the techniques. VM preemptions disrupt application availability, in addition to fault-tolerance overheads.

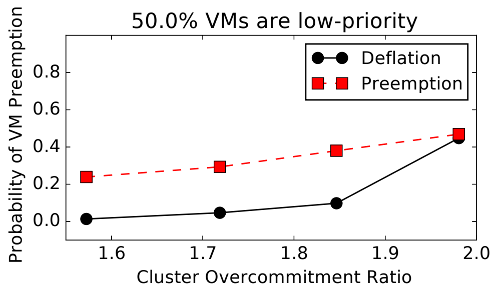
In these series of experiments, we again use the Eucalyptus trace and count the number of preemptions at different cluster sizes. We assume that the low-priority VMs



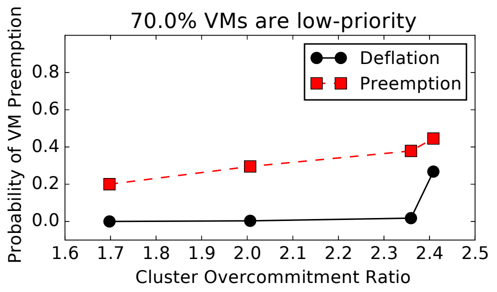
(a) Cluster Goodput



(b) Smaller fraction of deflatable VMs(30%) results in higher preemptions



(c) Deflation results in lower preemption probability



(d) Preemptions virtually eliminated if a high fraction of VMs are deflatable.

Figure 6.14: Probability of VM preemption with deflation vs. preemption-only.

use either preemption, or deflation (with the risk of getting preempted under extreme cases). As cluster overcommitment increases, the probability of preemptions increases for both deflation and preemption-only. The probability of deflation also depends on the fraction of VMs that can be deflated/preempted. If the fraction of VMs that can be deflated is low, then the average resource pressure faced by these VMs is high, and the high deflation can result in preemptions. This can be seen in Figure 6.14b, which shows the preemption probability when only 30% VMs are low-priority.

However, as the fraction of low-priority VMs increases, the resource pressure due to overcommitment can be absorbed by a larger number of VMs. This reduces the preemption probability with deflation. Figures 6.14c, 6.14d compare the preemption probability for 50% and 70% low-priority VMs respectively. We observe that the cluster overcommitment ratio can be increased as the fraction of the low-priority VMs increases, since we are able to reclaim more resources and drive up the overcommitment. When 70% VMs are low-priority (Figure 6.14d), then the preemption probability with deflation is zero for up to 2.3× overcommitment.

Thus deflation can reduce preemption probability when there are enough low-priority VMs to reclaim resources from, and can prevent preemptions *completely* even at overbooking levels approaching 2.3×. Alternatively, cluster sizes can be reduced by up to 2.3×, or accommodate a bursty load 2.3× the average arrival rate, without preempting applications.

6.7.6 Deflation in the Cloud

Currently, transient cloud servers are preempted upon resource pressure. We now consider a scenario where cloud operators use deflation instead of preemption. In particular, we evaluate the performance overhead when running on transient servers, which is primarily due to the overhead of periodic checkpointing for fault-tolerance.

We model preemptions and resource pressure in the cloud context by using Amazon EC2 spot prices. EC2 spot prices are dynamic and based on supply and demand via an auction mechanism, and thus are a good indicator of resource pressure, with higher prices indicating higher pressure. The magnitude of deflation depends on how high the price has risen above a bid price. We assume that even with deflation, VMs get preempted if spot price is $> 3\times$ bid-price—indicating that VMs can be deflated by up to 66%.

Figure 6.15 shows the performance overhead of running on three EC2 spot instances from June to August 2017. When using preemption only, the periodic checkpointing overhead can be as high as 60% compared to running on non-preemptible on-demand instances. With deflation, the overhead is halved in the case of the servers with high preemption rate. Importantly, the performance overhead due to deflation itself is less than 1.5%—while the rest of it is due to periodic checkpointing.

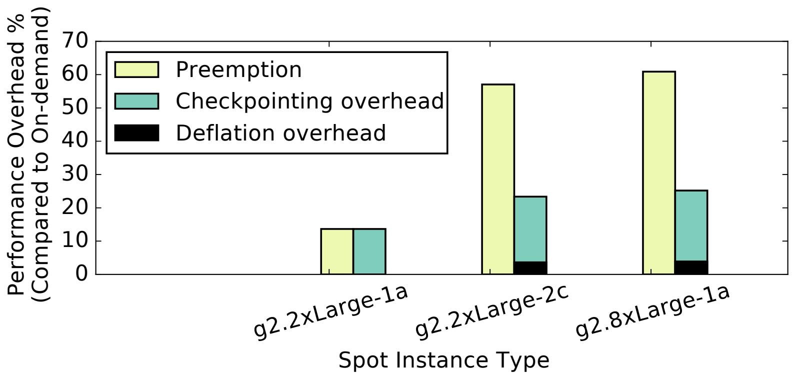


Figure 6.15: Performance overhead of fault-tolerance when using preemption and deflation

6.7.7 Summary and Discussion of Results

Our hybrid deflation mechanisms that combine hotplugging with transparent multiplexing can improve performance by up to 20% (Section 6.7.2). Modifications to make applications deflation-aware can improve performance under deflation by 6x (Section 6.7.3). CPU performance counters can be used to find the knees of application utility curves in a black-box manner (Section 6.4.1). Combined with knee-aware deflation, our cluster policies allow up to 2.3× cluster overcommitment without preempting VMs (Section 6.7.5). Deflation lowers the performance overhead of cloud transient VMs by more than 50% (Section 6.7.6).

Combined, the deflation mechanisms and policies provide many benefits. Using deflation allows applications, even unmodified interactive ones like SPECjbb, to run uninterrupted on low-priority transient resources. With preemption, this is only possible with costly fault-tolerance techniques such as continuous checkpointing [182]. While deflation does degrade performance, the degradation is relatively small when the utility curves are sublinear, which is often the case. Furthermore, simple modifications to applications can greatly enhance their performance under deflation. For example, SPECjbb's performance is reduced by less than 10% at 50% deflation as shown in

Figure 6.11b—a relatively small price to pay for uninterrupted operation, even during resource pressure.

In addition to providing availability benefits to users and applications, we believe that deflation is also feasible and beneficial to implement in data centers and clouds. We have shown that deflation can provide high cluster overcommitment (2.3×), and can be implemented effectively through hybrid mechanisms and black-box knee-finding approaches. Thus deflation is a practical technique that allows more application types (batch and interactive) to use low-priority resources.

6.8 Related Work

Our proposed deflation system draws upon many related techniques and systems.

Systems for Transiency. Current transient servers in the cloud offer significant cost savings (upto 90%), but are preemptible in nature. Running applications on cloud transient servers involves using a combination of fault tolerance and resource allocation policies, to mitigate the performance and cost effects of preemptions. Prior work has focussed on system [182, 197] and application [176, 180, 142, 224, 108, 209, 212] support for handling preemptions. Deflation is motivated by the need to avoid the performance, development, and deployment costs associated with preemption.

VM Overcommitment Mechanisms. Our deflation mechanisms rely on efficient VM overcommitment, which have been well studied and optimized to allow data center operators to pack more VMs onto their physical servers. Memory overcommitment typically relies on a combination of hypervisor and guest OS mechanisms, and has received significant attention [205, 48, 181]. Our hybrid memory deflation combines the use of memory hotplug and hypervisor swapping. Memory ballooning is another memory overcommitment technique with generally inferior performance to hotplug [120, 135]. The use of hotplug has also been proposed for reducing energy consumption [232].

Our use of CPU hotplugging is partly motivated by mitigating lock-holder preemption problems in overcommitted vCPUs [88, 158].

Virtual Cluster Consolidation. Using dynamic VM resource allocation and VM migration is common [216] to increase cluster utilization. VMWare's distributed resource scheduler [103] uses per-VM reservations (minimum limits) and shares for dynamically allocating resources—similar to our resource-pressure based local deflation policies. Crucially, part of our contribution is inferring the minimum limits (reservations) using performance-counter based knee-finding, as well as using proportional deflation without user-specified shares. Many approaches for performance-sensitive resource allocation among co-located VMs have been tried [134, 109, 242, 156, 104], but they assume some application performance model, which we do not. VM memory allocations can be set using working-set estimation [235, 74, 239], utility-maximzing [115], or market-based approaches [45, 59].

Knee Finding. Finding the knee or the inflection point of utility curves has a long line of related work—JAWS [102] uses the design of experiments to find the point on the knee, and Kneedle [171] uses curvature based techniques to specify and find knees. Curvature based techniques can certainly be applied in our context. However, since our knee finding is "online" and cannot have a profiling stage, the number of different points that we can infer application performance on, is small (~5). This makes curvature based and other techniques more challenging to apply. However, a more principled approach to knee-finding is part of future work.

Black-box knee finding is a far more daunting, due to the heterogeneity of applications, hardware and software configurations, etc., which make modeling application performance using externally visible metrics challenging. Our use of hardware performance counters to estimate performance knees is based on research in using performance counters to identify performance interference [157, 236, 87]. We emphasize that in our context, we only seek to identify a large change *change* in performance counter values, since building a general performance model for arbitrary applications is extremely challenging.

Elastic Auto-scaling. Vertical elastic scaling allows VM resources to be dynamically adjusted to react to changing workloads, to meet some target SLA [138]. Elastic scaling often uses control-theoretic [121] and other autonomic-computing techniques [46]. Elastic scaling is typically applied on a per-application or a per-server level, whereas deflation is a cluster-wide technique. Application performance models and workload prediction is a key component of elastic scaling [100, 155, 160, 187]. In contrast, deflation is a black-box, application agnostic, and reactive technique for handling resource pressure. Vertical elastic scaling has mostly been focused on CPU elasticity and uses existing VM overcommitment mechanisms such as multiplexing. Our deflatable VMs use a combination of overcommitment mechanisms that are adapt to application resource usage, and we consider the simultaenous deflation of *all* resources. Deflation also exposes an explicit performance tradeoff, whereas elastic scaling approaches typically only reclaim unused resources.

Elastic Applications. Deflation-tolerance and resource-elasticity are important attributes found in many applications. We have shown that many applications and operating system components are deflation-tolerant by default, and simple modifications can increase this tolerance. Dynamic heap sizing [66, 70, 222] is a popular technique for improving memory-elasticity of applications. The memory elasticity of data-parallel applications is enhanced in [117, 96]. Application-level ballooning [169] can also improve memory footprint of virtualized applications. Applications can also respond to deflation by serving less optional content [128], by reducing the quality of their results [195], or by giving them incentives for improved efficiency [174, 59].

6.9 Deflation Summary

In this chapter we proposed the notion of resource deflation as an alternative to preemption, for running low-priority applications. Deflatable VMs allow applications to continue running even under resource pressure, albeit at a lower performance. Our hybrid deflation mechanisms that use resource hotplugging minimize the performance impact of deflation. The performance-counter based knee-finding, along with proportional deflation policies, ensures that VMs are deflated to safe levels while maximizing overcommitment. Our mechanisms, policies, and deflation-aware applications allow cluster resources to be overcommitted by up to 2.3×, and result in 50% lower performance overhead compared to preemption in cloud spot markets.

CHAPTER 7
SUMMARY AND FUTURE WORK

7.1 Thesis Summary

Transient resource availability represents an exciting and important resource allocation model. In this dissertation, we have demonstrated the usefulness of the transiency-specific mechanisms and policies. This thesis explores the challenges posed by transient servers, and proposed fault-tolerance techniques and resource-management policies that enable a wide range of applications to make effective use of transient servers, especially in public cloud environments. We developed several new techniques that combine fault-tolerance techniques with transiency-specific resource management, that enable a wide range of applications to make effective use of low-cost cloud transient servers. We develop four systems that demonstrate the new contributions in fault-tolerance, transient resource allocation, cloud abstractions, and transient resource reclamation:

Interactive Services On Transient Servers. The revocable nature of transient servers leads to frequent downtimes for long-running interactive applications such as web services. We developed fault-tolerance mechanisms and risk-management policies as part of a derivative cloud called SpotCheck, which allows unmodified interactive applications to make use of cloud transient servers with minimal downtime and at low cost. SpotCheck is a derivative IaaS cloud that offers low-cost, high-availability servers using cheap but volatile servers from a native IaaS platforms. To do this, SpotCheck simultaneously ensures high availability, reduces the risk of mass server revocations, maintains high performance for applications, and keeps the costs down. We designed

SpotCheck to balance these competing goals. SpotCheck is able to provide more than four 9's availability to its customers, which is more than 10× that provided by the native spot servers. At the same time, SpotCheck's VMs cost nearly 5× less than the equivalent on-demand servers

Batch-Interactive Data-Intensive Processing On Transient Servers. Chapter 4 looked at the challenges of another class of applications—batch-interactive distributed data processing. I proposed the use of periodic application-level distributed checkpointing developed policies for server selection that minimize both fault-tolerance overheads and computation costs. The Flint system extends Spark with the aforementioned checkpointing and server selection tasks, and runs unmodified batch and batch-interactive applications on low-cost cloud transient servers while minimizing the performance degradation due to revocations.

Portfolio-driven Transient Resource Management. Server portfolios, proposed in Chapter 5, represent a generalization and evolution of the resource management policies proposed in the earlier chapters. Adapting portfolio theory to the transient server context allows the efficient construction of heterogenous transient sever clusters, that can be tailored to different applications. We implemented server portfolios as part of the ExoSphere system, which is a transiency aware cluster manager based on Mesos. Portfolios, along with ExoSphere's transiency-API, allow a wide range of distributed applications to define their own fault-tolerance and risk-management policies.

Resource Deflation For Transient Resource Reclamation. Finally, this thesis also asks the question whether it is possible to achieve transient resource allocation through different mechanisms, and not just through server preemption/revocation. Chapter 6 proposes and discusses one such approach: resource deflation, which is a technique for transient resource reclamation. Deflation generalizes revocation, and trades off availability for reduced application performance. Our deflation-based cluster manager extends virtual machine overcommitment techniques to reduce the overhead of reclaiming resources; uses hardware performance counters to infer the lower limits of deflation; and uses proportional deflation policies to manage multiple low and high priority applications. Deflation increases server utilizations, and allows cloud and data center operators to increase the resource overcommitment of their computing infrastructure.

7.2 Broader Applicability Concerns

In this section, we discuss some of the concerns about the broader applicability of the work proposed in this thesis. A large part of this thesis examines challenges and solutions for running different classes of applications on top of cloud transient servers. The low-cost transient servers offered by large public cloud platforms provide a good environment within which to examine and address these challenges. However, some of the specific characteristics of contemporary transient server offerings can influence the design of transiency-mitigation solutions and systems. Since transiency arises in other contexts (such as energy efficient and enterprise data centers), it is important to have general solutions that do not overly rely on the quirks and vagaries of cloud transient servers. For example, although exploiting a pricing quirk such as EC2's "free partial hour if revoked" can indeed lower costs [108, 106], such techniques may not carry over across time and cloud platforms, if these peculiarities are discontinued or not adopted by other cloud platforms. We have been deliberate in our attempts to minimize the use of such quirks—nevertheless due to the applied nature of our work, some concerns can remain. Below, we list some of the key assumptions made throughout this thesis, and comment and speculate on their validity in other contexts:

Revocation Warning: SpotCheck's bounded-time live migration relies on the presence of a small revocation warning. We believe that revocation warnings will continue to exist in cloud and data center environments, since transient server unavailability is triggered by resource pressure, and distinct from sudden unavoidable fail-stop failures. Green data centers could expose the UPS warning to applications, for instance.

Price/Availability Information: Flint leverages publicly available price and availability information (mean and correlations) provided by Amazon EC2, to minimize the cost and performance overhead of running applications on transient servers. However, this information may not be directly available in all cases—Google's and Microsoft's transient servers do not provide such information for instance. However, we speculate that to incentivize transient server use, public clouds ought to provide more details about the availability and mitigate the abruptness of revocations. For instance, Amazon recently updated their EC2 spot instance pricing to take into account longer term supply/demand characteristics and have lower volatility [57, 162]. This limitation does not apply in the case of private data centers that maintain detailed availability and revocation information, and systems like Flint can be used as intended.

Multiple Heterogeneous Server Types: One of the major assumption in SpotCheck, Flint, and ExoSphere is the presence of a large number of heterogenous server types that do not all share the same fate. That is, we assume that revocations are not completely correlated for all the servers, and that we can exploit this to reduce risk by diversifying. In large public clouds that have to provide different services to customers with different needs, it seems likely that this heterogeneity in usage patterns will continue to exist. In smaller data centers, resource management policies that operate on a per rack/pod/zone level may provide the required heterogeneity. Moreover, information gathered by private data center operators about the utilizations, workloads, etc., may allow us to treat individual hardware servers as distinct "markets" with different availability characteristics.

Non-adversarial Resource Provider: Finally, most of our systems basically serve as "middleware" that implement transiency-mitigation policies, and whose operation does not require any special co-operation with the underlying cloud platform. Our work assumes that the cloud platform will not actively hinder deploying applications using our systems and techniques. Transient resources only increase utilization and are good for both the resource provider and consumer, and we see very little reason why providers should violate their neutrality. We have not looked at the "second order effects" of our proposed systems. While it is true that increased use of transient servers will drive up their price and/or lower their availability, it must be noted that transient and on-demand servers come from the same pool of servers. Thus an increase in use of transient servers should see a corresponding decline in use of on-demand servers, thereby increasing the the supply of transient servers. However, it is possible that increasing popularity of transient servers results in higher volatility, and thus, more frequent revocations, reducing their appeal. However, we have shown that our fault-tolerance techniques even work with revocation rates that are 10× higher than current revocation rates, and increased volatility is a concern only for transiency-oblivious systems.

Finally, resource deflation presented in Chapter 6 looks at transiency from the perspective of resource reclamation, and none of the above concerns apply. Since virtualized clusters already support overcommitment mechanisms, we argue that resource deflation is a general purpose technique that can be adopted by both public clouds and private data centers. While we showed that deflation is better for application availability and performance, and even increases cluster utilization, it is certainly more complex to implement for the resource provider compared to preemption. However, we argue that increased utilization and application experience justifies this extra complexity, and the increasing robustness of overcommitment mechanisms makes deflation an appealing option. We are already seeing public clouds offer dynamic resource allocation in the form of burstable instances [209], paving a path for deflation-like techniques.

7.3 Future Work

Transiency mechanisms such as deflation may also need to be extended and generalized, especially in green data center environments, where their applicability has yet to be considered.

7.3.1 Applications

As transient availability becomes more pervasive, it may necessitate incorporating transiency support in a wider gamut of applications.

Distributed Machine Learning. Transient servers can provide the massive amounts of computational resources required by deep learning pipelines for tasks such as image and speech recognition, at low costs. Current deep learning applications require a large amount of model training on large clusters. However, transient server revocations lead to a loss of the in-memory state (the model parameters), and causes unnecessary slow-downs. Incorporating transiency awareness into distributed machine learning frameworks such TensorFlow [43] involves many challenges:

- Handling the effect of revocations on distributed data flow applications.
- How the asynchronous nature of machine learning algorithms interacts with varying degrees of parallelism due to revocations.
- Model checkpointing policies that go beyond periodic checkpointing, and take into account the program graph structure explicitly.

Load Balancing. Another popular class of applications that can run on transient servers are clustered web services that use load balancers. Such applications also require strict quality of service (QoS) guarantees, which is challenging due to the fluctuating cluster sizes due to revocations. Such a system would require heterogenous server selection policies to minimize large numbers of concurrent revocations, as well as application performance models that capture the effect of intermittent (partial) cluster failures.

7.3.2 Techniques

Another major direction for future work is to enhance some of the policies developed as part of this thesis.

Lazy Checkpointing For Spark. The Flint system uses periodic checkpointing of Spark RDD's. Although we showed that this checkpointing overhead is small, and reduces the program running time by less than 10%, an alternate approach is to instead only checkpoint RDDs *after* the revocation warning. Such a lazy checkpointing approach is not generally safe, since the time to completely write a checkpoint depends on the size of the RDD and the disk write speeds. However, it would be interesting to explore under what scenarios such a lazy checkpointing approach would be safe, and compare the performance with the current periodic checkpointing policy.

Online Portfolio Construction. The ExoSphere system uses the classic Modern Portfolio Theory to construct server portfolios. However, many alternate techniques for portfolio construction exist. One promising direction for improvement is portfolio rebalancing: adjusting the portfolio periodically based on changing market characteristics. Universal portfolios [80] using no-regret learning are potentially viable approaches for constantly rebalancing portfolios [196], which may be required in highly volatile market scenarios where the covariances cannot be assumed to be stable over time.

Deflation-aware Distributed Applications. The resource deflation paradigm proposed in the previous chapter allows unmodified applications to run on transient (but not necessarily preemptible) resources. Just as revocation-aware applications such as Flint provide significant benefits over their transiency-oblivious counterparts, investigating deflation-aware applications can also be a fruitful endeavor. In particular, distributed applications for data processing and scientific computing (such as Spark and MPI respectively) can be made deflation-aware. Deflation-aware applications have an option of voluntarily relinquishing resources under resource pressure, allowing them to be strategic about their overall resource footprint.

APPENDIX

THE ROLE OF BIDDING IN TRANSIENT CLOUD SERVER MARKETS

Cloud platforms now sell surplus idle server capacity at discounted prices to users to gain additional revenue. In some cases, transient server pricing can be dynamic and be governed through supply and demand. Amazon EC2 uses a market mechanism to sell this capacity where users place a bid for servers, and EC2 allocates them if the bid is higher than the spot price, which varies continuously based on supply and demand. When the spot price rises above a user's bid price, EC2 revokes the servers. EC2 determines the spot price by running a sealed-bid multi-unit uniform price auction [62]. Note that the underlying supply of surplus servers in the spot pool also changes, since EC2 may take resources from the spot pool to allocate new on-demand or reserved instances. Thus, the spot price changes dynamically both as users submit new bids, and as the spot pool's capacity changes.

Amazon conducts a second-price auction for their spot instances. Users place a single, fixed bid, which represents the maximum hourly price that they are willing to pay. The market price is based on all the bids and the available supply. Importantly, all users pay the same market price, which may be lower than the bid. If the market price increases above the user's bid, then the spot instance is revoked after a small (120 second) warning.

Spot price dynamics and the potential of unexpectedly losing resources introduces additional new complexities, which applications are typically not designed to handle. Addressing these complexities is an active research area. In particular, there has been substantial research on “optimal” bidding strategies for various applications and scenarios [194, 198, 227, 240]. In general, a bidding strategy determines the lowest bid price that ensures an application satisfies a performance target with high probability, e.g., finishing within a deadline. EC2 publishes three months of spot price history—and there are archives over multiple years—which prior work analyzes extensively to model price characteristics [119, 149, 210, 219, 234].

Designing bidding strategies can be highly complex, especially if a workload is distributed and users have to bid on many resources. In this case, requesting multiple units of the same resource with the same bid is risky, since all resources are governed by the same spot price, such that if one resource is revoked, they all are revoked. To reduce the probability of concurrent revocations, users might either spread their requests across many different resource types with different spot prices or place many different bids for different units of the same resource type. Bidding's complexity may be one reason why, despite its extremely low prices (50-90% less than on-demand instances), the spot market has low utilization [112].

EC2's cloud has attempted to reduce complexity by introducing tools, such as SpotFleets, which enable users to specify bidding policies that apply to large groups of resources from different markets. SpotFleets also includes default bidding policies for users that do not want to design their own policy. However, while bidding is a complex problem in theory, we argue that it is not a significant problem in practice due to at least three reasons.

Wide Range of Optimal Bids. Our spot price data analysis shows there is a wide band of bid prices that all yield optimal results, such that any bid within this range has a similar cost and availability as highly sophisticated bidding strategies. One reason this is not readily apparent is that prior work often compares the cost and performance of a bidding strategy to using higher-priced on-demand servers. However, in today's market, with low and stable prices, bidding strategies need not be sophisticated to reap significant savings compared to on-demand servers. Related work should instead compare their performance and cost with "dumb" bidding strategies.

Resources Always Available. Due to the large number of spot markets and their size, there are always many markets available where prices are low and stable, even when some markets are experiencing price spikes. Hence, upon revocation, a simple strategy that provisions a new spot server in another spot market and migrates an application to it is better than waiting for a spot price spike to subside. This migration approach nearly eliminates the unavailability of spot servers and reduces the practical impact of using bidding as a tool to control availability.

No Penalty for High Bids. Current spot market rules permit users to bid the maximum allowed bid price within each market with no penalty. Thus, sophisticated users can ensure extremely high availabilities on spot instances by placing maximum bids with little or no probability of paying a high price if the spot price were to rise.

Finally, not only do different bidding strategies yield little difference in their performance and cost, but some of our insights above are reflected in the default bidding strategies for EC2's SpotFleets tool [22]. Thus, Amazon is already nudging users to employ simple bidding strategies [20]. Based on these insights, we argue that users should ignore the potential complexity of bidding, and simply procure cheap EC2 spot servers using simple bidding strategies that we outline (or using Amazon's tools to employ such strategies). Rather than focusing on bidding, researchers should instead focus on modifying applications i) to gracefully handle unexpected resource revocation and allocation and ii) to efficiently seek out and migrate to the lowest cost resources. Selecting the best spot server to use at any time, i.e., the one with the lowest cost and best performance, is the primary problem that applications must address when using variable-priced resources. That is, if a resource's price rises significantly, then applications should be flexible enough to simply migrate to lower cost resources elsewhere in the cloud. For applications willing to adopt it, this approach can yield significant cost savings with little performance impact.

A.1 Related Work

Since EC2 introduced its spot market, there has been significant research both on analyzing and modeling spot prices and developing bidding strategies based on real data and models. One of the first papers analyzing spot price data raised questions about whether EC2's mechanisms for setting the spot price were market driven [62]. However, as the authors note later, the characteristics of the spot price changed, making it consistent with a market driven allocation [62]. A number of related papers also analyze spot price data to better understand its statistical characteristics [119, 149, 198, 210, 219, 234]. Analyzing and modeling spot price data is a prerequisite to developing bidding strategies that select the optimal bid to ensure a target level of performance at the minimum cost.

Recent work focuses on optimal bidding for MapReduce jobs. In [240], the authors focus on selecting a bid such that, with high probability, the completion time on spot instances is less than twice the running time on on-demand instances. The paper examines multiple scenarios: quitting job execution upon revocation, or making persistent requests, i.e waiting until price drops to resuming execution. In all variants, the work only considers bidding in a single spot market: if the price rises too high and instances are not available the MapReduce job must either quit or continue processing with fewer resources.

As we discuss, EC2 (and the cloud in general) is large enough that resources are nearly always available somewhere. Thus, unless an application is highly optimized for specific types of server architectures (which MapReduce is not) or has geographical constraints, waiting for the price of resources to drop is unnecessary. Related work makes similar assumptions about market constraints but focuses on different applications. For example, prior work develops bidding strategies for jobs with deadlines [227], such that it chooses a bid for a particular spot market so the job finishes before its deadline with high probability.

Restricting the problem to only a *single* spot market has also resulted in prior research focusing on the wrong price characteristics. Specifically, if restricted to a single spot market, the only important characteristic is availability, or the percentage of time the bid price is below the spot price. However, if we assume applications should not restrict themselves to only a single spot market, then availability is no longer important, as other cloud resources are available in other markets. In this scenario, the frequency of revocations is the primary attribute that affects performance, since every revocation incurs an overhead to request a new instance and migrate to it.

Unfortunately, modeling revocations is not as straightforward as modeling availability. Modeling availability simply requires fitting a probability density function (PDF) to a histogram over different spot prices, which gives a probability the spot price is equal to a particular value. The corresponding CDF then directly gives availability, which is equal to the probability the spot price is above a given value. Prior work models availability using both Pareto and exponential distributions [240]. In contrast, revocations are discrete events with inter-arrival times that are not cleanly captured by a single number. As in any queuing model, the distribution of inter-arrival times is also important. However, the frequency and distribution of revocation events is a function of the bid, and may be different at different bid prices. Even though revocations are the primary attribute that affects performance, we know of no prior work that models the distribution of these events at different bid prices in EC2.

Finally, in many cases, as in [194, 227], bidding strategies are with respect to idealized spot price distributions, e.g., mixed Gaussian, exponential, Pareto, etc., and not real data. These idealized models are often based on examining only a few markets even though thousands of spot markets exist, which have vastly different characteristics.

These characteristics are not likely captured by a one-size-fits-all model. Further, as [62] notes, price characteristics may change frequently due to changes in EC2's supply, demand, or its pricing algorithm, which may render models based on prior data unreliable. In many cases above, proposed solutions actually depend on the type and attributes of the particular model used in the analysis. As we discuss, though, the bidding problem in today's market (and possibly in future markets) is a red herring that is not particularly important for maximizing performance and minimizing costs using spot instances.

A.2 Do Optimal Bidding Strategies Matter?

To understand whether (and how much) optimal bidding strategies matter in EC2, we conduct a data-driven analysis of spot price data over a six month period from March to August 2015 (and longer periods where stated), as well as show aggregate statistics from *every* EC2 spot market. For ease of exposition, we focus on the most popular instance types in the most popular region, i.e., Linux instances in the us-east-1 region.

Bidding strategies optimize the cost-availability tradeoff for spot instances: as a user increases their bid, they may pay more per-hour, but their availability also increases. However, spot price data across many markets shows that there is a wide range of "optimal" bids that essentially yield the same availability for the same cost. To illustrate, Figure 2.4(a) shows a CDF of availability for instance types in five different markets over our six month period, where the x-axis is a user's bid normalized to the on-demand price, i.e., 1 is 1× the on-demand price, 2 is 2× the on-demand price, etc. As expected, availability monotonically increases with the bid. However, in each case, the CDF has a steep incline followed by an extremely long tail, such that there is little increase in availability after some bid threshold and only bids that fall within the steep range of the incline yield different availabilities. As the graph shows, this range of bids is quite small, providing only a narrow window where changing a bid will have a significant effect on availability.

Similarly, Figure 2.4(b) shows the cost a user would pay for the same instance types and the same bids. In this case, the cost on the y-axis is a fraction of the on-demand cost, i.e., 0.5 means the expected cost is 0.5× the on-demand price. As with availability, the cost is monotonically increasing with the bid amount. However, just as with availability, the cost curve has a long tail, such that higher bids result in little or no increase in cost. The only exception in these markets is the c3.xlarge instance type, which experiences two abrupt increases in cost at bid levels of 1.2× and 4.75× the on-demand price. The other instance types have nearly the same cost regardless of the bid level. This occurs because most markets always have a low and stable spot price, with the average spot price <0.2× the on-demand price. Just as with availability, bidding has little effect on the cost of spot instances.

Finally, as we discuss in the previous section, the frequency of revocations, as indicated by their mean-time-between-revocations (MTBR), is another important metric, since revocations incur overhead for applications that migrate to other available resources. Thus, Figure 2.4(c) shows the MTBR for different bids. The figure shows that MTBRs range from tens to hundreds of hours. In addition, the MTBRs also have a long tail in all but one market, such that bidding high does not significantly increase the MTBR and there is a wide range of bids that effectively yield the same MTBR.

While the analysis above uses only five spot markets as illustrative examples, we analyzed these properties in over 1500 spot markets over our six month period. Figure A.1 plots the range of bids such that any bid within the range is within 10% of the optimal bid for availability, cost, and MTBR. The optimal bid is simply the bid that yields the highest availability and MTBR for the lowest cost. Thus, we consider every bid within the range as effectively optimal that yields near the same result. As above, the y-axis is the length of the bid range as a factor of the on-demand price.

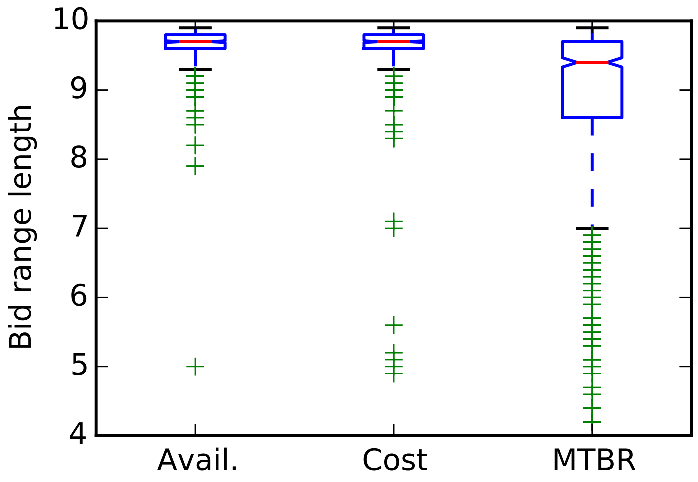


Figure A.1: Range of bids for which availability, cost, and MTBR is within 10% of optimal across 1500 markets.

Thus, a bid range length of 2 indicates a range of $[b, b + (2 * D)]$ for some bid b where D is the on-demand price. A smaller range indicates higher bid sensitivity, where an application should carefully select a bid from a small range of near-optimal bids. In contrast, a larger range indicates a low bid sensitivity.

We see from Figure A.1 that the bid ranges for the availability, cost, and MTBR are generally quite large, with a bid range near 9. Note that EC2 imposes a maximum bid of 10× the on-demand price. These results suggest that picking nearly any bid within the range of allowed bids yields the same optimal result. Put another way, users would need to "try hard" to make a "bad" bid by selecting a bid price that is exceedingly low compared to the average spot price. *Thus, in today's market, due to low prices (resulting in high availability) and price stability (resulting in long MTBRs), spot revocations are rare, but unavoidable, regardless of a user's bid.*

A.3 Future Of Spot Markets

Will markets get more volatile? We have examined price data over the past six years (in addition to our six month traces) and found that bidding has never been

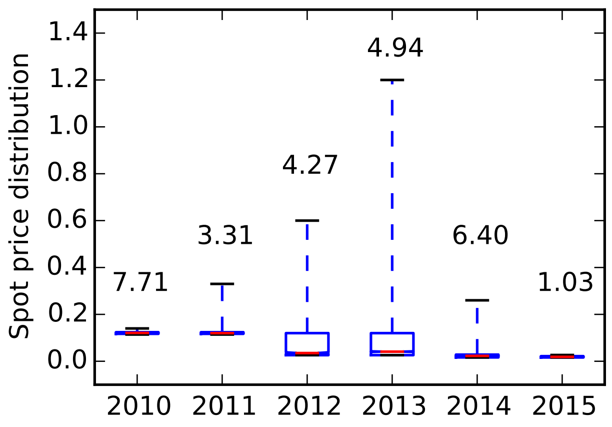


Figure A.2: *Spot price distribution for `m1.large` over the years. The number above each boxplot denotes the skewness of the distribution.*

a significant problem throughout the history of EC2's spot market. For example, as shown in Figure A.2, while the average spot price of the `m1.large` instance type since its inception has decreased (in accordance with decreasing on-demand prices), the spread of spot prices has not increased significantly either. However, while our analysis of historical spot price data leads us to conclude that bidding has never been an important problem, it is possible that it may become an important problem in the future if price characteristics change.

Will prices rise? A substantial increase in demand will undoubtedly cause an increase in average spot prices and any substantial price increase will cause price-sensitive spot users to become "priced out" of the market (which in turn may reduce demand and cause prices to drop). The second-order effects due to widespread adoption of the migration strategies we propose remains unclear, and a rigorous analysis, through game-theoretic or other means, is an open question. However, anecdotal evidence suggests that such effects may not come to pass—due to the significant capacity additions being made by all cloud providers on a regular basis, implying that there may always be some surplus capacity despite increasing demand in both the spot and on-demand markets.

Fixed rate. Google and Microsoft sell surplus preemptible servers at a fixed discounted price. This predictable pricing may benefit some users, since even Amazon has launched similar “Spot blocks” [21] instances, which have a guaranteed lifetime of six hours, but come at an extra cost (though still cheaper than the on-demand instances). Although researchers have argued that auctions are better at maximizing revenues than using fixed discounted prices, their applicability in the cloud domain is unclear. Specifically, spot and on-demand servers come from the same resource pool, and the cloud operator is likely to be more interested in increasing revenues from higher-priced on-demand servers that trying to maximize incremental revenue from much lower-priced surplus servers (either through auctions or fixed discounts).

BIBLIOGRAPHY

[1] Alibaba cloud. `http://www.alibabacloud.com`.

[2] Amazon ec2 instance offerings. `https://aws.amazon.com/ec2`.

[3] Amazon web services - cloud computing services. `https://aws.amazon.com/`.

[4] Google cloud computing. `http://cloud.google.com`.

[5] Ibm cloud. `http://www.ibm.com/cloud`.

[6] Joyent public cloud. `http://www.joyent.com`.

[7] Memcached. `https://memcached.org/`.

[8] Microsoft azure cloud computing platform and services. `http://azure.microsoft.com`.

[9] QEMU Microcheckpointing. `http://wiki.qemu.org/Features/MicroCheckpointing`.

[10] SPECjbb2005. `https://www.spec.org/jbb2005/`.

[11] TPC-W Benchmark. `http://jmob.ow2.org/tpcw.html`.

[12] Google's Green PPAs: What, How, and Why. http://www.google.com/green/pdfs/renewable-energy.pdf, April 2011.

[13] Heroku. http://www.heroku.com, May 1st 2014.

[14] PiCloud. http://www.multyvac.com, May 1st 2014.

[15] RightScale. http://rightscale.com, May 1st 2014.

[16] Single Root I/O Virtualization. https://www.pcisig.com/specifications/iov/single_root/, May 1st 2014.

[17] Amazon EC2 Spot Instances. `https://aws.amazon.com/ec2/spot/`, September 24th 2015.

[18] Amazon Elastic Map Reduce for Spark. `https://aws.amazon.com/elasticmapreduce/details/spark/`, June 2015.

[19] Docker. `https://www.docker.com/`, June 2015.

[20] Ec2 spot bid advisor. https://aws.amazon.com/ec2/spot/bid-advisor/, September 2015.

[21] Ec2 Spot Blocks, October 2015. https://aws.amazon.com/about-aws/whats-new/2015/10/introducing-amazon-ec2-spot-instances-for-specific-duration-workloads/.

[22] Ec2 spot-fleet. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html, September 2015.

[23] Eucalyptus workload traces. https://www.cs.ucsb.edu/~rich/workload/, 2015.

[24] Google preemptible instances. https://cloud.google.com/compute/docs/instances/preemptible, September 24th 2015.

[25] Livejournal Social Network Dataset. https://snap.stanford.edu/data/soc-LiveJournal1.html, June 2015.

[26] Lxc. https://linuxcontainers.org/, June 2015.

[27] Openstack. https://www.openstack.org, June 2015.

[28] Transaction Processing Performance Council - Benchmark H. http://www.tpc.org/tpch/, June 2015.

[29] Cloudstack. https://cloudstack.apache.org/, March 2016.

[30] Docker Swarm. https://www.docker.com/products/docker-swarm, March 2016.

[31] Hadoop Recovery. https://twiki.grid.iu.edu/bin/view/Storage/HadoopRecovery, March 2016.

[32] Kubernetes. https://kubernetes.io, June 2016.

[33] Lxd. https://linuxcontainers.org/lxd/, January 2016.

[34] Mpich: High performance portable mpi. https://www.open-mpi.org/, 2016.

[35] Openmpi checkpointing. https://www.open-mpi.org/faq/?category=ft, 2016.

[36] Risk-return trade-off. http://cvxopt.org/examples/book/portfolio.html, 2016.

[37] VMware ESX hypervisor. https://www.vmware.com/products/vsphere-hypervisor, March 2016.

[38] VMware vCenter. https://www.vmware.com/products/vcenter-server, March 2016.

[39] Windows containers. `https://msdn.microsoft.com/virtualization/windowscontainers/containers_welcome`, May 2016.

[40] Azure low priority batch vms. `https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms`, June 2017.

[41] Ec2 spot instances pricing. `https://aws.amazon.com/ec2/spot/pricing/`, January 2017.

[42] Ibm j9 java virtual machine. `https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/user/java_jvm.html`, 2017.

[43] Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, Kudlur, Manjunath, Levenberg, Josh, Monga, Rajat, Moore, Sherry, Murray, Derek G., Steiner, Benoit, Tucker, Paul, Vasudevan, Vijay, Warden, Pete, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 265–283.

[44] Agmon Ben-Yehuda, Orna, Ben-Yehuda, Muli, Schuster, Assaf, and Tsafrir, Dan. The rise of raas: The resource-as-a-service cloud. *Communications of the ACM 57*, 7 (July 2014), 76–84.

[45] Agmon Ben-Yehuda, Orna, Posener, Eyal, Ben-Yehuda, Muli, Schuster, Assaf, and Mu'alem, Ahuva. Ginseng: Market-driven memory allocation. In *VEE* (2014), ACM.

[46] Ali-Eldin, Ahmed, Tordsson, Johan, and Elmroth, Erik. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS),* (2012), IEEE, pp. 204–212.

[47] Alipourfard, Omid, Liu, Hongqiang Harry, Chen, Jianshu, Venkataraman, Shivaram, Yu, Minlan, and Zhang, Ming. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI* (2017), USENIX.

[48] Amit, Nadav, Tsafrir, Dan, and Schuster, Assaf. Vswapper: A memory swapper for virtualized environments. *VEE* (2014).

[49] Anderson, David P. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on* (2004), IEEE, pp. 4–10.

[50] Armbrust, Michael, Das, Tathagata, Davidson, Aaron, Ghodsi, Ali, Or, Andrew, Rosen, Josh, Stoica, Ion, Wendell, Patrick, Xin, Reynold, and Zaharia, Matei. Scaling Spark in the real world: performance and usability. *VLDB 8*, 12 (2015), 1840–1843.

[51] Armbrust, Michael, Xin, Reynold S., Lian, Cheng, Huai, Yin, Liu, Davies, Bradley, Joseph K., Meng, Xiangrui, Kaftan, Tomer, Franklin, Michael J., Ghodsi, Ali, and Zaharia, Matei. Spark SQL: Relational Data Processing in Spark. In *SIGMOD* (2015).

[52] AWS Case Study: Netflix. `http://aws.amazon.com/solutions/case-studies/netflix`.

[53] Bailey, David H, Barszcz, Eric, Barton, John T, Browning, David S, Carter, Robert L, Dagum, Leonardo, Fatoohi, Rod A, Frederickson, Paul O, Lasinski, Thomas A, Schreiber, Rob S, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications 5*, 3 (1991), 63–73.

[54] Banga, G., Druschel, P., and Mogul, J. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI* (February 1999).

[55] Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauer, Rolf, Pratt, Ian, and Warfield, Andrew. Xen and the art of virtualization. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 164–177.

[56] Barr, J. New - EC2 Spot Instance Termination Notices. `https://aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/`, January 6th 2015.

[57] Barr, Jeff. Amazon ec2 update âĂŞ streamlined access to spot capacity, smooth price changes, instance hibernation. `https://aws.amazon.com/blogs/aws/amazon-ec2-update-streamlined-access-to-spot-capacity-smooth-price-changes-instance-hibernation/`, November 2017.

[58] Beck, John, Comay, David, Ozgur, L, Price, Daniel, Andy, T, Andrew, G, and Blaise, S. Virtualization and Namespace Isolation in the Solaris Operating System (psarc/2002/174).

[59] Ben-Yehuda, Muli, Agmon Ben-Yehuda, Orna, and Tsafrir, Dan. The nom profit-maximizing operating system. In *VEE* (2016), ACM.

[60] Ben-Yehuda, Muli, Day, Michael D., Dubitzky, Zvi, Factor, Michael, Har'El, Nadav, Gordon, Abel, Liguori, Anthony, Wasserman, Orit, and Yassour, Ben-Ami. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, USENIX, pp. 423–436.

[61] Ben-Yehuda, O.A., Ben-Yehuda, M., Schuster, A., and Tsafrir, D. Deconstructing Amazon EC2 Spot Instance Pricing. In *CloudCom* (November 2011).

[62] Ben-Yehuda, O.A., Ben-Yehuda, M., Schuster, A., and Tsafrir, D. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM TEC 1*, 3 (September 2013).

[63] Bernstein, D., Ludvigson, E., Sankar, K., Diamond, S., and Morrow, M. Blueprint for the Intercloud - Protocols and Formats for Cloud Computing Interoperability. In *ICIW* (2009).

[64] Bestavros, A., and Krieger, O. Toward an Open Cloud Marketplace: Vision and First Steps. *IEEE Internet Computing 18*, 1 (January/February 2014).

[65] Binnig, Carsten, Salama, Abdallah, Zamanian, Erfan, El-Hindi, Muhammad, Feil, Sebastian, and Ziegler, Tobias. Spotgres-Parallel Data Analytics on Spot Instances. In *ICDEW* (2015).

[66] Bobroff, Norman, Westerink, Peter, and Fong, Liana. Active control of memory for java virtual machines and applications. In *ICAC* (2014), pp. 97–103.

[67] Boutin, Eric, Ekanayake, Jaliya, Lin, Wei, Shi, Bing, Zhou, Jingren, Qian, Zhengping, Wu, Ming, and Zhou, Lidong. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *OSDI* (2014).

[68] Boyd, Stephen, and Vandenberghe, Lieven. *Convex Optimization*. Cambridge University Press, 2004.

[69] Brealey, Richard A, Myers, Stewart C, Allen, Franklin, and Mohanty, Pitabas. *Principles of corporate finance*. Tata McGraw-Hill Education, 2012.

[70] Cameron, Callum, Singer, Jeremy, and Vengerov, David. The judgment of forseti: Economic utility for dynamic heap sizing of multiple runtimes. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 143–156.

[71] Carvalho, Marcus, Cirne, Walfredo, Brasileiro, Francisco, and Wilkes, John. Long-term slos for reclaimed cloud computing resources. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–13.

[72] Chandy, K Mani, and Lamport, Leslie. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS) 3*, 1 (1985).

[73] Chen, Junliang, Wang, Chen, Zhou, Bing Bing, Sun, Lei, Lee, Young Choon, and Zomaya, Albert Y. Tradeoffs between profit and customer satisfaction for service provisioning in the cloud. In *HPDC* (2011), ACM, pp. 229–238.

[74] Chiang, Jui-Hao, Li, Han-Lin, and Chiueh, Tzi-cker. Working set-based physical memory ballooning. In *ICAC* (2013), pp. 95–99.

[75] Chohan, N., Castillo, C., Spreitzer, M., Steinder, M., Tantawi, A., and Krintz, C. See Spot Run: Using Spot Instances for MapReduce Workflows. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)* (June 2010), USENIX Association.

[76] Ciavotta, Michele, Gianniti, Eugenio, and Ardagna, Danilo. D-space4cloud: a design tool for big data applications. In *Algorithms and Architectures for Parallel Processing*. Springer, 2016, pp. 614–629.

[77] Clark, Christopher, Fraser, Keir, Hand, Steven, Hansen, Jacob Gorm, Jul, Eric, Limpach, Christian, Pratt, Ian, and Warfield, Andrew. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (2005), NSDI'05, USENIX Association, pp. 273–286.

[78] Clark, J. Amazon Cloud Goes Down in Northern Virginia. The Register, September 13th 2013.

[79] Cooper, Brian F, Silberstein, Adam, Tam, Erwin, Ramakrishnan, Raghu, and Sears, Russell. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[80] Cover, Thomas M. Universal portfolios. In *The Kelly Capital Growth Investment Criterion: Theory and Practice*. World Scientific, 2011, pp. 181–209.

[81] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2008)* (April 2008), USENIX.

[82] Daly, John T. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems 22*, 3 (2006).

[83] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (December 2004).

[84] Delgado, Pamela, Dinu, Florin, Kermarrec, Anne-Marie, and Zwaenepoel, Willy. Hawk: Hybrid datacenter scheduling. In *USENIX ATC* (2015).

[85] Delimitrou, Christina, and Kozyrakis, Christos. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 127–144.

[86] Delimitrou, Christina, and Kozyrakis, Christos. Hcloud: Resource-efficient provisioning in shared cloud systems. In *ASPLOS* (2016).

[87] Delimitrou, Christina, and Kozyrakis, Christos. Bolt: I know what you did last summer... in the cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 599–613.

[88] Ding, Xiaoning, Gibbons, Phillip B, Kozuch, Michael A, and Shan, Jianchen. Gleaner: mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 73–84.

[89] Dinu, Florin, and Ng, TS. Understanding the Effects and Implications of Compute Node Related Failures in Hadoop. In *HPDC* (June 2012).

[90] DRBD. DRBD: Software Development for High Availability Clusters. http://www.drbd.org/, September 2012.

[91] Dubois, Daniel J, and Casale, Giuliano. Optispot: minimizing application deployment cost using spot cloud resources. *Cluster Computing* (2016), 1–17.

[92] Elton, Edwin J, Gruber, Martin J, Brown, Stephen J, and Goetzmann, William N. *Modern portfolio theory and investment analysis.* John Wiley & Sons, 2009.

[93] Engler, Dawson R, Kaashoek, M Frans, et al. Exokernel: An operating system architecture for application-level resource management. In *SOSP* (1995), ACM.

[94] Fabozzi, Frank J, Gupta, Francis, and Markowitz, Harry M. The legacy of modern portfolio theory. *The Journal of Investing 11*, 3 (2002), 7–22.

[95] Faghri, Faraz, Bazarbayev, Sobir, Overholt, Mark, Farivar, Reza, Campbell, Roy H, and Sanders, William H. Failure Scenario as a Service (FSaaS) for Hadoop Clusters. In *Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management* (2012).

[96] Fang, Lu, Nguyen, Khanh, Xu, Guoqing, Demsky, Brian, and Lu, Shan. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 394–409.

[97] Farley, Benjamin, Juels, Ari, Varadarajan, Venkatanathan, Ristenpart, Thomas, Bowers, Kevin D, and Swift, Michael M. More for your money: exploiting performance heterogeneity in public clouds. In *Symposium on Cloud Computing* (2012), ACM.

[98] Forbes. With the public clouds of amazon, microsoft and google, big data is the proverbial big deal. `https://web.archive.org/web/20180521163820/https://www.forbes.com/sites/johnsonpierr/2017/06/15/with-the-public-clouds-of-amazon-microsoft-and-google-big-data-is-the-proverbial-big-deal/`.

[99] Ghodsi, Ali, Zaharia, Matei, Hindman, Benjamin, Konwinski, Andy, Shenker, Scott, and Stoica, Ion. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI* (2011).

[100] Gong, Zhenhuan, Gu, Xiaohui, and Wilkes, John. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management* (2010), IEEE.

[101] Grandl, Robert, Ananthanarayanan, Ganesh, Kandula, Srikanth, Rao, Sriram, and Akella, Aditya. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 455–466.

[102] Grit, Laura, Irwin, David, Marupadi, Varun, Shivam, Piyush, Yumerefendi, Aydan, Chase, Jeff, and Albrecht, Jeannie. Harnessing virtual machine resource control for job management. In *Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)* (2007), Citeseer, p. 22.

[103] Gulati, Ajay, Holler, Anne, Ji, Minwen, Shanmuganathan, Ganesha, Waldspurger, Carl, and Zhu, Xiaoyun. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal 1*, 1 (2012), 45–64.

[104] Gupta, Vishal, Lee, Min, and Schwan, Karsten. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2015), VEE '15, ACM, pp. 79–92.

[105] Hamilton, James. Overall data center costs. `http://https://perspectives.mvdirona.com/2010/09/overall-data-center-costs/`.

[106] Harlap, Aaron, Chung, Andrew, Tumanov, Alexey, Ganger, Gregory R, and Gibbons, Phillip B. Tributary: spot-dancing for elastic services with latency slos.

[107] Harlap, Aaron, Tumanov, Alexey, Chung, Andrew, Ganger, Greg, and Gibbons, Phil. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *EuroSys* (2017), ACM.

[108] Harlap, Aaron, Tumanov, Alexey, Chung, Andrew, Ganger, Gregory R., and Gibbons, Phillip B. Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 589–604.

[109] He, Ligang, Zou, Deqing, Zhang, Zhang, Chen, Chao, Jin, Hai, and Jarvis, Stephen A. Developing resource consolidation frameworks for moldable virtual machines in clouds. *Future Generation Computer Systems 32* (2014), 69–81.

[110] He, Xin, Shenoy, Prashant, Sitaraman, Ramesh, and Irwin, David. Cutting the cost of hosting online services using cloud spot markets. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (2015), ACM, pp. 207–218.

[111] He, Xin, Shenoy, Prashant, Sitaraman, Ramesh, and Irwin, David. Cutting the cost of hosting online services using cloud spot markets. In *High-Performance Parallel and Distributed Computing* (2015), ACM.

[112] Higginbotham, S. Bidding Strategies? Arbitrage? AWS Spot Market is where Computing and Finance Meet. Gigaom, October 8th 2013.

[113] Hindman, Benjamin, Konwinski, Andy, Zaharia, Matei, Ghodsi, Ali, Joseph, Anthony D, Katz, Randy H, Shenker, Scott, and Stoica, Ion. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011).

[114] Hines, Michael R., and Gopalan, Kartik. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2009), VEE '09, ACM, pp. 51–60.

[115] Hines, Michael R, Gordon, Abel, Silva, Marcio, Da Silva, Dilma, Ryu, Kyung, and Ben-Yehuda, Muli. Applications know best: Performance-driven memory overcommit with ginkgo. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (2011), IEEE, pp. 130–137.

[116] Huberman, Bernardo A, Lukose, Rajan M, and Hogg, Tad. An economics approach to hard computational problems. *Science 275*, 5296 (1997), 51–54.

[117] Iorgulescu, Calin, Dinu, Florin, Raza, Aunn, Hassan, Wajih Ul, and Zwaenepoel, Willy. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 97–109.

[118] Jain, Navendu, Menache, Ishai, and Shamir, Ohad. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, USENIX Association.

[119] Javadi, B., Thulasiram, R., and Buyya, R. Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In *UCC* (December 2011).

[120] Joel H Schopp, Keir Fraser, and Silbermann, Martine J. Resizing memory with balloons and hotplug. 313–319.

[121] Kalyvianaki, Evangelia, Charalambous, Themistoklis, and Hand, Steven. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC* (2009), ACM.

[122] Kamp, Poul-Henning, and Watson, Robert NM. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International SANE Conference* (2000), vol. 43, p. 116.

[123] Karanasos, Konstantinos, Rao, Sriram, Curino, Carlo, Douglas, Chris, Chaliparambil, Kishore, Fumarola, Giovanni Matteo, Heddaya, Solom, Ramakrishnan, Raghu, and Sakalanaga, Sarvesh. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX ATC* (2015).

[124] Kaulakienė, Dalia, Thomsen, Christian, Pedersen, Torben Bach, Çetintemel, Ugur, and Kraska, Tim. Spotadapt: Spot-aware (re-)deployment of analytical processing tasks on amazon ec2. In *Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP* (New York, NY, USA, 2015), DOLAP '15, ACM, pp. 59–68.

[125] Khatua, S., and Mukherjee, N. Application-centric Resource Provisioning for Amazon EC2 Spot Instances. In *EuroPar* (August 2013).

[126] Kivity, Avi, Kamay, Yaniv, Laor, Dor, Lublin, Uri, and Liguori, Anthony. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[127] Kivity, Avi, Kamay, Yaniv, Laor, Dor, Lublin, Uri, and Liguori, Anthony. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, pp. 225–230.

[128] Klein, Cristian, Maggio, Martina, Årzén, Karl-Erik, and Hernández-Rodriguez, Francisco. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 700–711.

[129] Koomey, J., Brill K. Turner P. Stanley J., and Taylor, B. A simple model for determining true total cost of ownership for data centers. In *Uptime Institute White Paper* (2007).

[130] Kraska, Tim, Dadashov, Elkhan, and Binnig, Carsten. Spotlytics: How to use cloud market places for analytics? In *Datenbanksysteme fÃijr Business, Technologie und Web (BTW 2017)* (2017), Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald SchÃűning, Melanie Herschel, Jens Teubner, Theo HÃďrder, Oliver Kopp, and Matthias Wieland, Eds., Gesellschaft fÃijr Informatik, Bonn, pp. 361–380.

[131] Kumar, Umesh, and Kumar, Jitendar. A comprehensive review of straggler handling algorithms for mapreduce framework. *International Journal of Grid and Distributed Computing 7*, 4 (2014), 139–148.

[132] Lagar-Cavilla, Horacio Andrés, Whitney, Joseph Andrew, Scannell, Adin Matthew, Patchin, Philip, Rumble, Stephen M., de Lara, Eyal, Brudno, Michael, and Satyanarayanan, Mahadev. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems* (2009), EuroSys '09, ACM, pp. 1–12.

[133] Liu, C., and Mao, Y. Inception: Towards a Nested Cloud Architecture. In *HotCloud* (June 2013).

[134] Liu, Haikun, and He, Bingsheng. Reciprocal resource fairness: Towards cooperative multiple-resource fair sharing in iaas clouds. In *Proceedings of the*

International Conference for High Performance Computing, Networking, Storage and Analysis (2014), IEEE Press, pp. 970–981. Weighted DRF with resource trading among applications.

[135] Liu, Haikun, Jin, Hai, Liao, Xiaofei, Deng, Wei, He, Bingsheng, and Xu, Chengzhong. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on Parallel and Distributed Systems 26*, 5 (2015), 1350–1363.

[136] Liu, Huan. Cutting mapreduce cost with spot market. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing* (2011), HotCloud'11, USENIX Association, pp. 6–6.

[137] Liu, Z., Wierman, A., Chen, Y., Razon, B., and Chen, N. Data Center Demand Response: Avoiding the Coincident Peak via Workload Shifting and Local Generation.

[138] Lorido-Botran, Tania, Miguel-Alonso, Jose, and Lozano, Jose A. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing 12*, 4 (2014), 559–592.

[139] Lu, C., Ye, K., Xu, G., Xu, C. Z., and Bai, T. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)* (Dec 2017), pp. 2884–2892.

[140] Mace, Jonathan, Bodik, Peter, Fonseca, Rodrigo, and Musuvathi, Madanlal. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI 15* (2015).

[141] Mao, Ming, and Humphrey, Marty. A Performance Study on VM Startup Time in the Cloud. In *CLOUD* (June 2012).

[142] Marathe, Aniruddha, Harris, Rachel, Lowenthal, David, De Supinski, Bronis R, Rountree, Barry, and Schulz, Martin. Exploiting redundancy for cost-effective, time-constrained execution of hpc applications on amazon ec2. In *HPDC* (2014), ACM.

[143] Markowitz, Harry. Portfolio selection. *The journal of finance 7*, 1 (1952), 77–91.

[144] Mattess, M., Vecchiola, C., and Buyya, R. Managing Peak Loads by Leasing Cloud Infrastructure Services from a Spot Market. In *HPCC* (September 2010).

[145] Mazzucco, Michele, and Dumas, Marlon. Achieving performance and availability guarantees with spot instances. In *High Performance Computing and Communications (HPCC)* (2011), IEEE.

[146] Meisner, D., Sadler, C., Barroso, L., Weber, W., and Wenisch, T. Power Management for Online Data-Intensive Services. In *ISCA* (June 2011).

[147] Meng, Xiangrui, Bradley, Joseph, Yavuz, Burak, Sparks, Evan, Venkataraman, Shivaram, Liu, Davies, Freeman, Jeremy, Tsai, DB, Amde, Manish, Owen, Sean, et al. MLlib: Machine Learning in Apache Spark. *arXiv preprint arXiv:1505.06807* (2015).

[148] Meucci, Attilio. *Risk and Asset Allocation.* Springer Finance, 2005.

[149] Mihailescu, Marian, and Teo, Yong Meng. The Impact of User Rationality in Federated Clouds. In *CCGrid* (2012).

[150] Mills, Kevin, Filliben, James, and Dabrowski, Christopher. Comparing vm-placement algorithms for on-demand clouds. In *CLOUDCOM* (2011), IEEE.

[151] Mishra, Debadatta, Kulkarni, Purushottam, et al. Doubledecker: a cooperative disk caching framework for derivative clouds. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (2017), ACM, pp. 235–247.

[152] Mishra, Mayank, and Sahoo, Anirudha. On theory of vm placement: Anomalies in existing methodologies and their mitigation using a novel vector based approach. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on* (2011), IEEE, pp. 275–282.

[153] Misra, Pulkit A, Goiri, Íñigo, Kace, Jason, and Bianchini, Ricardo. Scaling distributed file systems in resource-harvesting datacenters. In *2017 USENIX Annual Technical Conference* (2017), USENIX Association, pp. 799–811.

[154] Murray, Derek G., McSherry, Frank, Isaacs, Rebecca, Isard, Michael, Barham, Paul, and Abadi, Martín. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[155] Nguyen, Hiep, Shen, Zhiming, Gu, Xiaohui, Subbiah, Sethuraman, and Wilkes, John. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)* (2013), pp. 69–82.

[156] Nitu, Vlad, Teabe, Boris, Fopa, Leon, Tchana, Alain, and Hagimont, Daniel. Stopgap: elastic vms to enhance server consolidation. *Software: Practice and Experience 47*, 11 (2017), 1501–1519.

[157] Novakovic, Dejan, Vasic, Nedeljko, Novakovic, Stanko, Kostic, Dejan, and Bianchini, Ricardo. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), no. EPFL-CONF-185984.

[158] Ouyang, Jiannan, and Lange, John R. Preemptable ticket spinlocks: improving consolidated performance in the cloud. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 191–200.

[159] Ouyang, Xue, Irwin, David, and Shenoy, Prashant. Spotlight: An information service for the cloud. In *IEEE International Conference on Distributed Computing Systems (ICDCS)* (2016).

[160] Padala, Pradeep, Shin, Kang G, Zhu, Xiaoyun, Uysal, Mustafa, Wang, Zhikui, Singhal, Sharad, Merchant, Arif, and Salem, Kenneth. Adaptive control of virtualized resources in utility computing environments. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 289–302.

[161] Panigrahy, Rina, Talwar, Kunal, Uyeda, Lincoln, and Wieder, Udi. Heuristics for vector bin packing. *research. microsoft. com* (2011).

[162] Pary, Roshni. New amazon ec2 spot pricing model: Simplified purchasing without bidding and fewer interruptions. `https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/`, March 2018.

[163] Pegus II, Patrick, Varghese, Benoy, Guo, Tian, Irwin, David, Shenoy, Prashant, Mahanti, Anirban, Culbert, James, Goodhue, John, and Hill, Chris. Analyzing the efficiency of a green university data center. In *International Conference on Performance Engineering* (2016), ACM.

[164] Plummer, D. Cloud Services Brokerage: A Must-Have for Most Organizations. Forbes, March 22nd 2012.

[165] Prakash, Chandra, Prashanth, Prashanth, Bellur, Umesh, and Kulkarni, Purushottam. Deterministic container resource management in derivative clouds. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on* (2018), IEEE, pp. 79–89.

[166] Pundir, Mayank, Leslie, Luke M., Gupta, Indranil, and Campbell, Roy H. Zorro: Zero-cost Reactive Failure Recovery in Distributed Graph Processing. In *SOCC* (August 2015).

[167] Qu, Chenhao, Calheiros, Rodrigo N, and Buyya, Rajkumar. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications 65* (2016), 167–180.

[168] Salama, Abdallah, Binnig, Carsten, Kraska, Tim, and Zamanian, Erfan. Cost-based Fault-tolerance for Parallel Data Processing. In *SIGMOD* (2015).

[169] Salomie, Tudor-Ioan, Alonso, Gustavo, Roscoe, Timothy, and Elphinstone, Kevin. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 337–350.

[170] Satchell, Stephen, and Scowcroft, Alan. A demystification of the black–litterman model: Managing quantitative and traditional portfolio construction. *Journal of Asset Management 1*, 2 (2000), 138–150.

[171] Satopaa, Ville, Albrecht, Jeannie, Irwin, David, and Raghavan, Barath. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on* (2011), IEEE, pp. 166–171.

[172] Schwarzkopf, Malte, Konwinski, Andy, Abd-El-Malek, Michael, and Wilkes, John. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys* (2013), ACM.

[173] Sedaghat, Mina, Wadbro, Eddie, Wilkes, John, De Luna, Sara, Seleznjev, Oleg, and Elmroth, Erik. Diehard: reliable scheduling to survive correlated failures in cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid)* (2016), IEEE/ACM, pp. 52–59.

[174] Shahrad, Mohammad, Klein, Cristian, Zheng, Liang, Chiang, Mung, Elmroth, Erik, and Wentzlaf, David. Incentivizing self-capping to increase cloud utilization. In *ACM Symposium on Cloud Computing 2017 (SoCC'17)* (2017), Association for Computing Machinery (ACM).

[175] Sharma, Prateek, Chaufournier, Lucas, Shenoy, Prashant, and Tay, Y. C. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference* (2016), ACM, pp. 1:1–1:13.

[176] Sharma, Prateek, Guo, Tian, He, Xin, Irwin, David, and Shenoy, Prashant. Flint: batch-interactive data-intensive processing on transient servers. In *EuroSys* (2016), ACM.

[177] Sharma, Prateek, II, Patrick Pegus, Irwin, David, Shenoy, Prashant, Goodhue, John, and Culbert, James. Design and Operational Analysis of a Green Data Center. *IEEE Internet Computing. Special Issue on Energy-Efficient Data Centers 21*, 4 (2017), 16–24.

[178] Sharma, Prateek, Irwin, David, and Shenoy, Prashant. How not to bid the cloud. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (June 2016), USENIX.

[179] Sharma, Prateek, Irwin, David, and Shenoy, Prashant. How Not to Bid the Cloud. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (June 2016), USENIX.

[180] Sharma, Prateek, Irwin, David, and Shenoy, Prashant. Portfolio-driven resource management for transient cloud servers. In *Proceedings of ACM Measurement and Analysis of Computer Systems* (June 2017), vol. 1, p. 23.

[181] Sharma, Prateek, and Kulkarni, Purushottam. Singleton: system-wide page deduplication in virtual environments. In *HPDC* (2012), ACM.

[182] Sharma, Prateek, Lee, Stephen, Guo, Tian, Irwin, David, and Shenoy, Prashant. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *EuroSys* (April 2015).

[183] Sharma, Prateek, Lee, Stephen, Guo, Tian, Irwin, David, and Shenoy, Prashant. Managing Risk in a Derivative IaaS Cloud. *IEEE Transactions on Parallel and Distributed Systems (To Appear)* (2017).

[184] Shastri, Supreeth, and Irwin, David. Hotspot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 493–505.

[185] Shastri, Supreeth, and Irwin, David. Towards index-based global trading in cloud spot markets. *HotCloud, June* (2017).

[186] Shen, Zhiming, Jia, Qin, Sela, Gur-Eyal, Rainero, Ben, Song, Weijia, van Renesse, Robbert, and Weatherspoon, Hakim. Follow the sun through the clouds: Application migration for geographically shifting workloads. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 141–154.

[187] Shen, Zhiming, Subbiah, Sethuraman, Gu, Xiaohui, and Wilkes, John. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011).

[188] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *MSST* (May 2010).

[189] Singh, R., Irwin, D., Shenoy, P., and Ramakrishnan, K.K. Yank: Enabling Green Data Centers to Pull the Plug. In *NSDI* (April 2013).

[190] Singh, R., Sharma, P., Irwin, D., Shenoy, P., and Ramakrishnan, K.K. Here Today, Gone Tomorrow: Exploiting Transient Servers in Data Centers. *IEEE Internet Computing 18*, 4 (July/August 2014).

[191] Singh, Rahul, Irwin, David, Shenoy, Prashant, and Ramakrishnan, K.K. Yank: Enabling green data centers to pull the plug. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 143–155.

[192] Singh, Rahul, Sharma, Prateek, Irwin, David, Shenoy, Prashant, and Ramakrishnan, KK. Here today, gone tomorrow: Exploiting transient servers in datacenters. *IEEE Internet Computing 18*, 4 (2014), 22–29.

[193] Siqi Shen, Kefeng Deng, Alexandru Iosup, and Epema, Dick. Scheduling jobs in the cloud using on-demand and reserved instances. In *EuroPar* (2013).

[194] Song, Y., Zafer, M., and Lee, K. Optimal Bidding in Spot Instance Market. In *Infocom* (March 2012).

[195] Stockhammer, Thomas. Dynamic adaptive streaming over http–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems* (2011), ACM, pp. 133–144.

[196] Stoltz, Gilles, and Lugosi, Gábor. Internal regret in on-line portfolio selection. *Machine Learning 59*, 1 (May 2005), 125–159.

[197] Subramanya, S., Guo, T., Sharma, P., Irwin, D., and Shenoy, P. SpotOn: A Batch Computing Service for the Spot Market. In *SOCC* (August 2015).

[198] Tang, S., Yuan, J., and Li, X. Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In *CLOUD* (June 2012).

[199] Tomas, Luis, and Tordsson, Johan. An autonomic approach to risk-aware data center overbooking. In *Transactions on Cloud Computing* (2014), IEEE.

[200] Vavilapalli, Vinod Kumar, Murthy, Arun C, Douglas, Chris, Agarwal, Sharad, Konar, Mahadev, Evans, Robert, Graves, Thomas, Lowe, Jason, Shah, Hitesh, Seth, Siddharth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Symposium on Cloud Computing* (2013), ACM.

[201] Verma, Abhishek, Pedrosa, Luis, Korupolu, Madhukar, Oppenheimer, David, Tune, Eric, and Wilkes, John. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)* (2015), ACM.

[202] Vintila, Alexandra, Oprescu, Ana-Maria, and Kielmann, Thilo. Fast (re-)configuration of mixed on-demand and spot instance pools for high-throughput computing. In *Workshop on Optimization techniques for resources management in clouds* (2013), ACM, pp. 25–32.

[203] Voorsluys, W., and Buyya, R. Reliable Provisioning of Spot Instances for Compute-Intensive Applications. In *AINA* (2012).

[204] Waldspurger, Carl A. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[205] Waldspurger, Carl A. Memory resource management in vmware esx server. *OSDI* (2002).

[206] Waldspurger, Carl A., Hogg, Tad, Huberman, Bernardo A., Kephart, Jeffrey O., and Stornetta, W. Scott. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering 18*, 2 (1992), 103–117.

[207] Wang, Cheng, Liang, Qianlin, and Urgaonkar, Bhuvan. An empirical analysis of amazon ec2 spot instance features affecting cost-effective resource procurement. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 3*, 2 (2018), 6.

[208] Wang, Cheng, Urgaonkar, Bhuvan, Gupta, Aayush, Kesidis, George, and Liang, Qianlin. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 620–634.

[209] Wang, Cheng, Urgaonkar, Bhuvan, Gupta, Aayush, Kesidis, George, and Liang, Qianlin. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. 620–634.

[210] Wee, S. Debunking Real-Time Pricing in Cloud Computing. In *CCGrid* (May 2011).

[211] Wen, Jiawei, Lu, Lei, Casale, Giuliano, and Smirni, Evgenia. Less can be more: Micro-managing vms in amazon ec2. In *International Conference on Cloud Computing* (2015), IEEE.

[212] Wieder, Alexander, Bhatotia, Parmod, Post, Ansley, and Rodrigues, Rodrigo. Orchestrating the deployment of computations in the cloud with conductor. In *NSDI 12* (2012).

[213] Williams, D., Jamjoom, H., and Weatherspoon, H. Plug into the Supercloud. *IEEE Internet Computing 17*, 2 (2013).

[214] Williams, Dan, Jamjoom, Hani, and Weatherspoon, Hakim. The xen-blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, ACM, pp. 113–126.

[215] Wolski, Rich, Brevik, John, Chard, Ryan, and Chard, Kyle. Probabilistic guarantees of execution duration for amazon spot instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), ACM, p. 18.

[216] Wood, Timothy, Shenoy, Prashant, Venkataramani, Arun, and Yousif, Mazin. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks 53*, 17 (2009), 2923–2938.

[217] Wood, Timothy, Shenoy, Prashant J, Venkataramani, Arun, and Yousif, Mazin S. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI* (2007), vol. 7, pp. 17–17.

[218] Xin, Reynold S, Gonzalez, Joseph E, Franklin, Michael J, and Stoica, Ion. Graphx: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM.

[219] Xu, H., and Li, B. A Study of Pricing for Cloud Resources. *Performance Evaluation Review 40*, 4 (March 2013).

[220] Xu, Z., Stewart, C., Deng, N., and Wang, X. Blending on-demand and spot instances to lower costs for in-memory storage. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications* (April 2016), pp. 1–9.

[221] Yan, Ying, Gao, Yanjie, Chen, Yang, Guo, Zhongxin, Chen, Bole, and Moscibroda, Thomas. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 484–496.

[222] Yang, Ting, Berger, Emery D, Kaplan, Scott F, and Moss, J Eliot B. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 103–116.

[223] Yang, Youngseok, Kim, Geon-Woo, Song, Won Wook, Lee, Yunseong, Chung, Andrew, Qian, Zhengping, Cho, Brian, and Chun, Byung-Gon. Pado: A data processing engine for harnessing transient resources in datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)* (2017), ACM.

[224] Yang, Youngseok, Kim, Geon-Woo, Song, Won Wook, Lee, Yunseong, Chung, Andrew, Qian, Zhengping, Cho, Brian, and Chun, Byung-Gon. Pado: A data processing engine for harnessing transient resources in datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 575–588.

[225] Yi, S., Kondo, D., and Andrzejak, A. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *CLOUD* (July 2010).

[226] Yi, Sangho, Kondo, Derrick, and Andrzejak, Artur. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* (2010), IEEE, pp. 236–243.

[227] Zafer, M., Song, Y., and Lee, K. Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs. In *CLOUD* (2012).

[228] Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J., Shenker, Scott, and Stoica, Ion. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[229] Zaharia, Matei, Das, Tathagata, Li, Haoyuan, Hunter, Timothy, Shenker, Scott, and Stoica, Ion. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP* (2013).

[230] Zaharia, Matei, Xin, Reynold S., Wendell, Patrick, Das, Tathagata, Armbrust, Michael, Dave, Ankur, Meng, Xiangrui, Rosen, Josh, Venkataraman, Shivaram, Franklin, Michael J., Ghodsi, Ali, Gonzalez, Joseph, Shenker, Scott, and Stoica, Ion. Apache spark: A unified engine for big data processing. vol. 59, ACM, pp. 56–65.

[231] Zaman, S., and Grosu, D. Efficient Bidding for Virtual Machine Instances in Clouds. In *CLOUD* (July 2011).

[232] Zhang, Dongli, Ehsan, Moussa, Ferdman, Michael, and Sion, Radu. Dimmer: A case for turning off dimms in clouds. In *Symposium on Cloud Computing* (2014), ACM, pp. 1–8.

[233] Zhang, F., Chen, J., Chen, H., and Zang, B. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *SOSP* (October 2011).

[234] Zhang, Q., Gürses, E., Boutaba, R., and Xiao, J. Dynamic Resource Allocation for Spot Markets in Clouds. In *Hot-ICE* (March 2011).

[235] Zhang, Qi, Liu, Ling, Ren, Jiangchun, Su, Gong, and Iyengar, Arun. iballoon: Efficient vm memory balancing as a service. In *Web Services (ICWS), 2016 IEEE International Conference on* (2016), IEEE, pp. 33–40.

[236] Zhang, Xiao, Tune, Eric, Hagmann, Robert, Jnagal, Rohit, Gokhale, Vrigo, and Wilkes, John. Cpi 2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 379–391.

[237] Zhang, Yunqi, Prekas, George, Fumarola, Giovanni Matteo, Fontoura, Marcus, Goiri, Íñigo, and Bianchini, Ricardo. History-based harvesting of spare cycles and storage in large-scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association, pp. 755–770.

[238] Zhang, Yunqi, Prekas, George, Fumarola, Giovanni Matteo, Fontoura, Marcus, Goiri, Inigo, and Bianchini, Ricardo. History-based harvesting of spare cycles and storage in large-scale datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 755–770.

[239] Zhao, Weiming, Wang, Zhenlin, and Luo, Yingwei. Dynamic memory balancing for virtual machines. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 37–47.

[240] Zheng, Liang, Joe-Wong, Carlee, Tan, Chee Wei, Chiang, Mung, and Wang, Xinyu. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 71–84.

[241] Zheng, Wei, Bianchini, Ricardo, Janakiraman, G. John, Santos, Jose Renato, and Turner, Yoshio. Justrunit: Experiment-based management of virtualized data centers. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 18–18.

[242] Zhou, Wenyu, Yang, Shoubao, Fang, Jun, Niu, Xianlong, and Song, Hu. Vmctune: A load balancing scheme for virtual machine cluster using dynamic resource allocation. In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on* (2010), IEEE, pp. 81–86.