SYSTEM SUPPORT FOR MANAGING RISK IN CLOUD COMPUTING PLATFORMS

A Dissertation Presented

by

SUPREETH SHASTRI

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

Electrical and Computer Engineering

SYSTEM SUPPORT FOR MANAGING RISK IN CLOUD COMPUTING PLATFORMS

A Dissertation Presented

by

SUPREETH SHASTRI

Approved as to style and content by:

David Irwin, Chair

Prashant Shenoy, Member

Lixin Gao, Member

Michael Zink, Member

Christopher Hollot, Department Chair
Electrical and Computer Engineering

Dedicated to

Archie, Amma, and America

ACKNOWLEDGMENTS

Much of my lifestyle and value system stems from my traditional upbringing in a middle class South Indian family. I grew up with stories celebrating Indian mythology and Indian mathematics. Long before Robert Kanigel chronicled about Ramanujan, I had already learnt that *an equation has no meaning unless it expresses a thought of God.* Everyone in my social circle amplified my educational successes and trivialized my non-intellectual failures. Through nature and nurture, I came to value education and the pursuit of knowledge above all else. I am indebted to my parents, my brother, and my extended family for shaping my younger self. Thanks to my college cohort at PESIT for picking up this mantle in my teen years. To Akku, Aru, Ash, Bindu, Boda, Juggi, KC, Manu, MoF, PJ, Pothnis, Roonie, Rush, Sree, Susan, Vikky and Vishwas: life would never have been this meaningful without you all; nor would I have the conviction to take this PhD to completion.

The American higher education system has had a formative influence on my life. While my stay at Columbia sprouted my curiosity in research, the time at UMass helped me take it to a professional level. Thanks to my alma maters, I was able to find, and then pursue my true calling. Also, without sustained funding from the U.S. National Science Foundation and the U.S. Federal Aviation Administration, this journey would have been arduous. Sincere thanks to the American public funding system.

My professors and friends at UMass have given me company through peace and panic. Prashant Shenoy has been a mentor and a role model; Mike Zink and Lixin Gao went beyond their roles as my committee members to help me; Tian Guo and Prateek Sharma were instrumental in getting my first paper across the finish line; Amr Rizk's queuing theory genius was an eye opener and working with him brought down my Erdos number to 6; Dong Chen, Abhishek Dwaraki, Divyashri Bhat and Sunil Kumar have played host to me numerous times, held me from wandering off-track, and spent numerous hours pondering about life in academia. Thank you all, and I hope our roads cross again.

I wish to express my gratitude to the ACM Symposium on Cloud Computing and USENIX Hot Topics in Cloud Computing. As a young researcher working on an unconventional topic, I was amazed by their openness to diverse ideas. Majority of my research is not only published in these two venues but are significantly better off because of it. Being accepted into this research cohort has done wonders at warding off my imposter syndrome as well as keeping my research grounded to reality.

PhD-ing as a parent turned out to be much more fun and frolic than I had imagined. Credit for that mostly goes to my 9-year old boy, Sam. We taught and learnt from each other. The time I spent with him not only made me a better dad but a better writer, thinker, and teacher. His silly jokes, snarky comments, empathy and pride towards my work have carried me through many rough waters. Sam 100, papa 0!

Finally and most importantly, the driving force and backbone of my doctoral journey have been my advisor David Irwin and my wife Archie. They believed I could do it, stuck with me through thick and thin, and poured their time and energy into making me a researcher. They supported me financially, accommodated my shortcomings, and celebrated every minor accomplishment wholeheartedly. I would not have lasted a moment without their unwavering support. This work is as much theirs as mine.

ABSTRACT

SYSTEM SUPPORT FOR MANAGING RISK IN CLOUD COMPUTING PLATFORMS

SEPTEMBER 2018

SUPREETH SHASTRI

M.S., COLUMBIA UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor David Irwin

Cloud platforms sell computing to applications for a price. However, by precisely defining and controlling the service-level characteristics of cloud servers, they expose applications to a number of implicit risks throughout the application's lifecycle. For example, user's request for a server may be denied, leading to *rejection risk*; an allocated resource may be withdrawn, resulting in *revocation risk*; an acquired cloud server's price may rise relative to others, causing *price risk*; a cloud server's performance may vary due to external factors, triggering *valuation risk*. Though these risks are implicit, the costs they bear on the applications are not.

While some risks exist in all Infrastructure-as-a-Service offerings, they are most pronounced in an emerging category called *transient cloud servers*. Since transient servers are carved out of instantaneous idle cloud capacity, they exhibit two distinct features: (i) revocations that are intentional, frequent and come with advanced warning, and (ii) prices that are low in average but vary across time and location. Thus, despite enabling inexpensive access to at-scale computing, transient cloud servers expose applications to risks, the scale of which were unseen in the past platforms. Unfortunately, the current generation

system software are not designed to handle these risks, which in turn results in inconsistent performances, unexpected failures, missed savings, and slower adoption.

In this dissertation, we elevate *risk management* to a first-class system design principle. Our goal is to identify the risks, quantify their costs, and explicitly manage them for applications deployed on cloud platforms. Towards that goal, we adapt and extend concepts from finance and economics to propose a new system design approach called *financializing cloud computing*. By treating cloud resources as investments, and by quantifying the cost of their risks, financialization enables system software to manage the risk-reward trade-offs, explicitly and autonomously.

We demonstrate the utility of our approach via four contributions: (i) mitigating revocation risk with *insurance policy*, (ii) reducing price risk through *active trading*, (iii) eliminating uncertainty risk by *index tracking*, and (iv) minimizing server's valuation risk via *asset pricing*. We conclude by observing that diversity and asymmetry in the creation and consumption of cloud compute resources is on the rise, and that financialization can be effectively employed to manage its complexity and risks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

"Risk comes from not knowing what you are doing."

Warren Buffett

1.1 The State of Cloud Computing

Cloud computing [8] is an umbrella term that refers to hardware infrastructure, system software or applications delivered as a service over the Internet. An early example of a public cloud at scale is Amazon's Elastic Cloud Compute (EC2), which started offering Infrastructure-as-a-Service (IaaS) virtual machines in 2006 [12]. IaaS cloud platforms provided numerous benefits including on-demand access, pay-as-you-go billing model, and near-infinite scalability—all of which came without any upfront capital investment. As a result, cloud computing quickly became the foundation of our information-based economy, providing large-scale compute power to nearly every segment of our society including commerce, science, communications, entertainment, finance, and healthcare. Such widespread adoption has propelled cloud computing to account for more than half of the worldwide IT spending by 2021 [27].

In order to benefit from this projected growth, major cloud providers—Amazon EC2, Microsoft Azure, and Google Compute Engine (GCE) have been rapidly expanding their datacenters. However, a growing infrastructure footprint makes it challenging to achieve high resource utilization. For example, a 2014 report from the Natural Resource Defense Council indicates the average utilization across cloud datacenters to be $\sim$40% [89]. To mitigate this problem, providers have started evolving their cloud platforms from fixed-price server rentals to full-fledged marketplaces that offer a wide variety of service contracts. As of this date, a given IaaS cloud server could be procured on Amazon EC2 under 12 different contracts, and on Google GCE under 6 different contracts. While their nomenclatures

vary, these contracts offer different combinations of pricing options, time commitments, performance guarantees, and availability constraints.

Apart from increasing the temporal and spatial multiplexing of the infrastructure, cloud contracts enable providers to offer purchasing options better tailored to the needs of a broad customer base. Cloud contracts specify the service level agreements (SLAs) and service level objectives (SLOs) that the provider is willing to offer to its customers. While SLOs simply indicate objectives that would be met with best effort, SLAs constitute guarantees that incur penalties upon violation. By precisely defining, and accordingly pricing the service-level characteristics of IaaS compute servers, the cloud providers are implicitly exposing their customers to several risks.

Customers experience risks throughout their computing lifecycle on the cloud. When users request for resources, it may be denied by the provider, leading to *Rejection Risk*. Even after resources have been acquired, they may become unavailable due to unexpected datacenter failures or intentional revocation by the provider, thus resulting in *Revocation Risk*. A procured cloud server may become less cost-effective relative to other servers over time, causing *Price Risk*. Since cloud is intrinsically a shared platform, the performance obtained by user's application may vary due to external factors, triggering *Variability Risk*. Given the complexity of cloud contracts and diversity of applications, the users may not be able to accurately value a cloud server's utility, thereby resulting in *Valuation Risk*. While not exhaustive, this list demonstrates the presence of new risks that applications are exposed to in the cloud.

1.2 Limitations of Current Systems

While an expansive set of contracts should help users in achieving a better fit for their workloads, the current generation software systems are ill-prepared to handle the associated risks. The origins of this can be traced back to times where compute servers were either individually owned or shared amongst cooperating users belonging to the same organization. In such settings, where the provider and consumers are working together, and not trying to maximize their individual utility, the need for explicit contracts on server characteristics was largely irrelevant. As a result, system software and applications evolved to treat all

servers alike expect for their hardware configurations. The notion that a compute server's characteristics do not change, at least in the lifetime of the application, has continued to influence the design of modern system software frameworks like Hadoop and Spark.

However, ignoring the realities of the underlying infrastructure exposes applications to several risks that result in inconsistent performances, unexpected failures, and volatile costs. But prior research as well as current generation software systems either oversimplify or ignore these risks. Though such practice is prevalent in applications that are not natively designed for cloud platforms, it is not exclusive to them. For example, even cloud generation cluster managers like Mesos and Kubernetes do not distinguish between majority of cloud contract types. In summary, we observe that while the cloud has evolved into an advanced marketplace, consumers lack the sophistication and toolset to manage the risks of operating there. Our work is an attempt to bridge this gap.

While cloud applications face some risk in all types of cloud contracts, none are as potent as the ones exposed in *transient cloud computing*. Transient servers originate from datacenter's unused capacity, which is idle at the given moment but could be summoned for other purposes at any time. Thus, they exhibit two distinct features: (i) revocations that are intentional, frequent and come with advanced warning, and (ii) prices that are low in average but vary across time and location. On one hand, transient servers have enabled access to cloud computing for a wide swath of applications that were hither to inhibited by the high costs of on-demand servers; but on the other hand, they expose applications to revocation-, price-, valuation-, and uncertainty-risks, the scale of which were unseen in the past. Consequently, this is resulting in missed savings, and worse, slower adoption.

1.3 Managing Risks in the Cloud

In this dissertation, we elevate risk management to a first-class system design principle. Our goal is to systematically identify the risks in cloud platforms, quantify their costs, and explicitly manage them for applications. In doing so, we observe that IaaS cloud platforms have turned into full-fledged marketplaces with contract offerings resembling those in the commodity and financial markets. Over the last 50 years, financial companies have engaged in sophisticated risk management strategies to operate effectively in their markets. We

Risks	Problems	Proposed Solution	Result
Revocation	Revocations pose a new fault-model that undermines the benefit of using transient servers	Cost-efficient insurance via *SpotOn*	91% cost-savings over on-demand
Price	Applications may pay higher price for less efficient servers in variable-priced cloud markets	Active server trading via *HotSpot*	50% cost reduction over insurance schemes
Uncertainty	Applications deployed on variable-priced cloud markets suffer from cost uncertainty	Server hosting that tracks *Cloud Indices*	Predictable costs that match the index-level
Valuation	Transient servers are difficult to price (for providers) and to value (for consumers)	Asset pricing via *Transient Guarantees*	5x increase in the overall value

Table 1.1: Thesis contributions

argue that with compute-time turning into a core investment, technology-enabled companies would need to understand and cope with the risks present in the cloud platforms.

This naturally leads us to adapt risk management techniques from finance and economics, where it has been extensively researched and practiced. However, as the rest of the thesis shows, there are significant differences between cloud servers and financial instruments such that it requires considerable modifications and enhancements to adapt ideas from these domains. Aptly, we call our approach to managing risks in cloud systems, *financializing cloud computing*. By identifying cloud risks and then devising techniques to quantify their costs, *financialization* enables system software to manage the risk-reward tradeoff autonomously. As we describe in the next section, this approach enables unmodified cloud applications to manage their risks and at times, benefit from them.

1.4 Summary of Contributions

Our central hypothesis is that we can adapt and extend techniques from economics and finance to transparently mitigate new types of risks that cloud platform expose to applications. In evaluating this hypothesis, we focus on four key cloud risks, and design new abstractions, mechanisms, and policies towards managing them. We highlight our contributions in Table 1.1, and describe them below.

1.4.1 Mitigating Revocation Risk with Insurance

Revocations pose a new fault-model that undermines the benefit of using transient cloud servers. Unlike the classical hardware failure, transient server revocations are intentional, frequent and come with advanced warning. In SpotOn [78], we investigate how to insure against revocation risk without incurring huge premiums.

We identify that the cost of insurance against transient server revocations is a function of application's footprint, transient server's market characteristics, and fault-tolerance mechanism's overhead. In order to make this cost-efficient, we (i) extend the classical fault-tolerance mechanisms of migration, checkpointing, and replication to the new fault scenario and model their overheads; and then (ii) design a greedy insurance policy that dynamically selects a combination of transient server and fault-tolerance mechanism that results in the lowest premium. We implement these in a service called SpotOn, which executes unmodified batch applications on EC2 spot servers. Evaluations on Amazon EC2 show that SpotOn is able to achieve near on-demand performance while also realizing ∼91% cost savings.

1.4.2 Reducing Price Risk through Active Trading

Transient cloud servers sold in variable priced markets, like EC2 spot markets, exhibit price variations, inversions and arbitrages. These lead to price risk, or the risk that a chosen server's price will increase relative to others. In HotSpot [70], we explore how to avoid price risk for unmodified cloud applications.

Through market analysis, we observe (i) that price risk is ∼500x more frequent than revocation risk, and (ii) that servers with high discount also tend to have low revocation risk (which is reflective of the supply-demand dynamics). With these key observations, we hypothesize that by employing active trading (i.e., hopping from the current server to a better one), a flexible application can reduce its costs without increasing its revocation risk. Then, we design a server hopping mechanism at the system level (via a self-migrating container) such that unmodified applications can utilize it. HotSpot, our prototype on Amazon EC2 demonstrates that active trading results in ∼50% savings relative to insurance based approaches.

1.4.3 Eliminating Uncertainty Risk by Index Tracking

Applications that run on variable-priced cloud servers suffer from cost uncertainty. Since the server prices are market-based, and could vary considerably (up to 10×), customers find it difficult to plan their IT expenses. In [72], we design an index-tracked cloud server to eliminate the uncertainty risk.

While prior approaches have tried to model and predict individual transient server markets, they have had limited success due to the proliferation of EC2 spot markets. We propose an alternative solution based on two key insights: (i) making price predictions at aggregate market level is more reliable than at individual server level, and (ii) knowing the benchmark for cost estimates a priori enables reactive server management systems to achieve the target cost-efficiency without sacrificing availability. Towards eliminating cost uncertainty, we introduce a market-based cloud index, and design a mechanism for index-tracking via server hopping. We implement and evaluate this system on Amazon EC2 spot markets, and demonstrate that it can reliably achieve the predicted cost-efficiency for a broad class of flexible applications.

1.4.4 Minimize Valuation Risk via Asset Pricing

For transient cloud server offerings, the providers do not reveal precise transiency information as it makes their administration challenging. However, this opacity makes it difficult for consumers to gauge their true value. In Transient Guarantees [73], we explore how to minimize the valuation risk.

By distilling transient server characteristics into three orthogonal axes of availability, volatility, and predictability, we introduce the notion of equilibrium price—i.e., the price beyond which the utility of a transient server (modulo its fault-tolerance overhead) is no better than an equivalent on-demand server. While equilibrium price is only applicable in retrospect, it helps consumers determine how their transient server fared. Interestingly, our market analysis using equilibrium price reveals that Amazon and Google transient server contracts do not maximize the server's value for either providers or consumers. To address these problems, we design a new asset-pricing abstraction called transient guarantee that offers probabilistic assurances on transiency characteristics. Through modeling and

evaluation, we show that transient guarantees not only help users in determining the value of transient servers upfront but also enable providers to increase their revenue by up to 5× without sacrificing their ability to revoke transient servers.

1.5 Dissertation Outline

The rest of the thesis is organized as follows. Chapter 2 provides the necessary background on cloud platforms, and motivates our approach of financialization. Chapter 3 describes SpotOn, a batch computing service that designs a cost-efficient insurance policy against revocation risk. Chapter 4 covers HotSpot, a platform for active server trading that mitigates the price risk. Chapter 5 details our market-based cloud index that enables index-tracking, and eliminates cost uncertainty. Then, chapter 6 introduces Transient Guarantees, an asset pricing model that minimizes the valuation risk of idle cloud capacity. Finally, we conclude and highlight the future work in chapter 7.

CHAPTER 2

BACKGROUND

> "I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone."
>
> Bjarne Stroustrup

This chapter serves as a primer on IaaS cloud server contracts, and how their composition translates into specific cloud risks. We also describe how the risks are elevated in transient cloud computing by using Amazon EC2 spot market as an illustrative example. Finally, we highlight the recent progress towards commoditizing compute-time on the cloud, and motivate our work on risk management via financialization.

2.1 IaaS Cloud Server Contracts

Early cloud server contracts bear strong resemblence to the conventional commodity contracts namely, spot and futures. Spot contracts offer commodities for immediate delivery, while futures contracts offer commodities for at a predetermined future date. These contracts are structured so that companies that rely on commodities can buy and sell them based on their expectations of workload and market prices. For example, an airline might purchase oil futures to meet its expected fuel demands over the next year. Then, as the year progresses, if the actual demand is lower or higher than expected at any point, the airline may sell its spare or buy extra as spot contracts to ensure the real-time demand is met. However, as providers began amassing expertise in operating large-scale datacenters efficiently, they evolved the cloud contracts along more specialized characteristics in order to increase their utilization levels as well as to closely match the needs of their customers. Be-

low, we highlight the prominent types of IaaS cloud contracts, using examples from Amazon EC2 and Google Compute Engine:

- **EC2 On-demand and GCE On-demand:** Most widely known and used, on-demand contracts allow users to request and relinquish cloud servers at any time. While they employ a fixed per-second pricing model, their price levels are considerably higher compared to other contract types, given their flexibility.
- **EC2 Reserved and GCE Committed Use:** These are the cloud's equivalent of futures contract, where customers sign up for 1-3 years of committed use and in return, the providers offer discounted rates and guaranteed access over that period. While well suited for long-term predictable workloads, these contracts eliminate much of the elasticity benefits of the cloud, and may increase the costs if not highly utilized.
- **EC2 Spot and GCE Preemptible:** The purpose of this contract is to increase datacenter's utilization levels by selling access to idle cloud capacity that otherwise cannot be turned off. As the instantaneous surplus capacity varies with time, the providers retain the right to revoke the offered servers with only a brief warning. While EC2 has adapted a market-based variable pricing option, GCE prices these at $\sim$30% of the on-demand equivalent.
- **EC2 Defined-duration:** Also called the Spot-blocks, these contracts are a middle ground between on-demand and reserved types. The provider guarantees 1-6 hours of access at a discounted price and will reclaim the server after that period. The billing model is same as that of EC2 spot markets i.e., market-based variable pricing. GCE does not offer an equivalent contract.
- **EC2 Burstable and GCE Shared Core:** Designed for applications that do not need consistently high-levels of CPU but may occasionally need to sprint, these contracts offer inexpensive access to shared cloud servers. The prices are significantly discounted and billed at fixed per-second granularity. The downsides include rate-limited performance as well as direct exposure to interference by other cloud tenants.
- **EC2 Dedicated and GCE Sole Tenant:** These are generic qualifiers that when applied to a contract, alter its characteristics significantly. As the name suggests, the

provider allocates a dedicated physical machine for the contract. Thus, users could expect predictable performances and arguably better privacy and security conditions. However, since this directly affects the provider's ability to multiplex user requests, the costs are significantly higher.

2.2 Implicit Risks in Cloud Contracts

In structuring the IaaS cloud server offerings as contracts governed by carefully crafted SLAs and SLOs, and pricing them accordingly, the cloud providers are implicitly exposing the customers to various risks. This is a big departure from the pre-cloud days where compute servers were primarily distinguished by their hardware capabilities, and any differences in their price, performance, and failure characteristics were abstracted from the users. Thus, we define risk as quantifiable cloud server characteristics that affect application's expected performance, behavior, and ultimately cost. We list below, significant risks exposed by the current generation cloud platforms:

- **Rejection Risk:** While the cloud promises infinite scalability in theory, the cloud providers many not be able or willing to fulfill the requested services at certain times. This may be due to limitations of the datacenter infrastructure, instantaneous oversubscription, planned maintenance, or unexpected failures. Thus, the rejection risk leaves customers unable to obtain the needed services. All contract types are subject to this risk with the exception of futures contracts—*EC2's reserved instances* and *GCE's committed use VMs*, both of which require customers to sign up for 1-3 years of service, ahead of time.
- **Revocation Risk:** Unlike rejection, the revocation risk affects the already allocated servers. This risk is most visible in the transient server contracts—*EC2's spot instances* and *GCE's preemptible VMs*, both of which grant providers the right to revoke an already allocated server after a brief warning. Though most contracts explicitly prohibit intentional revocations, they do not guarantee 100% availability. While providers pay out a penalty upon unintended service failures, the customers are not immune to revocation risks.

	Rejection	Revocation	Price	Variability	Workload	Valuation
EC2/GCE On-demand	Yes	No	No	Yes	No	No
EC2/GCE Reserved	No	No	Yes	Yes	Yes	No
EC2 Spot	Yes	Yes	Yes	Yes	No	Yes
GCE Preemptible	Yes	Yes	No	Yes	No	Yes
EC2 Defined-Duration	Yes	No	Yes	Yes	Yes	Yes
EC2/GCE Burstable	Yes	No	No	Yes	No	Yes

Table 2.1: Risks exposed by EC2 and GCE cloud contracts.

- **Price Risk:** This is the risk that an acquired cloud server becomes less cost-effective relative to others. Though this risk occurs routinely in variable-priced contracts like *EC2 spot markets*, it is not unique to them. Introduction of new generation of cloud servers, discontinuation of contracts or server families, and price reductions for certain servers are all examples that introduce price risks. While the timescales of price risks may vary, no contract type is immune to this risk.
- **Variability Risk:** Since the cloud is a shared compute platform, the risk of performance variability is intrinsic to it. While providers offer a contract type that mitigates this risk (namely, *EC2's dedicated instances* and *GCE's sole tenant VMs*), most contracts are not immune to it. However, unlike other risks, performance variability is hard to quantify and mitigate as external interferences that cause it vary across applications, locations, time, and server types.
- **Valuation Risk:** This risk manifests in user's inability to accurately value a cloud server's utility. Unspecified server characteristics, market-based pricing, performance variability, non-uniform effect of failures on applications are amongst the factors that contribute to this risk. While valuation risk's impact could be reduced by specialized cloud contracts that provide additional guarantees (like dedicated hosting and reserved pricing), it cannot be eliminated.

Table 2.1 succinctly represents the risks exposed by EC2 and GCE contracts. We note that the table offers only a coarse approximation of each contract's terms and composition as they are too complex to capture in a concise tabular format. The main takeaway is that

no type of contract is free from all the risks, and applications have to evolve to manage them explicitly. Also importantly, we observe that transient contracts are exposed to more risks than all other types. In the next section, we explore this in more detail.

2.3 Case Study: Transient Cloud Computing

The origin of transient cloud computing lies in cloud providers' attempt at repurposing the spare server capacity, which turns out to be a significant portion (up to 40%) of the cloud datacenters. Though unused, these servers could not be turned off due to variety of reasons including: (i) provisioning for peak expected capacity, (ii) internal and external fragmentation of physical machines caused by VM multiplexing, (iii) disruption-free administration and maintenance, (iv) SLAs that expect server requests to be fulfilled faster than the time it takes to cold boot physical machines, and (v) failover servers intended to maintain high availability guarantees.

Since the providers do not know in advance as to which of the unused servers would be required at what time, they have structured these offerings as transient server i.e., servers whose availability is not guaranteed. *EC2 spot instances* and *GCE preemptible VMs* are examples of this category. Because of their undesirable availability properties, transient servers are offered at highly discounted prices. For example, compared to their equivalent on-demand contracts, EC2 spot servers are discounted at 50-90% and GCE preemptible VMs are discounted at 70%. The main difference between EC2 and GCE transient offering is that the former employs a market-based variable pricing mechanism while the latter uses a fixed price.

As a result of their low pricing, this new contract type suddenly provided access to inexpensive cloud computing for a wide swath of applications that were hither to inhibited by the high costs of on-demand servers. For example, the Fermilab Scientific Computing Division employed spot servers to dynamically scale up their compute capacity by 4× during the discovery Higgs-Boson [15]. Similarly, a group of machine learning and natural language processing researchers recently set the record [16] for the largest ever high-performance cluster on the cloud by using 1.1 million vCPUs on spot servers.

Though transient servers offer a significant potential for cost savings, the magnitude of these savings is not guaranteed, and could ultimately be negative if their transiency characteristics change unexpectedly. The effects are more profound but better quantifiable in the EC2 spot markets, which adopts the following operating mechanism: users bid for spot servers at the time they need them; EC2 continually evaluates the supply-demand dynamics of their idle capacity to determine the clearing price; if a user's bid exceeds the spot price, they are allocated the server; when the spot price rises above the user's bid level, EC2 reserves the right to revoke the server with a two-minute warning [13].

EC2's global footprint is massive and complex: it operates in 16 worldwide regions each of which comprise of 2-6 availability zones, and has announced plans to add 6 new regions with 17 additional zones in the future [3]. Since EC2 sets a different dynamic spot price for each type of server in each availability zone of each region, the global spot market currently includes more than 7600 separate server "listings". Notwithstanding the global footprint, the spot prices are hard to predict even for identical servers within a region. For example, Figure 2.1 shows the price of `r3.4xlarge` Linux server in four availability zones of `US-East-1` region.

Elevated Risks in Transient Contracts. While renting out surplus capacity increases utilization for providers, and gives access to cheap computing for customers, the need to repurpose them for more important endeavors at a moment's notice affects application's performance. This worsens (i) the revocation risk since providers are reclaiming the servers, intentionally and frequently, (ii) the valuation risk since applications now have to employ fault-tolerance mechanisms to preserve their forward progress, thereby making it difficult to value the true utility of the transient server.

While variable-priced and market-based allocation schemes are effective in determining the right price and in automatically balancing the supply and demand in real-time, they result in the price volatility. In turn, this exacerbates (i) the price risk since the prices of all servers are varying in real-time, and (ii) the uncertainty risk since customers cannot plan for their IT budgets in advance. These two problems are somewhat minimized in GCE preemptible due to their fixed pricing but are not completely absent. This is because

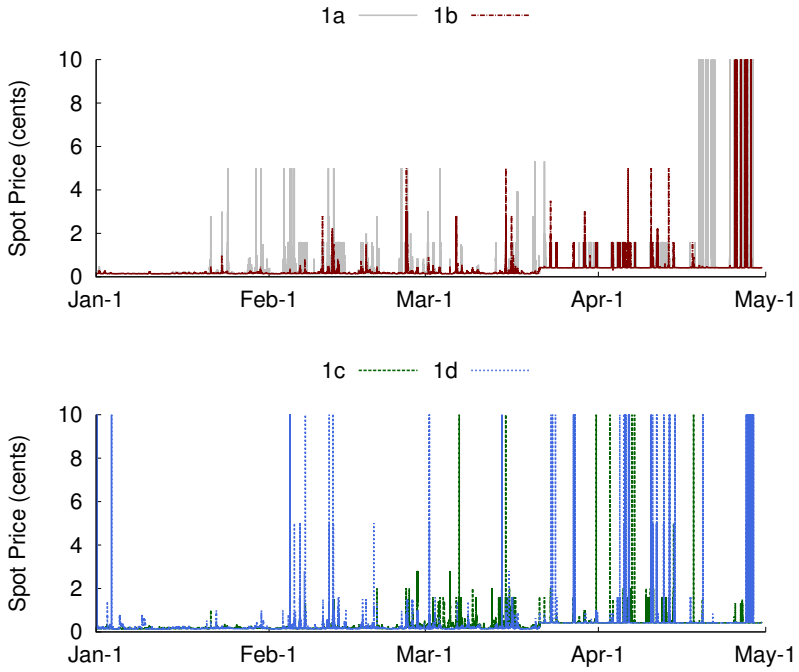


Figure 2.1: Price of a representative Linux server (r3.4xlarge) across four availability zones of the US-East-1 region.

the elevated risks of revocation and valuation affect application's performance and total incurred costs, which indirectly expose users to price- and uncertainty-risks.

2.4 Motivation

2.4.1 Towards Commoditized Compute

While our work financialization is motivated by cloud's evolution into a marketplace, the mechanisms and policies that benefit from this approach require compute-time to be a fungible commodity. Over the last 5-10 years, we are witnessing a confluence of technological advancements in virtualization and networking, especially in the context of datacenters. For example, resource containers [5, 52, 24, 11, 56] and nested virtualization [20, 91] are eliminating many of the implicit dependencies that bind applications to an underlying machine. At the same time, bandwidth and capacity gains in datacenter networking are reducing the time and performance penalties associated with migrating containerized applications across servers. Finally, efficiencies in datacenter administration has allowed the cloud providers to

reduce the duration of renting from hours to seconds [17, 53]. Together, these three trends are pushing compute-time towards being a fungible resource, thereby enabling applications to "move with the market." This has recently been demonstrated in Supercloud project [74] by system researchers at Cornell University.

For a long time, both researchers and practitioners have called for an open commodity markets for the cloud. However, such markets have not yet materialized due to several reasons: (i) overhead and correctness issues when migrating applications across providers, (ii) security and privacy concerns that preclude customers from running their applications on unvetted platforms, and (iii) economic and scale issues of being a cloud provider that create barriers to new entrants. As a result, public cloud computing is dominated by an oligopoly of Amazon, Microsoft, and Google, who have amassed the technical expertise, trust, and scale of economy needed to operate sustainably. Though cloud computing is not an open market, and in all likeliness evolving away from it, there is tremendous internal diversity and proliferation of risks in each of the provider silos to benefit from financializing cloud applications.

There is a long history of prior research on market-based resource allocation prior to the emergence of cloud computing, e.g., [85, 80, 77, 39, 40, 59, 9, 75]. However, prior work is not applicable to todays cloud markets, as it focused on (i) optimizing resource allocation in synthetic markets using virtual currency, (ii) did not address risk management in modern distributed applications, and (iii) did not envision the diversity and complexity of cloud server contracts.

2.4.2 Financializing Cloud Computing

Financialization [1] broadly refers to the process by which exchange of goods, services and risks is increasingly facilitated via intermediation by financial institutions and instruments. By translating all economic activities into a common medium, say currency, financialization makes it easier to rationalize about assets and risks. For example, financial derivates i.e. abstract instruments whose value is derived from the value of another underlying instrument, have become an effective tool in risk management. However, the term *financialization* was originally coined by Gerald Epstein in 2001 [31] to convey the impor-

tance of financial markets, financial motives, financial institutions, and financial elites in the operation of the economy and it's governing institutions.

In our work, we use *financialization* to refer to the design of computing system frameworks, abstractions, mechanisms, and policies, influenced by finance and economics, to enable applications to be "financially-aware", i.e. to treat compute resources as investments, rationalize about their risk-reward trade-offs and manage them in a way that adapts to changes in the market. As cloud contracts increasingly resemble those in the commodity and financial markets, the case for financialized computing becomes compelling. Since compute-time represents the new "fuel" for the IT economy, organizations and users that can significantly reduce their computing costs, while limiting their risks, will gain a competitive advantage.

CHAPTER 3

MITIGATING REVOCATION RISK WITH INSURANCE

"Certainty belongs to mathematics, not to markets."

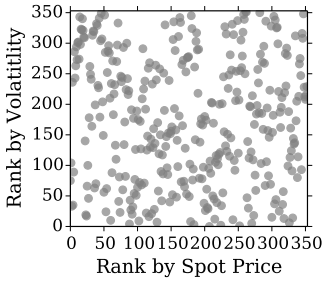
Bill Miller, Investor and portfolio manager

Spot servers expose applications to a new failure model in the form of server revocations, which are intentional and frequent but come with an advanced warning. While fault-tolerance mechanisms can be employed as insurance against spot revocations, applications need to balance their "premium" (i.e., the mechanism's cost and overhead) with their "payout" (i.e., the ability to survive revocations). In this chapter, we address this challenge for batch applications by designing a cost-aware insurance policy, which dynamically selects the best combination of spot server and fault-tolerance mechanism. Below, we present the design and evaluation of *SpotOn*, a batch compute service that incorporates this policy.

3.1 SpotOn Overview

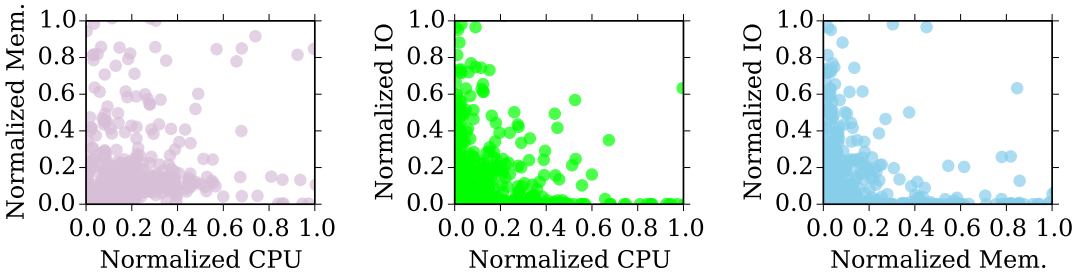
SpotOn's goal is to enable users to run their unmodified batch jobs at a performance similar to that of on-demand servers but at a price near that of spot servers. To do so, SpotOn dynamically determines (i) the best instance type and spot market to run the job, and (ii) the fault-tolerance mechanism that best balances the risk of revocation with the overhead of the mechanism. Our hypothesis is that by judiciously selecting the fault-tolerance mechanism and spot market, SpotOn can decrease the cost of running jobs, without significantly increasing the job's running time compared to using on-demand instances.

Motivation. SpotOn is motivated by two key observations (i) spot markets and batch jobs exhibit a wide range of characteristics, and (ii) the choice of best fault-tolerance mechanism is a function of both spot market and job characteristics.



Rank	Zone	Type	Volatility	Spot (¢)	On-demand (¢)
262	ap-northeast-1c	m1.large	137.51	0.50	5.23
261	ap-northeast-1c	m1.xlarge	82.10	0.50	5.23
282	us-west-1a	c1.xlarge	20.36	0.52	3.06
46	us-west-1a	m2.4xlarge	14.49	0.28	4.33
299	ap-southeast-1b	c1.medium	9.61	0.57	3.06
163	us-west-1a	m2.2xlarge	9.60	0.34	4.33
18	us-west-1a	m3.xlarge	8.83	0.25	2.63
307	ap-southeast-1a	c1.medium	8.34	0.62	3.06
347	eu-west-1a	cg1.4xlarge	6.21	1.64	6.66
39	us-west-1c	m2.2xlarge	6.07	0.26	4.33

Figure 3.1: Scatterplot of the rank in spot prices' volatility and magnitude for 353 markets (left). Table of the top 10 most volatile markets (in revocations/day when bidding the on-demand price) and their per-hour spot and on-demand price (right).



(a) CPU v.s Memory. (b) CPU v.s. IO. (c) Memory vs. IO.

Figure 3.2: Scatter-plot of normalized CPU, memory, and I/O resource usage per task in Google cluster traces.

To illustrate the former, we analyze all the 353 spot markets in the US-East-1 region over Dec-2014 to Mar-2015, Figure 3.1 gives some indication of the diversity in price characteristics across these markets. The figure shows a scatterplot of the rank of 353 markets in terms of their average spot price and volatility over the past three months (left), which demonstrates that markets differ widely in their combination of volatility and price, i.e., the lowest price is not always the least volatile. The figure also lists the top 10 most volatile markets and shows that their average spot price is as much as 10× less than the corresponding on-demand price.

To demonstrate the latter, we consider a random sample of 1000 jobs from Google cluster traces [61]. Figure 3.2 shows a scatterplot of CPU, memory, and I/O resource usage

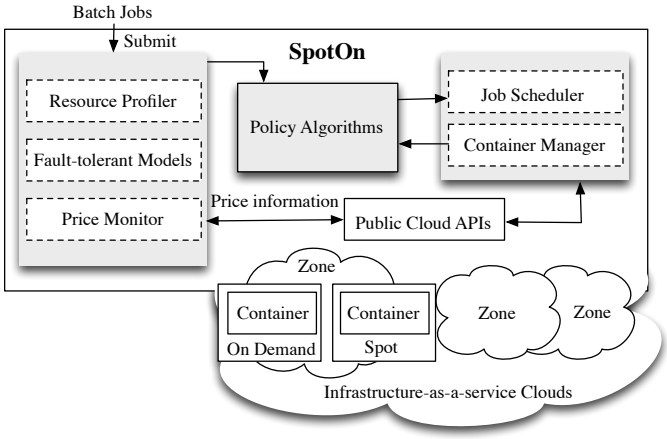


Figure 3.3: SpotOn's Architecture

for Google cluster trace jobs (normalized to the 99th percentile value). As we discuss in Section 3.2, the choice of fault-tolerance mechanism is a function of both resource usage and spot price dynamics, and is likely different for each job.

Architecture. We depict SpotOn's architecture in figure 3.3. Users interface with SpotOn by submitting their jobs as Linux Containers (LXC). We choose to package batch jobs within containers for a number of reasons. First, containers are convenient because they encapsulate all of a job's dependencies similar to a VM. Second, containers include efficient checkpointing and migration mechanisms, which SpotOn requires; unlike with VMs, the size of a container checkpoint scales dynamically with a job's memory footprint. Third, containers enable SpotOn to partition a single large instance type into smaller instances, which makes a broader set of spot markets available to run a job. Finally, containers require only OS support, and do not depend on access to underlying hypervisor mechanisms, which are typically not exposed by cloud platforms.

Based on a job's expected running time and resource usage profile, SpotOn monitors spot prices in EC2's global spot market and selects both the market and fault-tolerance mechanism to minimize the job's expected cost, without significantly affecting its completion time. SpotOn also chooses whether the job should use locally-attached or remote storage, e.g., via EBS. After making these decisions, SpotOn acquires the chosen instance from the underlying IaaS platform, configures the selected fault-tolerance mechanism, and executes

the job within a container on the instance. Upon revocation, SpotOn always continues executing a job on another instance in another market.

In the next two sections, we describe SpotOn's fault-tolerance model, and how a cost-aware selection of server is made in tandem with fault-tolerance mechanism.

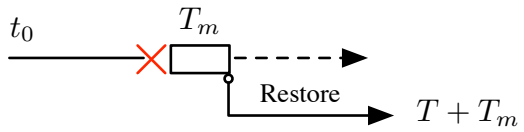
3.2 Modeling Fault-tolerance Overhead

SpotOn can employ three broad categories of system-level fault-tolerance mechanisms: (i) reactive job migration prior to a revocation, (ii) checkpointing of job state to remote storage, and (iii) replicating a job's computation across multiple instances. Each mechanism incurs different overheads (i.e., insurance premiums) during normal execution and upon revocation based on a job's resource usage. Figure 3.4 depicts these overheads, which we capture using the simple models described below.

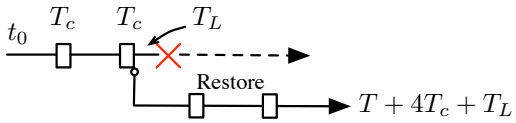
Reactive Migration. The simplest fault-tolerance mechanism is to migrate a job immediately upon receiving a warning of impending revocation. Since EC2 provides a brief two-minute warning, SpotOn can use this approach for jobs that are capable of checkpointing their local memory and disk state to a remote disk within two minutes. The time to checkpoint a job's state is a function of both the size of its local memory and disk state based on the network bandwidth and disk throughput between the job's VM instance and the remote disk. Of course, if a job's checkpoint does not complete within two minutes, this approach risks a failure that requires restarting a job. While there are many migration variants, a simple stop-and-copy migration is the optimal approach for batch jobs that permit downtime during migration.

Below, we model the migration time T_m for a job as a function of the size of its memory footprint (M) and local disk state (D), the average I/O throughput ($IOPS$) of the remote disk, and the available network bandwidth (B). We define $R_b = min(B, IOPS)$ and use R_b^s and R_b^r to represent the bottleneck when saving and restoring a job, respectively.

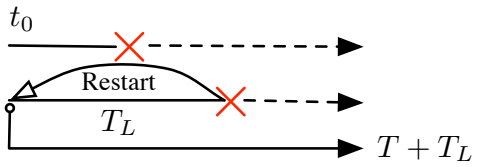
$$T_m = \frac{M + D}{R_b^s} + \frac{M + D}{R_b^r} \tag{3.1}$$



(a) Reactive Migration.



(b) Proactive Checkpointing.



(c) Replicating Compuatation.

Figure 3.4: Each fault-tolerance mechanism incurs a different overhead during normal execution and on revocation. Here, reactive migration incurs an overhead of T_m on each revocation, proactive checkpointing incurs an overhead of T_c for each checkpoint, and replicating computation incurs an overhead of T_L based on the work lost when both replicas are revoked.

The first term captures the time to save the memory and local disk state to a remote disk, while the last term captures the time to restore it. Thus, the overhead for reactive migration is a function of the magnitude of T_m and the market's volatility, i.e., the number of revocations over the job's run time.

Proactive Checkpointing.

Proactive checkpointing is an extension of migration that stores checkpoints at periodic intervals. The per-checkpoint latency T_c to checkpoint a job's state to remote disk is equivalent to the first term of the time to migrate as shown below.

$$T_c = \frac{M + D}{R_b^s} \tag{3.2}$$

With this approach, the number of checkpoints is not related to market volatility and the number of revocations, but on a specified checkpointing interval τ. Thus, the total time spent checkpointing a job with running time T is $\frac{T}{\tau} * T_c$. Importantly, proactive

checkpointing not only incurs an overhead for each checkpoint, but also requires rolling a job back to the last checkpoint on each revocation. For example, if a platform revokes a job right before a periodic checkpoint, then it loses nearly an entire interval τ of useful work. Thus, proactive checkpointing presents a tradeoff between the overhead of checkpointing and the probability of losing work on revocation: the smaller the interval τ the higher the checkpointing overhead during normal execution but the lower the probability of losing work on revocation and vice versa. This overhead is a function of a job's resource usage, i.e., its memory footprint, and the spot market's volatility.

Replicating Computation.

When replicating computation on multiple instances, the overhead is related to the magnitude and volatility of spot prices in the market, and not the size of a job's memory and local disk state. As a result, replicating computation provides SpotOn useful flexibility along multiple dimensions relative to the two previous mechanisms: (i) enables local storage, (ii) allows exploiting multiple zones/regions, and (iii) supports parallel jobs. The choice of using spot versus on-demand instances as replicas results in different overheads.

Replication on Spot Instance. Since the price of spot instances is often much more than a factor of two less than an equivalent on-demand instance, deploying multiple spot instances is often cheaper than executing a job on an on-demand instance. Given the probability of revocation P_r in each market, the completion probability P_c that at least one of n job replicas across different spot markets completes is one minus the probability that all of the jobs are revoked, or $P_c = 1 - \prod_{k=1}^{n} P_r^k$. Thus, replication across spot instances is better for shorter jobs, since they have a lower probability of all replicas being revoked.

Replication on On-demand Instance. The second replication approach is to execute a replica on an on-demand instance, which has a 0% revocation probability. To offset the expense of on-demand instance, SpotOn multiplexes multiple jobs on one on-demand instance, which effectively serves as a *replication backup server*. In this case, each job is given an isolated partition of the on-demand server's resources, such that the application executes slower than on a dedicated spot instance. On revocation, SpotOn loses any work associated with

the primary spot instance, which causes the job's progress to revert to that of the backup replica. SpotOn may then simply run the job at the slower rate, or acquire a spot instance in another market and migrate the backup server's job replica to it.

3.3 Cost-aware Insurance Policy

SpotOn employs a greedy cost-aware policy that selects the spot market and fault-tolerance mechanism in tandem to minimize a job's expected cost per unit of running time (modulo overhead) until it completes or gets revoked. SpotOn re-evaluates its decision whenever a job's state changes, e.g., due to a revocation, in order to select a new instance type and market to migrate the job.

We profile each spot market as a function of jobs' remaining running time, T. In particular, we define a random variable Z_k for each spot market k to represent the amount of time a job can run on a spot instance without being revoked. We then define the probability that Z_k is less than a job's remaining running time $P_z = P(Z_k \leq T)$, which represents the probability that a job's spot instance from market k is revoked before it completes. We use $E(Z_k)$ to denote the expected time a job executes before being revoked. For a given running time T, we can compute both $P(Z_k \leq T)$ and $E(Z_k)$ over a recent window of prices, e.g., the past day, week, or month. For each spot market k, we also maintain the average spot price $\bar{C}_{sp}^k$, and the ratio of on-demand to spot price ratio, r. Finally, we determine the optimal checkpointing frequency τ based on the job and market characteristics [29]. We use these values in computing the expected cost $E(C_k)$ and expected time $E(T_k)$ for running each job in a particular spot market k, as summarized in table 3.1. Below, we detail how to compute the expected cost per unit of running time for each of the three mechanisms.

Expected Cost of Migration. In this case, the expected cost until the job either completes or is revoked is the probability the job is revoked (which is a function of its remaining running time) multiplied by the cost of running the job to the first revocation plus the probability the job finishes without being revoked multiplied by the cost of running the job to completion. On each revocation, the job incurs migration overhead T_m. On similar lines, we determine the expected time the job either completes or is revoked. Since reactive

Mechanism	Expected cost and run-time
Reactive Migration	$E(T_k) = (1 - P_z) * T + P_z * (E(Z_k) - T_m)$ $E(C_k) = \left[P_z * E(Z_k) + (1 - P_z) * T\right] * \bar{C}_{sp}^k$
Proactive Checkpointing	$E(T_k) = E(Z_k) - \frac{E(Z_k)}{\tau} * T_c - \frac{\tau}{2}$ $E(C_k) = \left[P_z * E(Z_k) + (1 - P_z) * T\right] * \bar{C}_{sp}^k$
Replication (Spot)	$E(T_k) = P_c * T$ $E(C_k) = (1 - P_c) * \sum_{k=0}^{n} (\bar{C}_{sp}^k E(Z_k)) + P_c \sum_{k=0}^{n} \bar{C}_{sp}^k T$
Replication (On-demand)	$E(T_k) = P_z * \frac{E(Z_k)}{r} + (1 - P_z) * T$ $E(C_k) = \left[P_z * E(Z_k) + (1 - P_z) * T\right] * (1 + \frac{1}{r}) * \bar{C}_{sp}^k$

Table 3.1: Expected runtime and cost under different fault-tolerance mechanisms

migration is the best option if migration is feasible, SpotOn generally uses it whenever a job's memory footprint permits migration within the two minute warning.

$$\begin{aligned} E(T_k) &= (1 - P_z) * T + P_z * (E(Z_k) - T_m) \\ E(C_k) &= \left[P_z * E(Z_k) + (1 - P_z) * T\right] * \bar{C}_{sp}^k \end{aligned} \tag{3.3}$$

Expected Cost of Checkpointing. The expected cost of checkpointing is based on the checkpointing interval (defined by the slack) and the potential loss of work due to revocations. As above, we compute the expected cost until a job either completes or is revoked. However, in computing the expected job running time $E(T_k)$, we subtract the useful work completed by the job based on the checkpointing interval and any work lost on the revocation.

$$\begin{aligned} E(T_k) &= E(Z_k) - \frac{E(Z_k)}{\tau} * T_c - \frac{\tau}{2} \\ E(C_k) &= \left[P_z * E(Z_k) + (1 - P_z) * T\right] * \bar{C}_{sp}^k \end{aligned} \tag{3.4}$$

Expected Cost of Replication (Spot). When replicating across spot instances, we do not re-run our selection policy on each revocation. Instead, if all spot instances are revoked, we re-start the job on an on-demand instance to ensure the job completes. The expected cost when replicating a job with remaining running time T across n spot markets is the expected cost if all spot instances are revoked plus the expected cost if the job completes,

weighted by the probability of each event occurring. Here, P_c and P_r are the probability of a job completing and being revoked; C_{sp}^k is the average price of the spot instance for market k.

$$
\begin{aligned}
E(T_k) &= P_c * T \\
E(C_k) &= (1 - P_c) * \sum_{k=0}^{n} (\bar{C}_{sp}^k E(Z_k)) + P_c \sum_{k=0}^{n} \bar{C}_{sp}^k T
\end{aligned}
\tag{3.5}
$$

Expected Cost of Replication (On-demand). Computing the cost of replicating on a "slower" and cheaper on-demand instance is similar to checkpointing, except that we incur an additional cost for the discounted on-demand instance. Here, we assume SpotOn pays the same price for the backup on-demand instance as it does for the primary spot instance, which mirrors the price for replicating across two spot instances above, and makes the different replication approaches comparable. In this case, if the ratio of the on-demand to spot price is r, then we assume the remaining running time of our job on the backup instance is $r * T_i$, since we partition the resources of the backup on-demand instance based on its price. The expected cost below is then similar to checkpointing, but multiplies the price of the spot instance by a factor of two to account for the cost of the primary spot instance and the backup on-demand instance.

With on-demand replication, if our primary spot instance is revoked, the useful work done is dictated by the progress of the backup server, which is running a factor of r slower than the primary. Note that unlike checkpointing, the useful work lost on each revocation is a function of the ratio r and not a fixed checkpointing interval τ. Thus, while the fraction of work lost on a revocation at any time remains the same, the absolute work lost increases with job running time. Developing mixed policies that periodically checkpoint the on-demand backup server to mitigate the impact of using on-demand replication for long running jobs is future work. Thus, we can compute the expected job run time as below.

$$
\begin{aligned}
E(T_k) &= P_z * \frac{E(Z_k)}{r} + (1 - P_z) * T \\
E(C_k) &= \left[P_z * E(Z_k) + (1 - P_z) * T\right] * (1 + \frac{1}{r}) * \bar{C}_{sp}^k
\end{aligned}
\tag{3.6}
$$

Putting it all together. Given a job's resource vector, our cost-aware policy uses a brute-force approach that simply computes the expected cost of using each fault-tolerance mechanism until the job either completes or is revoked across each spot market, and then chooses the least cost mechanism and market. When acquiring the selected server, SpotOn needs to set a bid level. Prior work [66] has shown that current spot markets tend to spike from very low to very high, and thus results in a long-tailed price CDF that is not very responsive to minor changes in bidding level. Thus, while SpotOn can be configured to bid at any desired level, the default option is set to that of the on-demand price level.

3.4 Implementation

We implement a prototype of SpotOn on EC2 in `python`. The prototype includes a job manager hosted on an on-demand instance and agent daemons that run on each spot instance. Users package SpotOn jobs as Linux Container (LXC) images, which include the entire state necessary to run the job (including any operating system libraries). The image includes a start script at a well-known location within the image that SpotOn executes to launch the job. Users store the image in a known directory inside an EBS snapshot in EC2, which they authorize SpotOn to access. Users then submit jobs by selecting their instance type and provide SpotOn an identifier for the EBS snapshot hosting their job's container image. To control the use of local versus remote EBS storage, jobs write intermediate data to and from a well-known directory, which SpotOn configures to be either attached to an EBS volume or attached to the local disk.

SpotOn's job manager selects the EC2 spot market and fault-tolerance mechanism for each job based on the cost-aware policy in Section 3.3. To execute the policy, the job manager monitors and records spot prices across EC2 markets. For each market, the job manager computes the expected cost of each fault-tolerance mechanism using the historical price data, as well as the the job's running time and resource usage vector. Our current prototype assumes a job's running time and resource usage vector are accurate and does not monitor a job's resource usage while it is running. In addition, our current prototype does not support "phased" jobs, where resource usage changes significantly during different phases of execution. After computing the expected cost for each market and fault-tolerance

mechanism, the job manager selects the least cost fault-tolerance mechanism and spot market combination to run the job. The job manager interacts with EC2 to monitor prices, place bids, and fetch instance information using the EC2 web services APIs. If the current spot price in the market is above the on-demand price, then the job manager selects the market with the next lowest expected cost.

Once EC2 allocates the spot instance, the job manager launches a small agent daemon within the instance, which it uses to remotely execute commands to launch the container and start the job. To issue a termination warning, EC2 writes a termination time into the file `/spot/termination-time` on the spot instance, which the agent polls every five seconds. Upon receiving a warning, the agent notifies the job manager, which selects a new instance type using the same policy as above based on the remaining running time of the job. One exception is for the replication across spot policy, which does nothing on each revocation, but rather restarts a job only after all replicated instances have been revoked. The job manager computes the remaining run time by subtracting both the completed running time and the overhead of checkpointing and migration operations. For checkpointing, the job manager takes a container checkpoint at a periodic interval using CRIU (Checkpoint in User Space) for LXC via the agent based on the slack. The job manager takes EBS snapshots at the same time to checkpoint the disk.

To ensure network connectivity, SpotOn uses Virtual Private Clouds (VPC) in EC2 to manage a pool of IP addresses. The VPC allows the application provider to assign or reassign any IP address from their address pool to any instance. We assume that batch jobs need not be externally contacted but that batch jobs may need to access the public Internet. NAT-based private IP addresses suffice for this purpose and we assume that the VPC manages a pool of NAT-based private IP addresses, one of which is assigned to each SpotOn container. Upon migration, after stopping the container, the job manager detaches the container's IP address from the original instance and reattaches it to the new instance.

The job manager also detaches the container's EBS volume from the original instance and reattaches it on the new instance. When rolling back to a previous checkpoint, the job manager reattaches the EBS snapshot of the disk associated with the last container

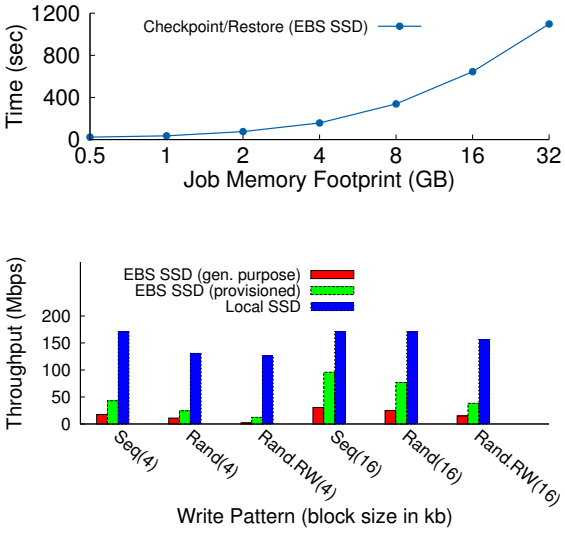


Figure 3.5: The time to checkpoint-restore a container is a function of a job's memory footprint (top). The I/O throughput for local disks is an order of magnitude greater than for remote disks over a range of workloads (bottom).

checkpoint. Once the IP address and EBS volume are attached, the job manager restarts the container on the new instance from the last checkpoint.

3.5 Evaluation

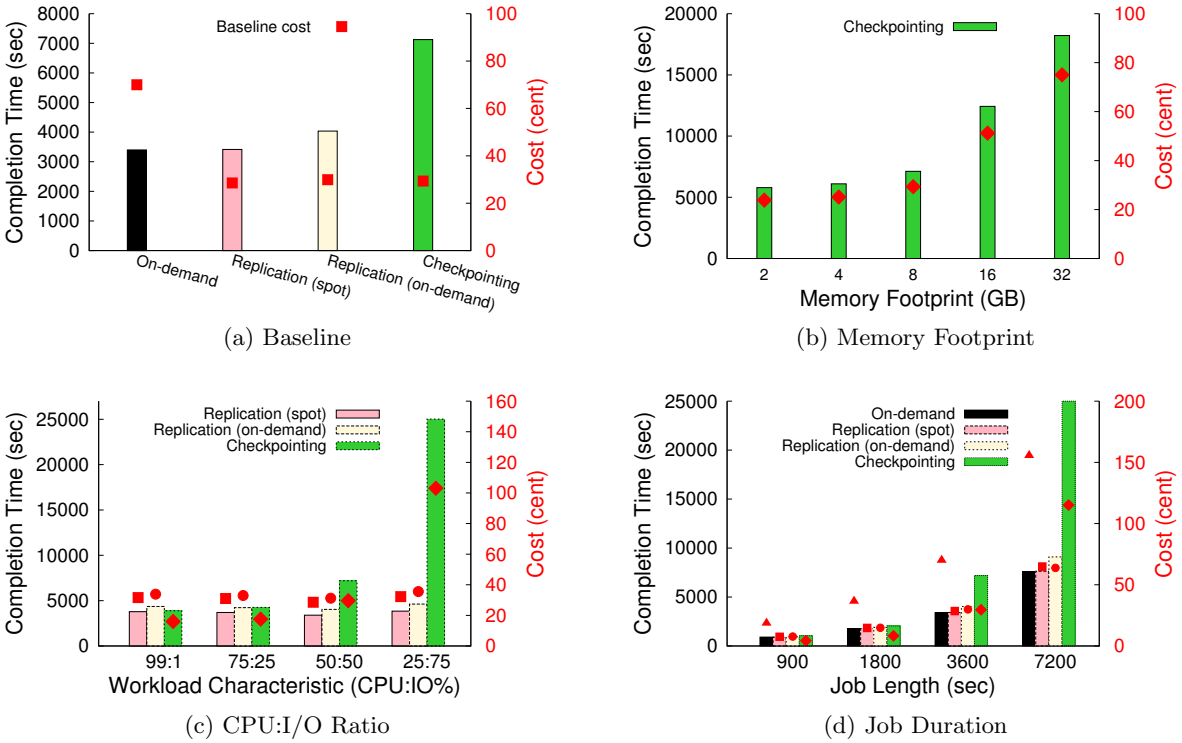
The goal of our evaluation is to quantify the benefit of SpotOn's cost-aware selection policy that chooses the fault-tolerance mechanism and spot market to minimize costs, while mitigating the impact of revocations on job completion time. We compare the cost and performance of our policy with three other policies: a control policy that always executes jobs on an on-demand instance, our basic policy that always selects the lowest price spot market using checkpointing and reverts to an on-demand instance on the first revocation, and a variant of our cost-aware policy that only uses checkpointing. We conduct experiments using our prototype and in simulation. The prototype experiments demonstrate the impact of resource usage and price characteristics on real jobs, while the simulations assess the impact on performance and cost when using our cost-aware policies to execute multiple jobs over time with realistic price traces.

We first conduct microbenchmarks to verify the assumptions of our models and to seed our simulator. In particular, we plot LXC checkpoint/restore time in Figure 3.5(top) as a function of a job's memory footprint to verify relationship between checkpoint/restore overhead and memory. The graph demonstrates that it is possible to migrate jobs that use less than roughly 4GB of memory within EC2's two-minute warning time. In addition, for Figure 3.5(bottom) we use the FIO tool to measure the local versus remote EBS storage throughput for multiple I/O workloads (in this case using the SSD variant of EBS); we see that, as expected, the local I/O throughput is an order of magnitude larger than the remote EBS throughput, which favors using local storage for I/O-intensive jobs. Our simulator uses Figure 3.5 to compute a job's checkpoint/restore and I/O overhead based on its resource usage. The simulator also imposes delays of 62 seconds and 224 seconds for booting an on-demand and spot instance, respectively, based on our experiments.

Prototype Results

We use our prototype to examine the impact of resource usage and spot price characteristics on a job's performance and cost. To do this, we write a synthetic job emulator that enables us to set a job duration, working set size, and CPU:I/O ratio on a reference machine. Using our emulator, we first create a baseline job that runs for roughly one hour, has a memory footprint, i.e., working set size, of 8GB, and has a CPU:I/O ratio of 1:1. That is the job spends half its time computing and half its time waiting on I/O to complete.

For our baseline experiment, we assume the cost of the spot instance is 20% of the cost of the on-demand instance and the revocation rate is 2.4 revocations per day (or 0.1 revocations per hour). We chose 2.4 revocations per day as a median between the extreme values in Figure 1 and the many markets that currently experience nearly zero revocations per day. We execute the job on a `r3.2xlarge` instance type, which costs 70¢ per hour, and measure its average completion time across multiple runs to be 3399s. Figure 3.6a shows the job's completion time (each bar corresponding to the left y-axis) and its cost (each dot corresponding to the right y-axis) when running on an on-demand instance versus running on a spot instance and i) replicating on a backup on-demand instance, ii) replicating across two spot instances, and iii) checkpointing every 15 minutes. To fairly compare the two



(a) Baseline

(b) Memory Footprint

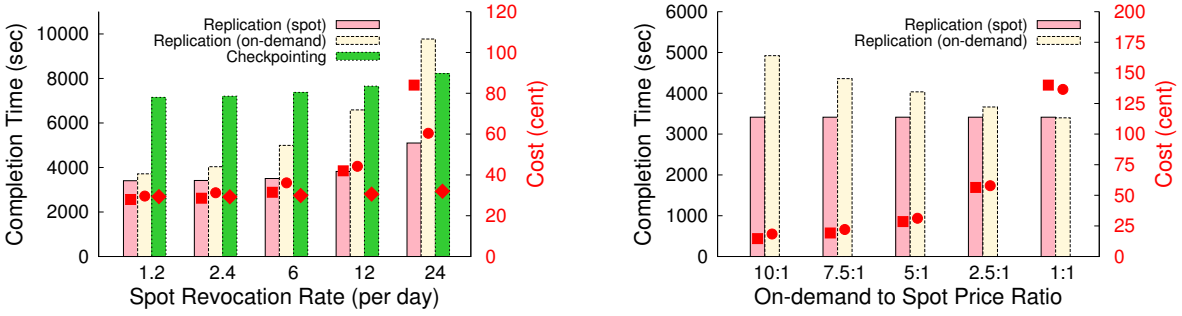
(c) CPU:I/O Ratio

(d) Job Duration

Figure 3.6: Performance and cost for our baseline job when running on an on-demand instance versus running on spot instances using different fault-tolerance mechanisms (a). We also plot them as job's memory footprint (b), CPU:I/O ratio (c), and duration (d) vary.

replication approaches, when replicating on a backup on-demand instance we assume the job runs at 20% the performance of the dedicated instance and is charged 20% of the cost of the backup.

Our baseline experiment shows that both forms of replication and checkpointing reduce the job's cost by over a factor of two compared to running on an on-demand instance. However, both replication mechanisms complete the job sooner than when using checkpointing. The reason is that the probability of revocation over the job's running time is only 10%, so 90% of the time the job will finish without incurring any performance overhead due to a revocation. In contrast, checkpointing repeatedly incurs the overheads from Figure 3.5. In addition, checkpointing requires using a remote disk to facilitate migration, while replication is capable of using the local disk. Thus, replication benefits from the I/O intensity of our baseline job. Note here that the cost of replicating on a backup server and checkpointing is similar, since the backup server doubles the cost (as we fix the amount we pay for the



(a) Revocation Rate

(b) On-demand:Spot Price Ratio

Figure 3.7: The impact of varying the spot revocation rate (a), and on-demand:spot price ratio (b) on our baseline job's performance and cost.

backup server to be equal to that of the spot instance), while checkpointing nearly doubles the running time, which also doubles the cost.

Figure 3.6 also plots the job's performance and cost as its memory footprint and CPU:I/O ratio change. Figure 3.6b shows that, as expected, an increase in the memory footprint causes an increase in the overhead of checkpointing, while it has no effect on the replication approaches. Figure 3.6c then shows that, as the job becomes more I/O-intensive, the job completion time and cost of checkpointing rise due to the need to use remote I/O. In contrast, the cost and performance of the replication approaches remain constant. Note that for CPU-intensive jobs the cost of replication is slightly more than the cost of checkpointing, as there is less benefit to using the local disk, but both variants of replication incur the cost of additional compute resources.

Figure 3.6d shows that checkpointing has the lowest cost for short jobs (< 1 hour), since short jobs require fewer checkpoints and less overhead. However, the longer the job, the higher checkpointing's overhead and cost. While the overhead of both replication variants also increase with job duration, due to the increased probability of losing work due to revocation, the increase is less than with checkpointing.

Figure 3.7a shows that as the revocation rate increases the cost and performance of replication becomes worse relative to checkpointing. Replication is highly sensitive to the revocation rate, since revocation's result in rolling back to either the progress of the slower backup server or to the start. In contrast, checkpointing's cost and performance is more

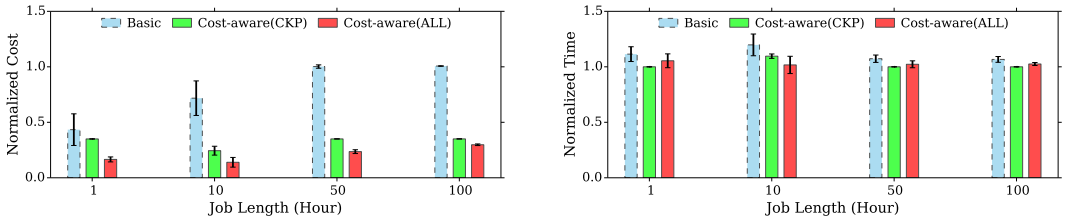


Figure 3.8: Job cost (a) and performance (b) as a function of the job length.

robust to an increasing revocation rate, since it only loses at most the smaller time window between each checkpoint. The figure also demonstrates the key difference between replication across spot and replication on on-demand: under a high revocation rate (24 per day) replication across spot has low running time, but a high cost (since it reverts to using an on-demand instance), while replication on on-demand has a higher running time but a much lower cost, since it always makes progress. Finally, Figure 3.7b shows that as spot prices rise relative to the on-demand price, the replication variant that uses an on-demand backup server takes longer to complete. This is due to increased multiplexing of jobs on the backup server at a higher spot price. Since checkpointing and replication across spot do not use an on-demand backup server, they are robust to this effect.

Result: *The relative performance and cost of each fault-tolerance mechanism is a complex function of a job's duration, memory footprint, and CPU:I/O mix, as well as the spot price's magnitude and volatility.*

Policy Results

We use our simulator to assess SpotOn's cost and performance over a long period of time; in this case, we consider the price for all spot instances in the `us-east-1a` zone over three months from December 2014 to March 2015. Our simulator assumes users submit jobs to run on `m1.large` instance types. Here, we normalize the job's performance and cost for each policy to the performance and cost of executing the job on a dedicated on-demand instance. For the next set of experiments, we use a baseline job that has a memory footprint of 7.5GB and a running time of ten hours on an `m1.large` on-demand instance, such that we fix the checkpoint frequency to be hourly based on the slack.

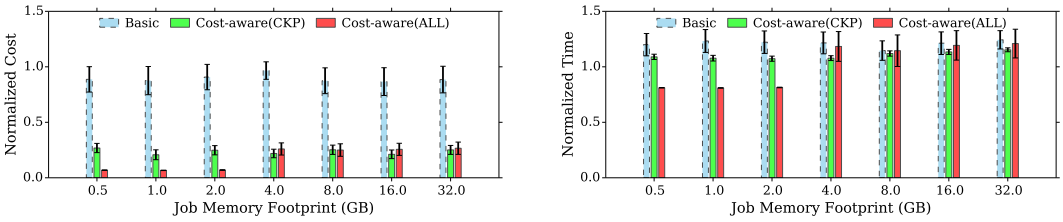


Figure 3.9: Job cost (a) and performance (b) as a function of the job's memory footprint.

We first evaluate SpotOn's selection policies as we vary the duration of the jobs. We simulate the execution of 30 jobs arriving randomly over the three-month time window and record the cost and completion time for each of our policies. Figure 3.8 shows the results normalized to the cost and completion time when using an on-demand instance. We include error bars for the 95% confidence intervals over the 30 jobs. The results demonstrate that all cost-aware policies incur a lower cost than using an on-demand instance for a similar performance level. In addition, our basic policy, which always selects the spot instance with the lowest price without regard to volatility and migrates to an on-demand instance after the first revocation, has a significantly higher cost than either of our cost-aware policy variants, which demonstrates the benefits of considering volatility in addition to price when choosing a market. Our cost-aware policy also has a lower cost than a variant that only uses checkpointing, which demonstrates the benefit of using replication in addition to checkpointing. However, the cost benefit of replication decreases as job duration increases, since the probability of revocation increases (which in turn increases the overhead of replication). Finally, the completion time for each approach is similar and near the completion time of the job on an on-demand instance.

Result: *SpotOn's cost-aware policy reduces the cost of running a job by as much as 86% compared to running on an on-demand instance, while increasing the job's completion time by only 2%. When compared to a cost-aware policy that only uses checkpointing, SpotOn reduces cost by up to 74%, again while increasing job completion time by only 2%.*

Next, Figure 3.9 examines the impact of a job's memory footprint on each policy. As before, all policies reduce costs relative to on-demand with the same level of performance. In fact, for small jobs less than 4GB, the completion time is lower than using an on-demand

instance. This is due to inversions in spot market prices, where the spot price of a particular instance type drops below the spot price of a smaller instance type. Since SpotOn always seeks the lowest-cost resources, it takes advantage of these price inversions. Figure 3.9 also shows that jobs with memory footprints that use less than 4GB incur a much lower cost and have higher performance than jobs that use more memory. This occurs because reactive migration is feasible in this case, and reactive migration does not incur the performance overhead of checkpointing or the cost overhead of replication. When reactive migration is not possible after 4GB, the cost-aware and cost-aware checkpointing policies have a similar cost with performance similar to an on-demand instance.

Result: *Using reactive migration for jobs with low memory footprints substantially decreases costs, in this case by over a factor of four, due to its low overhead. Our cost-aware policy uses reactive migration whenever it is feasible.*

We next examine the impact of the revocation rate on cost and performance. Here, we synthetically inject revocations at specific rates in the price trace to observe their impact. As the revocation rate increases, we see that the cost savings from our cost-aware policy relative to a cost-aware policy that only uses checkpointing decreases. This occurs because the overhead of replication increases under more volatile market conditions more than the overhead of checkpointing. However, in each case, our cost-aware policy has a lower cost than the other policies, since it chooses checkpointing only when it is the lowest cost option. The increase in revocation rate also increases job completion times, but in all cases the completion time remains near the completion time when using an on-demand instance. As before, price inversions combined with low revocation rates result in our cost-aware policy executing jobs faster than when using on-demand instance in some cases.

Result: *The benefit of using replication in addition to checkpointing decreases as the revocation rate increases. Since SpotOn's cost-aware policy chooses checkpointing when it is the lowest cost option, it results in the lowest cost and highest performance across all policies and revocation rates.*

Lastly, to get a sense of SpotOn's potential for savings with a real workload, we randomly select 1000 tasks from a Google cluster trace [61] and compare the cost of SpotOn's greedy cost-aware policy and running the jobs on an `m1.large` on-demand instance. Our results

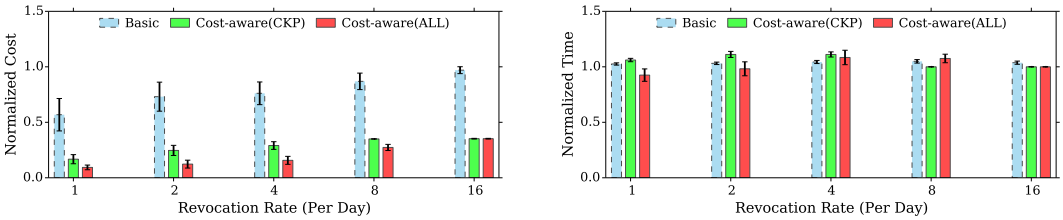


Figure 3.10: Job cost (a) and performance (b) as a function of the revocation rate.

show a cost savings of 91.9% when using SpotOn's cost-aware policy versus the `m1.large` on-demand instance. In addition, the total running time across all jobs when using SpotOn actually *decreases* by 13.7%. In this case, the decrease occurs because SpotOn often chooses to execute jobs on spot instance types that are faster than the `m1.large` because their spot price is actually cheaper than an `m1.large` on-demand instance.

3.6 Related Work

SpotOn is similar to recent startup companies, such as ClusterK [54] and Spotinst [46], that offer low prices by executing batch jobs submitted by users on spot instances. However, their policies for handling revocations are not public, so it is unclear if they restart jobs if spot instances fail, or if they use fault-tolerance mechanisms to mitigate the impact of revocations.

Prior work examines bidding [51, 97, 76, 96, 81, 50] and checkpointing [84, 44, 95] policies for batch jobs to minimize the cost of spot instances and mitigate the impact of revocations. This work generally evaluates bidding and checkpointing policies in simulation without considering how job resource usage affects their overhead (and cost) relative to other fault-tolerance mechanisms. While these prior works have largely focused on checkpointing as the fault-tolerance mechanism, one exception is the work by Voorsluys and Buyya [84], which considers replicating computation across two spot instances. However, since they only consider simulated compute-intensive jobs where the cost of checkpointing is low, they find replication performs poorly by comparison; our results indicate replication is effective at current spot prices, especially for I/O-intensive jobs, since it enables use of local storage.

In recent work, researchers have designed system frameworks and middleware to run different classes of applications on spot servers including interactive applications [68], map-reduce jobs [99], in-memory cache [88], in-memory storage [94] and machine learning [36]. However, these applications require low latency and high uptimes, which preclude spot servers outside of the current zone, and certain fault-tolerance mechanisms. By focusing narrowly on batch jobs that permit some downtime, SpotOn has much more flexibility, enabling it to chose from multiple fault tolerance mechanisms, exploit spot markets in multiple regions, and use local storage.

3.7 Conclusion

SpotOn optimizes the cost of running non-interactive batch jobs on the spot market. We design a policy to balance the tradeoff between the fault-tolerance overhead and the ability to recover from revocations. Our results using Google cluster traces and Amazon EC2 market prices demonstrate that batch applications can benefit from inexpensive spot servers despite their challenging failure model.

Status. SpotOn has been implemented using Linux containers and prototyped on Amazon EC2. Additional details on its design, implementation and evaluation are in [78].

CHAPTER 4

REDUCING PRICE RISK THROUGH ACTIVE TRADING

"Every man lives by exchanging."

Adam Smith, Wealth of Nations (1776)

Cloud spot markets expose applications to *price risk* i.e., the risk that a VM's price will increase relative to others. Since spot prices vary continuously across hundreds of different types of VMs, flexible applications can mitigate price risk by moving to the VM that currently offers the lowest cost. To enable this flexibility, we present HotSpot, a resource container that "hops" VMs—by dynamically selecting and self-migrating to new VMs—as spot prices change. HotSpot containers define a migration policy that lowers cost by determining when to hop VMs based on the transaction costs (from vacating a VM early and briefly double paying for it) and benefits (the expected cost savings). As a side effect of reducing the price risk, HotSpot is also able to reduce the number of revocations without degrading performance. HotSpot is simple and transparent: since it operates at the systems-level on each host VM, users need only run an HotSpot-enabled VM image to use it. We implement a HotSpot prototype on EC2, and evaluate it using job traces from a production Google cluster. We then compare HotSpot to using on-demand VMs and to spot VMs (with and without fault-tolerance) in EC2, and show that it is able to lower cost and reduce the number of revocations without degrading performance.

4.1 The Importance of Price Risk

While using fault-tolerance-based approaches on spot VMs offers significant cost savings relative to using on-demand VMs, revocations are not frequent events in EC2's current market, and thus the savings relative to using spot VMs without any fault-tolerance is

Figure 4.1: The average Time-to-Revocation (TTR) for 402 Linux spot VMs in EC2's `us-east-1` region when bidding at the on-demand price and 10× the on-demand price over a two month period. The average TTR across all servers is ∼25 and ∼47 days, respectively, for 1× and 10× bids.

often not significant. To understand why, recall that users do not pay their bid price for VMs, but instead pay the spot price. As a result, there is no penalty for bidding high, as long as applications are flexible enough to switch to lower cost VMs if their current VM's price rises [66]. Thus, even a simple strategy that bids well above the on-demand price is highly effective, since applications can prevent revocations by shifting to a cheaper fixed-price on-demand VM once the spot price rises to near the on-demand price, and before it comes close to the bid price. Since a VM's spot price rarely exceeds its on-demand price, the revocation rate at the on-demand bid level is low for most spot VMs.

Figure 4.1 illustrates this point by showing the average Time-to-Revocation (TTR) over a two month period (from 2017-03 to 2017-04) for 402 Linux spot VMs across five AZs of the `us-east-1` region when bidding the on-demand price and when bidding 10× the on-demand price. As the graph shows, while a few VMs have low TTRs, the vast majority of VMs have high TTRs. The horizontal lines show the number of VMs that did not experience any revocation over the two month period, which includes >35% and >75% of spot VMs when bidding the on-demand price and 10× the on-demand price, respectively. Overall, the average TTR across the 402 Linux spot VMs in `us-east-1` is ∼25 and ∼47 days when bidding the on-demand price and 10× the on-demand price, respectively.

These results reflect that spot prices often experience long periods of stability interspersed with short periods of volatility, as illustrated in Figure 4.2. In this case, the

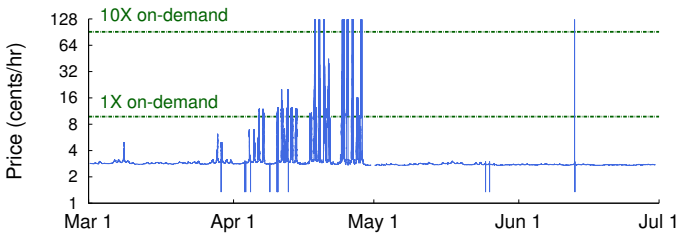


Figure 4.2: Example spot price trace for an `m4.large` VM with long periods of price stability and short periods of volatility.

`m4.large` spot VM in `us-east-1a`[1] maintains a low and stable spot price for much of the four month period, while experiencing a few highly volatile periods. As the graph shows, during volatile periods, the spot price can rise higher than the on-demand price. These steep rises could occur for many reasons including: "convenience bidding" where users bid high assuming the spot price will not spike and do not vacate VMs when it does [23]; EC2 reclaiming spot VMs by artificially raising their price; or unavailability of on-demand VMs that increases demand for spot VMs [57]. Regardless of the reason, applications should react to price spikes by vacating high-priced spot VMs.

Our analysis above indicates that revocation risk in the current market is low, although it could increase in the future, especially if EC2 alters its bidding rules such that users pay their bid price instead of the spot price, or if users adopt our optimizations and those proposed in prior work. In contrast, the market's price risk—or the risk that a VM's price will increase relative to others—is much higher than the revocation risk. We quantify price risk by measuring the average difference between the price of each AZ's cheapest VM (in terms of its normalized cost per unit of resource) at t_0 and its price at each time $t > t_0$ as the market's cheapest VM changes. In this case, we normalize relative to a VM's EC2 Compute Unit (ECU)—Amazon's measure of a VM's integer processing power [10].

The difference between the price of the cheapest server at t_0 and the price of the cheapest server as it changes reflects the cost savings possible from hopping VMs assuming the application i) can fully utilize a VM of any capacity, ii) incurs no overhead when migrating

[1]Note that AZ labels are not consistent across EC2 accounts. For example, one account's `us-east-1a` may be labeled as `us-east-1b` under another account.

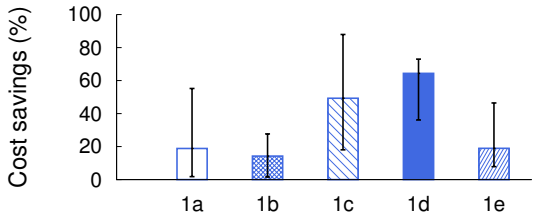


Figure 4.3: Ideal cost savings from automated VM hopping within each AZ in the `us-east-1` region over one month.

between VMs, and iii) does not experience revocations. Under these assumptions, any approach that does not hop to the cheapest VM incurs a higher cost. Note that we relax these assumptions in HotSpot's design (§4.3), since in practice applications cannot always fully utilize VMs of any capacity and do incur an overhead when migrating. As a result, our analysis here only sets an upper bound on HotSpot's cost savings, and does not reflect the migrations HotSpot would actually make.

Figure 4.3 shows the potential cost savings from hopping VMs within each AZ of the `us-east-1` region over two months starting 2017-03. The average savings in each AZ is between 15% and 65% with an average of 33% across all AZs. Since the results are dependent on the cheapest VM's price at the start of each interval, the error bars reflect the maximum and minimum savings from 10 randomly selected start times within the two month period. The low minimum savings reflect times where the price of the cheapest VM at t_0 remained stable and was always near that of the dynamic cheapest VM. Note that the figure represents additional savings relative to using spot VMs without hopping, which is already significantly cheaper (∼50-90%) than using on-demand VMs.

Figure 4.4 then shows the average time until the cheapest VM in the market changes, which we call the Time-to-Change (TTC), on the y-axis for each AZ of the `us-east-1` region. In this case, the cheapest VM across the 402 spot VMs changes every 1.1 hours, which shows that there is an opportunity to reduce cost by migrating to the cheapest spot VM. Since the TTC is two orders of magnitude less than the TTRs in Figure 4.1, applications are much more likely to experience a change in the cheapest spot VM during

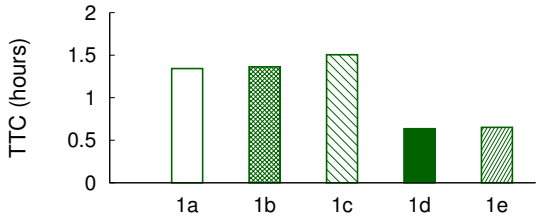


Figure 4.4: The Time-to-Change (TTC) for the cheapest VM in each of AZs of the `us-east-1` region over a two month period. The average TTC across all AZs is 1.1 hours.

their execution than a revocation. Of course, each change in price of the cheapest spot VM might be small relative to the cost of a revocation.

As mentioned above and discussed in §4.3, HotSpot containers migrate to new cloud VMs to maximize their cost-efficiency. Thus, the migration policy does not consider either revocation risk or application performance, which are both important metrics. In general, at any time, the lowest cost VM is not necessarily the one with the absolute lowest revocation risk or highest capacity. As a result, migrating to optimize cost-efficiency has the potential to increase revocation risk and decrease performance. Of course, HotSpot could define a migration policy that also considers revocation risk and performance. Configuring such a migration policy would require users to compare and weight the relative importance of each metric. Fortunately, as we show below, hopping VMs to minimize cost tends to also lower revocation risk and does not decrease performance on average. Thus, HotSpot's migration policy focuses on minimizing cost, and does not consider revocation risk or performance.

4.1.1 Impact on Revocation

Prior work generally estimates revocation risk in terms of a spot VM's TTR at a given bid level based on its historical spot price over some previous time period, e.g., a few days to weeks. However, since HotSpot frequently switches VMs, its revocation risk is not a function of any single spot VM's TTR, but the TTR of the multiple VMs it uses (weighted by the time it spends on them). Our key insight is that there is a relationship between a VM's instantaneous revocation risk and its current spot price relative to its on-demand price. This relationship derives from the observation that the lower the ratio of the spot price to the on-demand price, the further away the supply/demand balance is from being

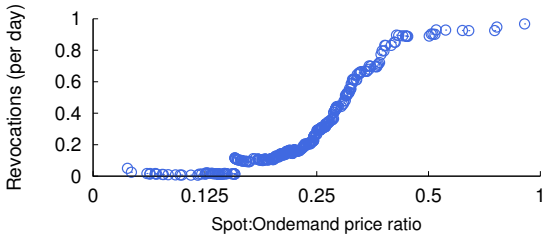


Figure 4.5: The x-axis is the spot-to-on-demand ratio, while the y-axis is the average revocation rate across spot VMs when the spot price is less than or equal to the x-axis value.

constrained and thus causing a spike in prices that triggers revocations to occur. That is, assuming the spot price is based on supply and demand (as Amazon claims [6]), the lower a VM's spot price relative to its "risk free" on-demand price, the greater the change in the balance of supply and demand required for the spot price to rise above the on-demand price. As a result, a spot VM's spot-to-on-demand price ratio is an indirect measure of its instantaneous revocation risk relative to other spot VMs.

Figure 4.5 illustrates the relationship between the revocation risk and the spot-to-on-demand ratio across 402 Linux spot VMs in the `us-east-1` region from 2017-03 to 2017-04. The graph shows that, as the average spot-to-on-demand ratio decreases, the average revocation risk also decreases. While the spot VM with the lowest spot-to-on-demand ratio is not always the same as the most cost-efficient one at any time, the most cost-efficient VM often has a low spot-to-on-demand ratio. This correlation exists, in part, because on-demand prices for VMs in the same family are uniform when normalized per unit of resource, so a low normalized spot price implies a low spot-to-on-demand ratio relative to other spot VMs in the same family [2]. The cost per ECU-hour for VMs in different families is also similar. To illustrate, Figure 4.6 shows the on-demand price per ECU-hour for all families in the `us-east-1` region. The c4, m4, r4, and i3 families range from 1.25-2.56 ¢/ECU-hour, while the more specialized memory-optimized (x1), storage-optimized (d1), and GPU instances (p2) have a higher normalized cost.

However, as we discuss in §4.3, HotSpot normalizes spot prices per unit of resource *utilized*, so variations in an application's workload also affect a VM's cost-efficiency. In this case, we define utilization as the VM's average CPU utilization. As a result, if a VM has a

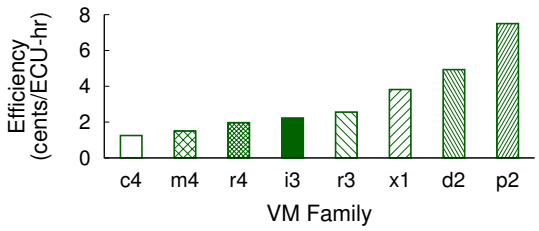


Figure 4.6: The cost-efficiency of on-demand VMs in the `us-east-1` region for different types of VMs in EC2.

low average CPU utilization, e.g., $\ll$100%, then it is wasting CPU capacity, which increases the relative cost of the CPU capacity it utilizes. Thus, there is the possibility that the VM with the lowest spot-to-on-demand ratio could have a high capacity that is over-provisioned for the application and wastes resources, resulting in a low cost-efficiency. If only high-capacity VMs over-provisioned for an application were to have a low spot-to-on-demand ratio, it is technically possible for the most cost-efficient VM to have a high revocation risk. However, we have not seen this scenario occur, as there are usually many spot VMs at all capacities with low spot-to-on-demand ratios. Thus, given the correlations above, migrating to the most cost-efficient VM results in a low revocation risk. For example, in the experiment from Figure 4.3, the most cost-efficient VM was *never revoked* over the two month period.

Of course, the high TTRs in Figure 4.1 from §4.1 show that revocation risk is not a serious concern in EC2. However, our insight above implies that i) hopping VMs to reduce price risk and lower cost will not increase revocation risk, and ii) if spot prices were to become more volatile, VM hopping could reduce revocation risk. Table 4.1 illustrates the latter point. Here, we emulate a more volatile market by taking the 10 most volatile spot VMs in the `us-east-1` region and assume that we can only migrate among them. The table shows the average revocation rate for each of these volatile VMs from 2017-03 to 2017-04, as well as the average revocation rate that results from migrating to the spot VM with the lowest spot-to-on-demand ratio (and the lowest revocation risk) as spot prices fluctuate. As the table shows, hopping to the spot VM with the lowest instantaneous revocation risk

Spot Market	Revocations (per day)
c3.8xlarge.vpc.us-east-1d	15.4
g2.2xlarge.us-east-1d	12.1
r3.xlarge.us-east-1d	9.4
g2.8xlarge.vpc.us-east-1d	9.0
r3.8xlarge.us-east-1d	7.9
c3.2xlarge.vpc.us-east-1d	6.9
g2.2xlarge.vpc.us-east-1d	6.0
g2.2xlarge.us-east-1b	5.6
g2.2xlarge.us-east-1a	5.1
r3.4xlarge.us-east-1a	4.5
HotSpot VM	**2.3**

Table 4.1: Migrating to the spot VM with the lowest spot-to-on-demand ratio has >0.5× revocations than other servers.

results in a revocation rate nearly 0.5× that of any single VM. Thus, VM hopping is a useful mechanism for managing revocation risk if it ever increases.

4.1.2 Impact on Performance

As with revocation risk, HotSpot's migration policy does not consider performance when determining when and where to migrate. Since HotSpot migrates based on cost-efficiency, which is a function of resource utilization, it favors VMs that do not waste resources. Given this, HotSpot may select a host VM that is under-provisioned and degrades an application's performance. During our initial analysis of spot price data in `us-east-1` (from 2016-08 to 2016-09), we observed that there was a volume discount: higher-capacity VMs were cheaper on average than lower-capacity ones. As a result, the most cost-efficient VM on average aligned with a high-capacity VM that prevented performance throttling. However, our more recent analysis in Figure 4.7 across multiple regions shows that this volume discount no longer applies. This change reflects how markets conditions can alter HotSpot's performance.

The figure shows the average normalized price per unit of resource for spot VMs in the `m4` family from 2017-03 to 2017-04, where the error bars represent the maximum and minimum price across each region's AZs. The graph indicates there is no consistent relationship between VM capacity and normalized cost: neither high-capacity nor low-capacity VMs are consistently cheaper per unit of resource. As a result, migrating solely based on

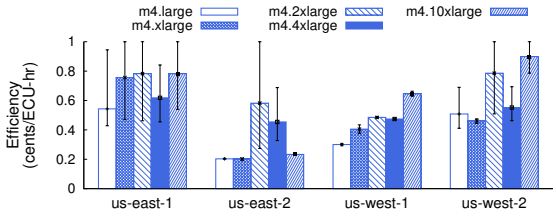


Figure 4.7: The average normalized cost per ECU for the `m4` family of spot VMs across multiple regions. The error bars represent the maximum and minimum cost across all AZs.

cost-efficiency should not favor either low-capacity VMs, which degrade an application's performance, or high-capacity VMs, which may improve its performance. However, as we discuss in §4.3, HotSpot's migration policy takes into account multiple other factors when making migration decisions that favor higher-capacity VMs, assuming equal cost-efficiency. Thus, while HotSpot optimizes for cost-efficiency, in practice, it can also improve application performance.

Summary. Our data analysis indicates that hopping to the most cost-efficient spot VMs can reduce cost up to 15-65% on average relative to not hopping. While HotSpot considers only cost-efficiency in selecting its host VM, based on our analysis, this VM also tends to have a low revocation risk and does not degrade performance on average. As a result, HotSpot does not explicitly consider revocation risk or performance in its migration policy.

4.2 Active Trading by Server Hopping

To enable active trading of cloud servers, we present HotSpot, a resource container that automatically "hops" spot VMs—by selecting and self-migrating to new VMs—as spot prices change. Our key insight is that applications can proactively and transparently migrate to spot VMs that currently offer the lowest cost. An important design goal of HotSpot is simplicity: to use it, applications need only select and run a HotSpot-enabled VM image that requires little configuration. Applications then execute inside a resource container, while HotSpot's systems-level monitoring and migration functions run transparently in the host VM. Thus, HotSpot is self-contained, requiring no application modifications or external infrastructure, as its functions execute within its current host VM.

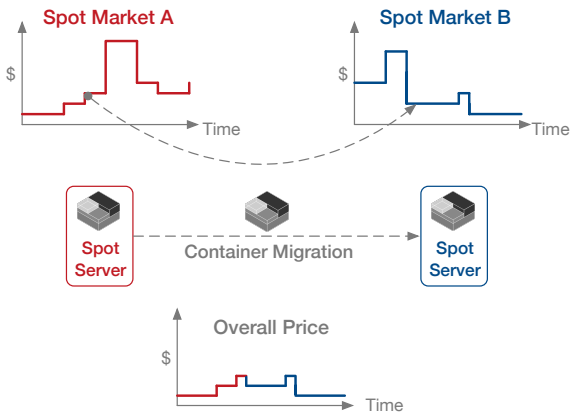


Figure 4.8: When the price of HotSpot's host VM rises, it self-migrates or "hops" to another VM with a lower cost.

HotSpot is motivated by both the maturing of systems-level migration for virtualized cloud hosts, e.g., via resource containers [5, 11] or nested virtualization [20, 74, 91], and continuing advances in data center networking, which are reducing the overhead of migrating memory state and accessing remote disks. Figure 4.8 illustrates HotSpot's basic function: when a VM's price spikes, HotSpot migrates its container to another VM with a lower price to maintain a high *cost-efficiency*. We define cost-efficiency as the cost (in dollars) per unit of resource an application utilizes per unit time. As we discuss in §4.3, we define utilization based on a VM's average CPU utilization. Figure 4.9 depicts HotSpot's basic control loop, which i) monitors real-time spot prices across the market and application resource usage, ii) uses the information to determine when and where to migrate based on its migration policy, and then iii) executes the migration. These functions are hidden from applications, which run within an isolated virtualized environment—a resource container in our prototype—capable of systems-level migration.

4.3 HotSpot Design

4.3.1 Migration Policy

HotSpot's migration policy determines when and where to migrate its resource container based on current spot prices across all VMs within its AZ and its current resource usage. Since migrations across AZs and regions require significant additional overhead to migrate

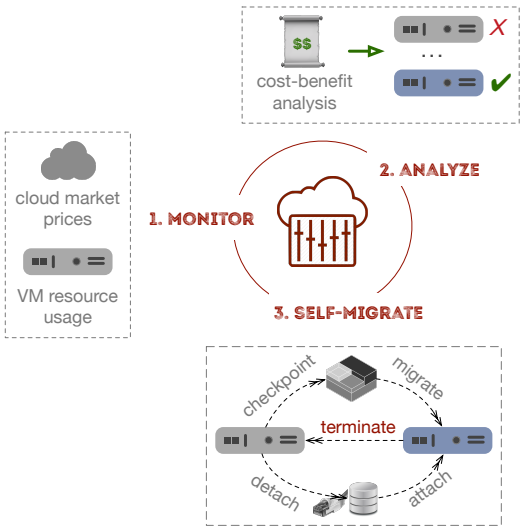


Figure 4.9: Depiction of HotSpot's basic control loop, which monitors spot prices and application resource usage, determines when and where to self-migrate based on its migration policy, and then executes the migration.

disk state, we limit HotSpot to migrating within an AZ, where it can migrate disk state by re-mounting a remote disk volume, e.g., via EC2's Elastic Block Store (EBS), with little overhead. We discuss policies for "global" migration across AZs and regions in [71]. As mentioned in §4.1, HotSpot's migration policy optimizes for cost-efficiency, or the cost per unit of resources an application utilizes per unit time. However, quantifying a VM's utilization is complex, since it includes many resources, e.g., computation, memory, I/O, etc. Our policy quantifies cost-efficiency based on processor utilization, and only uses an application's memory footprint to eliminate candidate VMs. Specifically, since memory significantly degrades performance when constrained, our migration policy includes a rule to never migrate to a VM with less memory than the container's memory footprint. Since platforms often price network and disk I/O capacity separately from VMs, our policy does not consider them, although we could apply a similar elimination rule as for memory.

At each time t, HotSpot's migration policy computes the expected cost-efficiency of every type of spot and on-demand VM within an AZ in units of \$/ECU-utilized/hour. To do so, HotSpot estimates the expected utilization on every potential VM i based on the utilization of its current VM. HotSpot makes this estimate by considering two separate cases based on the current utilization.

Low Utilization. If the ECU utilization on the current VM c is below an upper threshold (near 100%), we approximate ECU utilization u_i on VM i by proportionately scaling the ECU utilization of c across the number of ECUs offered by i. In this case, since c is not fully utilized, new VMs that have more ECUs than c should be even less utilized, while new VMs that have less ECUs than c should have a proportionate increase in utilization up to 100%.

High Utilization. Alternatively, if the current VM c's utilization is above the upper threshold, new VMs that have fewer ECUs than c should also have a utilization near 100%. However, if new VMs have more ECUs than c, we do not know how many additional ECUs the application is capable of consuming. In this case, our policy makes the aggressive assumption that the application can saturate any number of ECUs. This assumption encourages HotSpot to try out higher-capacity VMs, which can improve application performance.

Given the two cases above, we estimate the utilization u_i of a new VM i based on the utilization of the current VM c using the equation below. The equation includes a variable ϵ, which sets our upper threshold and dictates how aggressive the policy migrates to higher-capacity VMs. Note that the variables that are a function of t represent values that are dynamic and change over time.

$$u_i(t) = \begin{cases} min(\frac{u_c(t)}{ECU_i/ECU_c}, 1) & \text{if } u_c(t) < 1 - \epsilon \\ 1 & \text{if } u_c(t) > 1 - \epsilon \end{cases}$$

Since HotSpot is application-agnostic, we also make the simplifying assumption that applications are able to utilize any number of cores (or hardware threads) on a new VM, specified as vCPUs in EC2. Note that, if our assumptions wrong, then the migration policy will self-correct, as the actual utilization and cost-efficiency will be less than the expected value, which, if low enough, will trigger another migration. Extending the policy to model container performance and infer its degree of parallelism is future work. Given the estimated utilization u_i on each new VM i, HotSpot computes its cost-efficiency as below, where $p_i(t)$ is the VM's current spot price.

$$e_i(t) = \frac{p_i(t)}{ECU_i \times u_i(t)} \tag{4.1}$$

Next, for each potential host VM i that is more cost-efficient than our current host VM, HotSpot performs a cost-benefit analysis to determine when and where to hop VMs.

4.3.2 Cost-Benefit Analysis

Our analysis in §4.1 assumes an ideal migration policy that is able to migrate with no overhead to the VM with the highest instantaneous cost-efficiency at each time t. Of course, in practice, migrations can incur a substantial overhead. This overhead comes in two different forms: a *performance overhead* that stems from the downtime (or performance degradation) caused by a migration, and a *cost overhead* that stems from paying for two VMs during the migration and vacating a VM early (before the end of its billing period). The combined cost of these overheads represents a migration's *transaction cost*. HotSpot's migration policy only hops VMs if it expects the savings to outweigh the transaction cost.

Transaction Cost. From above, the two things HotSpot requires to estimate the transaction cost are i) the time remaining T_r in the current billing interval and ii) the time required T_m to migrate the container. HotSpot tracks T_r, which is computed from the running time on the current VM and the billing interval, which is well-known. The migration time T_m is a function of the container's memory footprint and network bandwidth. As we discuss in §4.4, HotSpot uses a memory-to-memory stop-and-copy migration that copies the container's memory state from the memory of the source VM to the memory of the destination VM without saving it to stable storage. Thus, we estimate T_m based on the container's memory footprint M at time t of the migration divided by the available bandwidth B, plus a constant time overhead O to interact with EC2's APIs to configure the new VM, so $T_m(t) = M(t)/B + O$. We evaluate the migration time and the constant time overheads in §4.4.

During the migration, HotSpot must pay for both the source and destination VM. In addition, since HotSpot uses a stop-and-copy migration, it does no work on either VM during the migration time, i.e., both VMs have 0% application utilization. Thus, we compute the transaction cost to migrate to a new VM i at time t as below.

$$C_i(t) = p_i(t) \times T_m(t) + p_c(t) \times max(T_r, T_m) \tag{4.2}$$

Here, $p_i(t)$ and $p_c(t)$ are the current spot price of the new host VM i and current host VM c, respectively. This equation represents the cost of the source and destination VM over the migration time, since neither is doing useful work, plus the cost of the remaining unused time HotSpot must pay in the source VM's billing interval. Note that the source VM's spot price $p_c(t)$ is fixed, as EC2 charges for the spot price at the beginning of each billing interval. Since migration times are short, as shown in §4.4, we also assume the destination's spot price is fixed during the migration.

Expected Savings

HotSpot estimates its expected cost savings $S_i(t)$ from migrating to new host VM i at time t as the difference between its cost-efficiency $e_c(t)$ on its current VM c and its expected cost-efficiency $e_i(t)$ on new VM i multiplied by both the time T_i it expects to spend on i and the number of ECUs it uses.

$$S_i(t) = (e_c(t) - e_i(t)) \times (u_i(t) \times ECU_i) \times T_i \tag{4.3}$$

The expected net cost-benefit N_i from hopping to VM i is then $S_i(t) - C_i(t)$: HotSpot only hops VMs if this value is positive. Among all hosts i where $e_i(t) < e_c(t)$ and $N_i > 0$, the migration policy selects the one that maximizes the net benefit N_i. Note that the expected savings is, in part, a function of the number of ECUs on the new VM i, while the transaction cost above is independent of VM capacity. This favors hopping to higher-capacity hosts, which improve performance, assuming the container can utilize their resources, as higher-capacity hosts are able to "pay back" their transaction costs faster than lower-capacity hosts.

The key unknown variable in Equation 4.3 is T_i, or the time HotSpot expects to spend on a new host VM i. HotSpot must have an accurate estimate of T_i to determine whether the expected savings exceed the transaction costs. However, T_i is challenging to estimate, since it depends on the future value of numerous other variables, including the remaining lifetime of the container, the application's future resource usage, and the relative spot prices of every VM. A significant phase change in any of these variables can decrease the time T_i

that HotSpot spends on a new VM, reducing the expected savings and altering the cost-benefit analysis.

Since HotSpot requires estimates of each of these variables to estimate T_i, its migration policy is a heuristic. Our current policy makes simple assumptions to infer these variables, and leaves more complex heuristics to future work. First, we assume containers are long-lived, and thus do not terminate before paying off their transaction costs. We also assume an application's utilization is constant in the near term. Given these assumptions, HotSpot estimates T_i as the maximum of the billing interval and the average Time-to-Change (TTC) of the lowest cost VM. The TTC is the average time until we expect to hop VMs in the ideal case. However, if the TTC is shorter than the billing interval, HotSpot is unlikely to migrate until the end of the billing interval to prevent a large double payment.

Finally, HotSpot may migrate to a more cost-efficient VM, incurring transaction costs, only to find an even more cost-efficient VM becomes available before it has recouped the previous transaction costs. Thus, our policy also reduces the expected savings $S_i(t)$ in Equation 4.3 from hopping to a new host VM i to account for any "unpaid" transaction costs not recouped from previous migrations. As a result, HotSpot only hops to a new VM if its price is low enough to yield a positive net benefit even after paying off accumulated transaction costs from previous migrations. This choice is conservative in rate-limiting migrations and preventing accumulating large transaction costs that increase cost. HotSpot also enables users to specify a minimum time between migrations to rate-limit them.

Bidding. When requesting a new spot VM, HotSpot must place a bid. Since EC2's current spot market requires applications to pay the spot price and *not* their bid price, HotSpot does not require a sophisticated bidding strategy, as it automatically migrates to a new VM if the spot price rises. In contrast, the bidding policy is more important in prior work [99, 38, 78], which commits to spot VMs until they are revoked, as the greater the bid the lower the probability of revocation and the higher the potential cost. Thus, HotSpot adopts a simple bidding strategy: it always bids the maximum price, which is 10× the on-demand price in EC2. Since HotSpot also includes on-demand VMs in its cost-benefit analysis, it will never pay near its bid price, since it will always migrate to an on-demand VM if the spot

price of all spot VMs ever rises above their corresponding on-demand price. Note that the specific bidding policy is orthogonal to HotSpot's design. For example, if EC2 were to change the spot market rules such that applications paid their bid price, instead of the spot price, HotSpot could support a different bidding policy without altering any of its other functions.

4.3.3 Qualitative Discussion

In reducing price risk, HotSpot addresses problems with current fault-tolerance-based approaches that manage revocation risk. In particular, the primary problem with fault-tolerance-based approaches is that applications configure their fault-tolerance mechanism based on the historical TTR from traces of past spot prices. However, as discussed in §4.1, spot prices often experience phases of stability and volatility. Thus, if applications experience a phase change in the prices, they may incorrectly configure their fault-tolerance mechanism, e.g., by checkpointing too much or too little, causing them to "pay" non-optimal premiums. The benefits of employing fault-tolerance are also probabilistic, and must be amortized across long time periods or a large number of applications. As a result, any individual application may end up paying high premiums without ever receiving a payout, e.g., if a revocation never occurs. In comparison, HotSpot has the following advantages.

- **More Deterministic.** Since migration decisions are largely based on *current* cost, risk, and performance information, they are more deterministic than decisions on how to configure fault-tolerance mechanisms, which are based on probabilistic expectations of *future* cost, risk, and resource usage.
- **Lower Overhead.** Automated VM hopping does not incur fault-tolerance overhead based on probabilistic information. While each migration incurs an overhead, it serves as a natural checkpoint that applications only "pay" if the expected savings exceed the costs. HotSpot often migrates more frequently than the optimal checkpointing interval in fault-tolerance-based approaches, which obviates the need for these approaches.
- **Lower Risk.** VM hopping reduces price *and* revocation risk, since low-cost VMs also have a low revocation risk.

HotSpot's design has some limitations. In particular, to remain application-agnostic, our migration policy makes a number of simplifying assumptions in §4.3 that do not apply to all applications. Designing migration policies for specific applications that limit these assumptions is future work. In addition, to simplify our design, HotSpot VMs are self-contained and do not coordinate their migration decisions with other HotSpot VMs. As a result, HotSpot's local migration decisions may not be globally optimal for distributed applications with complex dependencies. Applying HotSpot to distributed applications by coordinating their migration is future work.

Finally, HotSpot uses stop-and-copy migrations that cause application downtime. We do not consider live migration because containers do not yet support it. While live migration decreases application downtime, it still causes some performance degradation during the migration and increases the total migration time T_m, and thus also incurs a transaction cost. However, live migration requires a different transaction cost model. We plan to incorporate live migration into HotSpot once it becomes reliable for containers.

4.4 Implementation

We implement HotSpot's controller daemon in `Python`, including the migration policy from §4.3 and the monitoring and migration functions described below. Specifically, HotSpot uses EC2's Python binding `Boto3`, and Linux Container (LXC) 2.0.7's Python API.

4.4.1 Spot Price and Resource Monitoring

Our prototype operates within an AZ and monitors real-time spot prices, which continuously vary, from each spot VM. While the number of VMs varies across AZs, the largest AZs in `us-east-1` have 172 types of spot and on-demand VMs with distinct prices. To monitor spot prices, HotSpot's controller polls EC2's REST API and keeps a window of recent prices in memory, while maintaining a log of historical prices on disk. Likewise, HotSpot also monitors container resource usage via `lxc-info`, including its CPU, memory, bandwidth, and block I/O usage. HotSpot also caches a window of recent processor utilization and memory footprint readings in memory, and stores the historical usage data on disk. Despite the large number of VMs, the performance overhead of monitoring is not

Operation	Min (sec)	Mean (sec)	Max (sec)
Price and Resource Monitoring	<1	<1	<1
Acquire On-demand VM	16	28	31
Acquire Spot VM	31	67	167
Transferring Disk & Network	18	28	48
Terminate Source VM	31	44	46
Total	∼64-80	∼101-140	∼126-262

Table 4.2: Migration latencies for EC2 API operations.

significant. In addition to spot prices, HotSpot also maintains a table of VM types and their resource allotments, including their memory and number of ECUs and vCPUs, required for computing cost-efficiency. This table also stores each VM's on-demand price.

4.4.2 Host Migration and Handoff

When triggered by the migration policy, HotSpot's controller must request and migrate to a new EC2 host. The controller performs a sequenced handoff to complete a migration to a new host VM by first requesting the VM via EC2's REST API, waiting until it is running, and then transmitting the container's memory state to it. The source controller must also perform a hand-off to a new controller running on the destination VM. This hand-off requires transferring the metadata necessary to request and configure new host VMs via EC2's REST API, including the credentials necessary to access the API, such as the Secret Key. The source controller must also transfer container configuration meta-data, including the IP address and name of the root EBS volume, which the destination VM must configure via EC2's REST API before re-activating the container. Once the source controller transfers this information, the destination controller terminates the source VM.

We ran a series of microbenchmarks to quantify the overhead of the REST API operations associated with migration, as listed in Table 4.2. Since there is some variance in the latency, we report the mean, maximum, and minimum latency over 25 experiments. The table shows that price and resource monitoring overhead (from monitoring 402 spot prices) is negligible, even at per-second resolution. While the latency to acquire an on-demand or spot VM is between 15s and 167s, these operations do not result in application downtime,

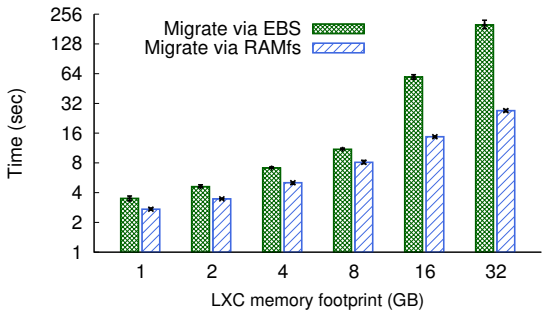


Figure 4.10: The time to transfer a container's memory state and restore it as a function of its memory footprint. The graph shows the average from four trials with error bars representing the minimum and maximum transfer time.

as the source VM continues to run during this period. Likewise, the 30-46s required to terminate the source VM also does not incur downtime. Application downtime is a function of the time to i) disconnect and reconnect the container's disk and network interfaces, and ii) physically transfer the container's memory state.

As the table shows, the time to disconnect/reconnect the disk and network interfaces ranges from 17s to 48s. Figure 4.10 then shows the average time to transfer container memory state as the application memory footprint increases. The EBS approach checkpoints container memory state to a remote disk on the source VM and then reads it from the remote disk on the destination VM, while RAMfs signifies an approach that uses a direct memory-to-memory network transfer of container memory state. As the graph shows, the memory-to-memory transfer enables migrations of up to 32GB in ∼30s (using EC2's 10Gbps interfaces), while transfers of 32GB over EBS take ∼200s (using I/O-optimized EBS drives).

Note that, while the memory-to-memory transfers are near linear in the amount of data transferred, the EBS approach appears super-linear. While we do not know the specific reason for this super-linearity, it could be due to caching effects in EBS. For example, the time difference between the EBS and memory-to-memory transfers is small for low memory footprints and only increases as the size of the memory footprint increases. This might indicate the presence of an EBS cache that can accommodate low memory footprints. In any case, HotSpot's prototype uses direct memory-to-memory transfers, resulting in application downtimes ranging from 20s to 80s, depending on the size of the memory state. Of course,

even these minimal downtimes could be eliminated with native support for live migration, which GCE already provides.

HotSpot uses EC2's Virtual Private Cloud (VPC) to assign its container a separate IP address from the controller daemon, which uses the default public IP address allocated by EC2. The controller selects an available private IP address from the VPC's address space via EC2's REST API and configures the container with this IP address, such that all traffic to the VPC IP address is forwarded to the container. When migrating, the controller transfers this IP address to the new VM by detaching it from the source VM and re-attaching it to the destination VM. Thus, when restarted, the migrated container always retains the same VPC-allocated IP address.

4.5 Evaluation

We evaluate HotSpot at small scales using a prototype on EC2, and at large scales over a long period in simulation using a production Google workload trace [61] and publicly-available EC2 spot price traces. Our simulator, also implemented in Python, executes the same migration policy as our prototype, but replaces its real-time monitoring with functions that read spot price and resource usage data from traces. The traces include each application's processor utilization and memory footprint over time. Instead of migrations, the simulator inserts a downtime derived from our microbenchmarks based on an application's current memory footprint.

We compare HotSpot with three other approaches, which select the single optimal i) on-demand VM, ii) spot VM, and iii) spot VM plus optimal fault-tolerance mechanism to run the application. The first case represents current practice; the second case is akin to EC2's SpotFleet tool, which automates bidding for spot instances (at the on-demand price by default), and the third case uses SpotOn [78], which is a representative fault-tolerance-based approach. Note that SpotOn only switches VMs on a revocation. Our evaluation then compares three metrics—cost, performance, and revocation risk—for each approach.

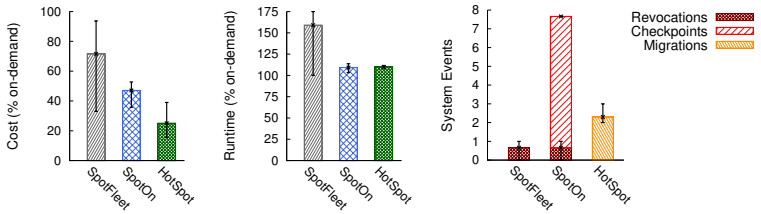


Figure 4.11: Comparison of cost (left), run time (middle), and revocation-related events (right) when using on-demand VMs, spot VMs without fault-tolerance (SpotFleet), spot VMs with checkpointing (SpotOn), and HotSpot when running our baseline job on our HotSpot prototype. The error bars represent the maximum and minimum of each metric across three trials.

4.5.1 Prototype Results

We intend these experiments to exercise our prototype and isolate key factors, such as price characteristics and application resource usage, that influence HotSpot's relative cost, performance, and revocation risk, and not to quantify its benefits in practice. These experiments do not reflect the possible values of all time-varying variables, such as spot prices and resource usage, that influence HotSpot, as there are too many variables to emulate in a controlled setting. To enable control of application resource usage, we use a job emulator that generates a fixed, predictable CPU load and memory footprint. We use this emulator to create a baseline job that takes 30 minutes to execute on a `m4.4xlarge` VM with a steady memory footprint of 8GB and processor utilization of 50%. We run this job in an LXC container on EC2 for each approach above, and perform all HotSpot functions, i.e., acquiring, migrating, checkpointing, terminating, etc., on real EC2 VMs.

To enable price control, we also generate synthetic spot price traces that reflect important characteristics in the real market. We define synthetic spot prices for five separate VMs (each of type `m4.4xlarge`) that vary based on a sinusoidal price function with a period of one hour and peak/trough values equal to \$0.8/hour, and \$0.08/hour, respectively. In this case, the maximum spot price of the VMs is equal to the on-demand price of a `m4.4xlarge` VM, which costs \$0.8/hour in `us-west-1`. We use synthetic spot prices instead of EC2 spot price traces, since synthetic prices are defined by a well-known function that we can alter to examine the effect of changing price volatility on cost and performance.

We initialize each experiment by setting the start time of each price function to a random offset within its period. Thus, on average, the TTC of the lowest-cost VM is 12 minutes, which is 5.5× faster than we observed in reality. Thus, our migration policy sets the TTC to 12 minutes when performing its cost-benefit analysis. We also set an emulated bid price in the experiment equal to the maximum price, which results in an average of one revocation per hour. While this revocation rate is high relative to real revocation rates, we select it so our emulated job has the potential to experience revocations. This enables us to illustrate the impact of revocations on the relative cost and performance of each approach, although the magnitude of our results is not representative of real spot VMs.

We construct synthetic price traces to exhibit the correlation between price level and revocation risk in the real market, where a low-cost server is less likely to be revoked. While our experiments isolate some key factors in HotSpot's design, they do not isolate all of them. In particular, we set the billing interval to one minute, rather than one hour, so the experiments do not include the increase in cost from vacating a VM early. We use a short billing interval to enable us to isolate other factors when running half-hour jobs. Thus, in these experiments, HotSpot's transaction cost derives solely from its migration overhead. Note that this overhead is higher, as a fraction of a job's running time, the shorter the job. The migration overhead ranges from $25 - 56$s, or $0.7 - 1.6\%$ of the running time for our baseline half-hour jobs. We evaluate real-world performance over longer periods with an hour-long billing interval in simulation.

Baseline Experiment

Figure 4.11 compares the cost, running time, and revocation risk of using the on-demand, SpotFleet, SpotOn, and HotSpot approaches. Note that the y-axis of the first two graphs is normalized relative to the metric's value using on-demand VMs. For each approach, we execute three trials with a different set of randomly chosen offsets for each spot VM in the synthetic price function, where the error bars reflect the maximum and minimum values. As expected, the cost (left) decreases between 30-75% when we switch from using on-demand VMs to any approach that uses spot VMs, since spot VMs are cheaper on average than on-demand VMs. Thus, even when the SpotFleet approach, which does not use fault-tolerance

or migration, experiences a revocation and has to restart the job from the beginning, it remains cheaper than using an on-demand VM. The cost further decreases for SpotOn (by 34%) and HotSpot (by 65%) compared to SpotFleet, since SpotOn benefits from periodic checkpoints that limit the work lost after a revocation, while HotSpot does not experience a revocation. Finally, we see that HotSpot's cost is 45% less than SpotOn, since HotSpot always migrates to the lowest cost VM, while SpotOn remains on each VM until it is revoked and thus experiences high price periods.

Figure 4.11(middle) shows that the job's running time increases relative to using an on-demand VM. This occurs because our experiment, in contrast to EC2's actual spot market, does not include VMs with multiple capacities at different normalized prices. As a result, there is no opportunity to improve on the performance of an on-demand VM by selecting a higher-capacity spot VM. In the figure, the average running time of SpotFleet is worse than both SpotOn and HotSpot. For SpotFleet, the decrease in running time derives from the large overhead of having to restart the job from the beginning after each revocation, while, for SpotOn and HotSpot, the decrease derives primarily from the smaller overhead of checkpointing and migration, respectively. As shown in Figure 4.11(right), while HotSpot migrates an average of $\sim$2 times, its performance overhead does not exceed either SpotFleet or SpotOn. Unlike SpotFleet, HotSpot experiences no revocations and never has to recompute lost work. The figure also shows that HotSpot executes fewer migrations than SpotOn executes checkpoints. However, each checkpoint only has an overhead of 8s, while the migration overhead of the 8GB memory footprint ranges from 25-56s. These overheads balance out such that SpotOn and HotSpot maintain a similar performance.

Changing Memory Footprint

We next evaluate how changes in an application's memory footprint, which dictate the transaction cost of migration, affect cost and running time relative to our baseline 8GB memory footprint. Figure 4.12 shows the results where the error bars indicate the maximum and minimum value across three trials. Using the same configuration as our baseline experiment, we vary the memory footprint from 8GB to 64GB. As before, the cost (left) of using an on-demand VM is high relative to the other approaches in nearly all cases.

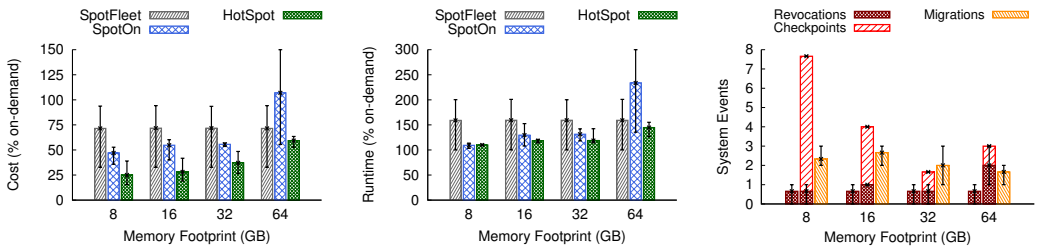


Figure 4.12: Comparison of cost (left), run time (middle), and system events (right) when using on-demand VMs, spot VMs without fault-tolerance (SpotFleet), spot VMs with checkpointing (SpotOn), and HotSpot as the memory footprint varies. The error bars represent the maximum and minimum of each metric across three trials.

In this experiment, note that the cost and performance of using an on-demand VM is the same for any size memory footprint. Only SpotOn with a 64GB memory footprint costs slightly more than using an on-demand VM due to its high periodic checkpointing overhead. SpotOn's cost decreases relative to using on-demand VMs as the memory footprint and the resulting checkpointing overhead decrease. SpotFleet maintains the same cost across all memory footprints, since it does not perform any checkpoints or migrations. HotSpot's cost is consistently lower than both SpotFleet and SpotOn, since it always migrates to the lowest-cost server. As the memory footprint increases, the overhead of these migrations also increase, which reduces the cost savings relative to using on-demand VMs. However, even for a 64GB memory footprint, HotSpot's cost is 40% less than the on-demand VM.

Figure 4.12(middle) shows that HotSpot's running time is also consistently equal to or less than the running time of SpotFleet and SpotOn as the memory footprint varies. SpotFleet's running time remains constant since it does not depend on the memory footprint, and is greater than HotSpot's running time even for a 64GB memory footprint. SpotOn's running time is nearly equal to HotSpot's running time for the 8GB memory footprint, as discussed in the baseline case. However, SpotOn's running time increases more rapidly as the memory footprint increase compared to HotSpot's running time. For a 64GB memory footprint, HotSpot's running time is nearly 40% lower than SpotOn's running time.

The differences in running time stem from the overheads of checkpointing, migrating, and recomputing lost work after each revocation. Figure 4.12(right) plots the number of revocations, checkpoints, and migrations, where the order of the bars for SpotFleet, SpotOn,

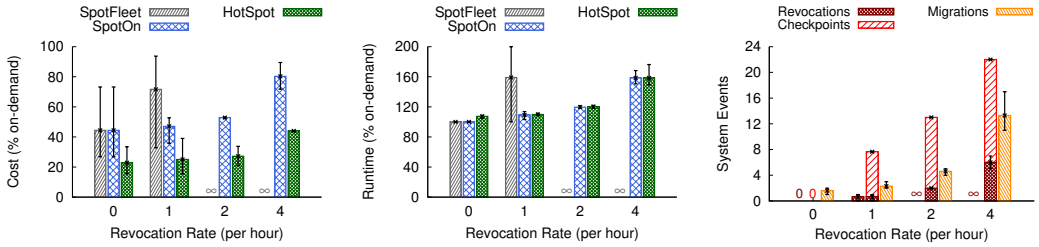


Figure 4.13: Comparison of cost (left), run time (middle), and system events (right) when using on-demand VMs, spot VMs without fault-tolerance (SpotFleet), spot VMs with checkpointing (SpotOn), and HotSpot as the spot price volatility changes. The error bars represent the maximum and minimum of each metric across three trials. Once the revocation rate increases to two per hour SpotFleet never finishes, so we label ∞ for its cost, running time, and system events.

and HotSpot are the same as the adjacent figures. We do not include the on-demand approach, since it does not experience any revocation-related events. The graph shows that HotSpot never experiences a revocation, since it always migrates to the lowest-cost server with a low revocation risk. In contrast, SpotFleet and SpotOn experience at least one revocation on average. For low memory footprints, SpotOn checkpoints frequently, since the overhead of checkpointing is low. However, for large memory footprints, the checkpointing overhead is high so SpotOn only checkpoints once. Thus, the revocations for SpotOn at large memory footprints incur a large recomputation overhead that increases its running time.

Changing Spot Price Volatility

We also vary the frequency of revocations relative to our baseline to illustrate the impact of market volatility on cost and running time. In this case, we vary the revocation rate by changing the periodicity of our sinusoidal price function. Figure 4.13 plots the resulting revocation rate (in revocations per hour) on the x-axis. Again, all spot-based approaches in Figure 4.13(left) cost less on average than using an on-demand VM. The error bars represent the maximum and minimum cost across three trials. We also see that HotSpot has a lower cost than the other approaches across all revocation rates. As the spot price becomes more volatile, HotSpot's cost advantage improves relative to both SpotFleet and SpotOn due to the overhead they experience from both checkpointing and recomputing lost work after a revocation.

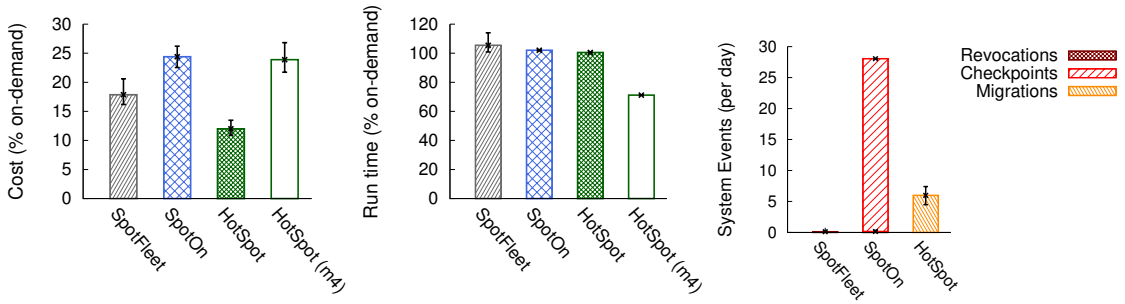


Figure 4.14: Comparison of cost (left), run time (middle), and system events (right) when using on-demand VMs, spot VMs without fault-tolerance (SpotFleet), spot VMs with checkpointing (SpotOn), and HotSpot from simulating jobs from a production trace on spot VMs based on EC2 spot price traces. The error bars represent the maximum and minimum over five trials.

Figure 4.13(middle) plots the job running time as the revocation rate changes. We see that SpotFleet's performance is highly sensitive to the revocation rate, since each revocation incurs a large recomputation overhead. Note that once the revocation rate increases to two per hour SpotFleet never finishes, so we label ∞ for its cost, running time, and system events. By comparison, both SpotOn and HotSpot have similar running times across all revocation rates. While SpotOn experiences some revocations at higher revocation rates, as shown in Figure 4.13(right), the performance impact of these revocations is limited by its periodic checkpoints. As before, the order of the bars for SpotFleet, SpotOn, and HotSpot in Figure 4.13(right) are the same as the adjacent figures.

Since the checkpointing overhead is low for our 8GB baseline memory footprint, SpotOn is able to checkpoint many times over the length of the job, e.g., one checkpoint every 6 minutes in the baseline case of one revocation per hour. This frequent checkpointing limits the performance impact of revocations. In comparison, the lowest-cost VM changes more frequently as price volatility increases, which requires HotSpot to migrate more frequently. While each migration incurs more overhead than each checkpoint, HotSpot experiences no revocations and thus incurs no recomputation overhead. As in Figure 4.11's baseline, the checkpointing and migration overheads of SpotOn and HotSpot, respectively, balance out such that their performance is similar across all volatilities.

4.5.2 Simulation Results

Our simulator uses job traces from a production Google cluster [61], and the same EC2 spot price traces described in §4.1. The Google cluster traces report each job's memory footprint and normalized CPU utilization every five minutes. Since the Google cluster trace normalizes the server capacity between zero and one, it does not contain the actual CPU capacity of the servers. As a result, we re-normalize the server capacity to between 6.5 and 195 ECUs, which represent the minimum and maximum number of ECUs offered by EC2 across its VM types. For the on-demand approach, we select the VM type that has the closest number of ECUs to the ECUs in the trace. For the spot-based approaches, we initially select the lowest cost VM that matches each job's average ECU utilization. For SpotFleet, we also use this policy to select a new VM on each revocation. SpotOn and HotSpot use their own respective policies for selecting VMs after a revocation. We also assume each job's performance is a linear function of its resource utilization, i.e., ECUs utilized. That is, if a job is at 100% utilization of its normalized server capacity, and then migrates to a server with half the number of ECUs, we assume a 2× slowdown. If a job's utilization in the trace is 100% we do not know what its utilization would be on a higher capacity server. In this case, we make the pessimistic assumption that the job cannot scale up to use more ECUs than it did in the original trace. Our experiments assume EC2's standard one-hour billing interval, and sets the expected time to spend on a new server T_i equal to the MTTC of 1.1 hours based on our analysis in §4.1.

We select 1000 random jobs from the job trace and assume each job starts at a random time within the EC2 spot price trace. Since HotSpot containers are long-lived, we restrict our evaluation to jobs with durations greater than 24 hours to reduce the relative effect on the total cost from terminating the container before the end of its last billing interval. For shorter jobs, HotSpot should re-use the same container, rather than spawn a new container per job, to mitigate these end-of-lifetime effects. As before, we compare HotSpot's cost, performance, and revocation risk across the different approaches. Note that when the HotSpot container experiences a revocation, it restarts its job from the last migration time.

Figure 4.14(left) shows the average cost of each approach, where the error bars reflect the maximum and minimum values across five trials. As the graph shows, all of the spot-

based approaches have a significantly lower cost than using on-demand VMs. In addition, HotSpot is able to further lower the average cost compared to both SpotFleet and SpotOn. In addition, Figure 4.14(middle) shows that all the spot-based approaches slightly increase the average running time relative to using the on-demand VM specified in the job trace. In particular, HotSpot increases the running time compared to using on-demand VMs by <0.5% due its migration overhead, which is less than the increase caused by both SpotOn and SpotFleet. This increase in running time occurs because jobs in the original trace tend to run on over-provisioned servers, such that their average utilization is low. Thus, there is never an opportunity to improve performance by migrating to even higher capacity servers.

To demonstrate HotSpot's ability to improve both cost and performance, we plot an another scenario that normalizes HotSpot's cost and performance relative to running jobs on an `m4.large` on-demand VM with 6.5 ECUs, which we call HotSpot (m4). Since running on `m4.large` on-demand VMs periodically bottlenecks job performance, such that utilization reaches 100%, HotSpot is able to decrease the running time from migrating to higher capacity servers. However, HotSpot's cost advantage also decreases, as the `m4.large` is more cost-efficient on average than the high-capacity on-demand VMs originally selected by the jobs. Even so, in this case, HotSpot costs ∼25% of using `m4.large` on-demand VMs while decreasing the running time by ∼27%.

Finally, Figure 4.14(right) shows the revocation, checkpointing, and migration rate per day of each approach. HotSpot migrates on average ∼6 times per day, while SpotOn executes ∼28 checkpoints per day. While difficult to see in the graph, all approaches also experience revocations. Specifically, SpotFleet, SpotOn, and HotSpot have average revocation rates of 0.118, 0.152, and 0.02 revocations per day, respectively. Thus, HotSpot migration policy reduces the revocation rate compared to the other spot-based approaches.

4.6 Related Work

Many researchers have recognized the opportunity to reduce cost by leveraging spot VMs, however, there is little prior work similar to HotSpot that supports proactive migration as market conditions change. Instead, the focus of prior work has been on selecting the "optimal" spot VM based on an application's expected resource usage and future spot

prices [36, 38, 65, 67, 68, 78, 99, 94]. Much of the prior work focuses on reducing revocation risk by configuring fault-tolerance mechanisms, such as checkpointing and replication [36, 38, 65, 67, 68, 78]. In §4.5, we compare with SpotOn [78], which automatically selects and configures the optimal spot VM and fault-tolerance mechanism to execute a job. Prior work applies similar fault-tolerance-based approaches to specific distributed applications including Hadoop [99], Spark [65, 67], parameter servers [36], and matrix multiplication [38]. In contrast, HotSpot is transparent to the application and operates at the level of a single server (not a distributed system). Unlike HotSpot, prior work does not dynamically migrate to new VMs as spot prices change, but instead selects new VMs after a revocation [78, 65, 67] or at periodic intervals [36].

Since prior work often implicitly commits to running on a particular spot VM until a revocation occurs, the bidding strategy is important in balancing high costs due to an increase in the spot price (when bidding too high) and the performance penalty from increased revocations (when bidding too low). Thus, there is a significant body of work on spot VM bidding strategies [99, 48, 97, 76, 81, 50, 51, 92]. In contrast, the bidding strategy is not as important to HotSpot, as it proactively migrates as spot prices change. HotSpot never commits to a spot VM, and often migrates to a new VM before prices spike and cause revocations. Prior work also notes that EC2's spot market is artificial, since Amazon both operates the market and is the sole provider. For example, Ben-Yehuda et al. showed that before 2011, EC2 spot prices were not consistent with a constant minimal price auction [22, 21]. However, HotSpot's cost benefits do not rely on spot prices being driven by supply and demand, but only that there is a price difference between VMs.

Finally, similar to HotSpot's container migrations, Smart Spot instances migrate nested VMs between spot VMs in EC2 [42]. However, Smart Spot instances use a centralized scheduler that monitors a group of nested VMs and determines an optimal packing of them on spot VMs to reduce cost. Smart Spot Instances also do not consider revocation risk in their placement decisions, and suggest applications use fault-tolerance mechanisms, such as replication or checkpointing to mitigate this risk, which violates transparency.

4.7 Conclusion

This paper presents HotSpot, a container that automatically "hops" spot VMs—by selecting and self-migrating to new VMs—as spot prices change. We demonstrate the benefits of hopping VMs in EC2's spot market, and its effectiveness in reducing revocation risk and improving performance. We implement a prototype on EC2, and evaluate it using job traces from a production Google cluster. We compare HotSpot to using on-demand VMs and spot VMs (with and without fault-tolerance) in EC2, and show that it is able to lower cost and reduce the revocation rate without degrading performance.

CHAPTER 5

ELIMINATING UNCERTAINTY RISK BY INDEX-TRACKING

> "Prediction is very difficult, especially if it's about the future."
>
> Danish proverb

Applications that run on transient cloud servers suffer from *cost uncertainty* since spot prices are market-based. This inability to predict future spot prices affects both customers and applications: former, because they cannot plan their IT expenses in advance and latter, because the spot price determines the availability and performance characteristics of the transient servers. While researchers have proposed techniques for modeling and predicting prices of individual spot markets, their utility have been limited given the proliferation of spot markets, which now exceed 7600 on Amazon EC2.

In this work, we address the challenge of providing a reliable cost-estimate to flexible applications hosted on cloud spot markets. Our work is motivated by a simple but key market observation that spot markets are reliably predictable at aggregate levels (e.g., a datacenter, or a server family) than at individual server level. Towards quantifying this, we devise a novel index for cloud spot markets. We analyze EC2's global markets over 6-months to validate our hypothesis and to identify additional market insights. Building on these insights, we design an index-driven server hosting mechanism, and implement it on top of an open-source spot server framework. Evaluations on EC2 spot markets, via prototyping and simulation, show that our system not only matches the index-predicted cost-efficiency but does so while maintaining high availability.

5.1 Understanding Uncertainty

To model and predict the behavior of EC2 spot markets, most of the prior work analyzes the historical price traces from individual spot markets and then picks a queuing model that

best describes the analyzed dataset. While intuitive, this approach faces several challenges that limit its utility. First, EC2 spot markets are massive and complex: it comprises of ∼7600 independently priced server "listings" across 44 zones in 16 regions. By comparison, there are only around 6000 stocks listed across both the New York Stock Exchange and NASDAQ. While a number of researchers [92, 22, 99, 76, 66, 7, 41, 87] and startups [4, 46, 54] have proposed techniques for modeling and predicting spot market prices, there is no guarantee a one-size-fits-all model even exists as price characteristics are based on local supply-demand conditions. Second, EC2 spot markets exhibit price inversions and arbitrages such that a higher capacity server may be priced lower than a lower capacity one, or identical servers across zones may be priced differently based on the real-time supply-demand conditions. It is impossible for static prediction methods to account for such real-time inversions a priori. Finally, as application's resource usage changes over time, its choice of optimal server is unlikely to remain same throughout its lifetime.

In this work, we take first principles approach to observing and understanding the physical infrastructure-level realities in cloud datacenters.

5.1.1 On Diversity

In order to allow users to select the best fit server for their application, cloud providers sell a large number of server types that differ in their resource capabilities. However, these numerous server types are carved out of a limited number of physical machines. For example, EC2 offers 23 general-purpose server types (in $T2$, $M3$, $M4$, and $M5$ series) that are internally hosted on just 4 types of physical machines[1]. While the total number of physical machines in a given datacenter does not change drastically in short timeframes (like hours, days or even weeks), the number of servers of each type is likely to vary more frequently (as governed by the administrative policies, supply-demand dynamics of different contract types etc.). In other words, despite $m4.large$ and $m4.16xlarge$ being sold in separate spot markets, their availability, revocation characteristics and in turn, their prices are not likely to be independent. Prior works have largely ignored this physical reality in

[1]this inference directly follows from EC2's listing of dedicated hosts, a contract type where physical machines are rented instead of virtualized servers

assuming that price and revocation characteristics of different spot markets are independent and identically distributed (likely because the spot price traces under consideration did not explicitly reveal this).

Property 1: *Spot markets originating from the same physical machine family are not free from mutual interference.*

This has two implications to the users of idle capacity: First, it is not prudent to model the behavior of spot markets individually without regards to other markets that share the same underlying physical machines. Second, spot markets that do not share a common underlying machine type could be expected to be free of mutual interference (barring datacenter-wide emergency or maintenance events).

5.1.2 On Stability

Though individual market's spot prices vary drastically (up to 10×), presumably based on the supply-demand dynamics of the given server type, the overall idle capacity of datacenter paints a different picture. In the first public release of its kind, Microsoft researchers published [45, 28] detailed workload- and utilization characteristics of Azure datacenters in 2017. While it was known a priori [89] that significant portions of datacenter resources remain idle, Azure traces shed light on the exact nature of this idleness: the actual CPU utilization varies by the order of half the datacenter capacity but the users are not dynamically scaling their allocated servers to match the actual real-time utilization. Thus, Azure datacenters do not experience large swings in server allocations either at the customer level or at the datacenter level. The reported median volatility for server allocations is 6.3% hourly, 2.6% daily, 3.2% weekly (at the datacenter level). These findings also corroborate with observations from multiple Google datacenters [26], where researchers proposed that large chunks of idle capacity experience higher availability (>98.9%) over the window of 6 months.

Property 2: *For public cloud providers, datacenter's aggregate idle capacity tends to be stable.*

If compute was a fungible resource like oil and electricity, and all of datacenter's idle capacity was offered in a single marketplace to be consumed by perfectly flexible applica-

tions, then property-2 implies (i) that there would be a single unified clearing price like in the commodity spot markets, and (ii) that this clearing price would be largely stable and predictable (via the efficient market hypothesis [33] since the overall supply and demand are stable). Thus, flexible applications operating in this hypothetical setup would always pay the *fair market value* as well as have *predictable expenses*. Obviously, the assumptions on application's flexibility and compute's fungibility are not (yet) practical. However, in the next section, we explore a first order approximation that helps us benefit from this insight.

5.1.3 Market Indices

The market properties of §5.1 prompt us to analyze and model spot markets at aggregate levels instead of individual ones. Specifically, two granularities of aggregation are natural choices: (i) all the markets belonging to a given datacenter, and (ii) set of markets originating from a given server family (for e.g., all compute-optimized servers or all general-purpose servers housed in the datacenter). Towards collectively modeling a group of spot markets, we employ *market indices*.

A market index, in finance and economics, is a statistical measure of the value of a collection of items, and is useful in representing their collective movement in a time-series. For example, the Consumer Price Index (CPI) measures the changes in the price level of a pre-determined market basket of consumer goods purchased by typical households. Economists use the annual percentage change in CPI as a measure of inflation, which in turn guides the monetary policies on wages and taxes, interest rates, and cost of living adjustments. Similarly, stock market indices like the Dow Jones Industrial Average, the Standard and Poor's 500 and the NASDAQ Composite report the statistical measure of a prominent set of publicly traded stocks, and are considered as broad indicators of the country's economy.

Thus, financial companies that engage in sophisticated market strategies to manage their cost-risk-performance tradeoffs, rely on these indices to evaluate their positions as well as to make investment decisions. We argue that with compute-time turning into a core investment, technology-enabled companies would benefit from an index that succinctly describes the behavior of cloud spot markets. More importantly, when applied to spot

markets, market indices overcome several limitations of current approaches. First, index composition is based on directly-observed market properties, and not on indirect inferences from historical price traces. Second, by revealing the fair market value of idle compute capacity in real-time, the index provides a cost-benchmark for flexible applications. Lastly, it yields an open framework that can be easily extended and adapted to the needs of specific applications or market phenomena that the user is trying to model.

5.2 Market Index for the Cloud

The goal of the index is to succinctly describe the spot price characteristics of a group spot markets to reveal insights and to enable decision making. First, we describe the index construction methodology and then apply it on to EC2 spot markets to characterize its salient features.

5.2.1 Methodology

Our index construction methodology comprises of four components: characterization, composition, weighting, and consistency.

Characterization. Any compute server is characterized by the quartet of CPU, memory, storage and network. However, cloud servers are typically defined only by their CPU and memory since storage and networking are decoupled and sold separately. For EC2 servers, the compute capacity varies between 1 and 349 ECUs (EC2's measure of CPU capacity), and memory capacity varies between 0.5 to 1952 GiB. Thus, in order to normalize these two independent metrics, we compute their geometric mean for each server. Putting it all together, $\hat{P}_i(t)$ represents the normalized price of server i, with C_i number of ECUs, M_i GiB of RAM and a market price of P_i at time t.

$$\hat{P}_i(t) = \frac{P_i(t)}{\sqrt{C_i \cdot M_i}} \tag{5.1}$$

Composition. Composition determines the set of spot server markets that go into computing the index. While this is primarily driven by the market properties of §5.1, it could be further trimmed to accommodate application's resource constraints or expanded to study

broader market phenomena. For example, tuple `(us-east-1,16GB)` describes the set of spot markets in all six datacenters of `us-east-1` region that have a memory size of at least `16GB`.

Weighting. Index weighting determines the relative impact that each constituent item has on the final index value. The commonly used weighting mechanisms are (i) *equal weighting*, where each item contributes equally, (ii) *size-proportional weighting*, where each item contributes proportional to its size or capacity, and (iii) *attribute weighting*, where each item is weighted as per the score it gets for its attributes. In the real-world, DJIA uses equal weighting, S&P 500 uses market capitalization of stocks as their weight, and S&P 900 Growth uses growth prospect scores of stocks as their weight. For our purposes, since EC2 does not publish any details about the overall or available spot pool capacity, we simply employ equal weighting.

$$\mathbb{I}(t) = \frac{\sum_{i=1}^{N} \hat{P}_i(t)}{N} \tag{5.2}$$

Consistency. Consistency is the property of an index to absorb market changes i.e., addition or removal of elements, or alteration to the characteristics of elements, in such a way that the index values are comparable across those changes, and over time. Since we employ normalization and equal weighting, it is trivial to incorporate introduction of new spot markets, discontinuation of existing ones and even changes in resource capacities. However, in EC2 spot markets, spot servers may become temporarily unavailable i.e., no matter how high the users bid, EC2 will not make any new allocations for servers of that type. To communicate this situation, EC2 has set a bidding cap of 10× the equivalent on-demand price such that no user can outbid EC2, when it wishes to allocate certain type of spot servers for other purposes. Thus, in order to keep our indices consistent, we temporarily exclude all the 10× markets from index computation for as long as their prices remain at the cap level.

In summary, the index value at a given time represents the average price per unit of compute time (for the selected group of servers).

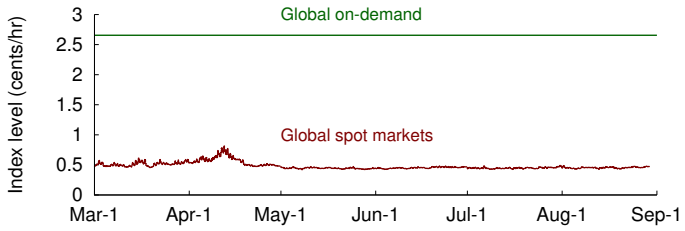


Figure 5.1: Index level for the global Linux spot markets (2406 across all 14 regions).

5.2.2 EC2 Spot Markets

The goal of applying the index on EC2 spot markets is twofold: first, to validate the market properties presented in §5.1, and second, to derive insights that can drive spot server selection. In the interest of space, we analyze indices only for Linux markets and only at select geographical locations.

First, we observe the spot markets at the highest possible aggregation i.e., global level. Figure 5.1 shows the index for all 2406 active Linux markets worldwide, with Y-axis plotting the index-level and X-axis indicating the day of the year. The graph shows that EC2's spot market is remarkably stable, in aggregate, with prices around 0.5 cents/hr, which equates to 80% discount over the global on-demand average.

Second, we observe the aggregate markets at the datacenter level with Figure 5.2 plotting the index-level for the three zones of `US-West-1`. We see that the characteristic peaky behavior of individual spot markets does not exist at the zone level. We also note that despite being located in the same geographical region, price variations across different zones are largely uncorrelated. This is because each zone is a separate datacenter with its own usage patterns and administrative overheads such that unused server capacity at a given time need not match across the datacenters.

Next, we decrease the granularity of aggregation to observe groups of servers belonging to three distinct set of families: compute-optimized, memory-optimized, and storage-optimized. Figure 5.3 shows these families for the `US-West-1a` zone. While there is increased volatility compared to the zone-level index, the values are still stable and predictable. Finally, we increase the levels of aggregation with Figure 5.4 demonstrating the corresponding

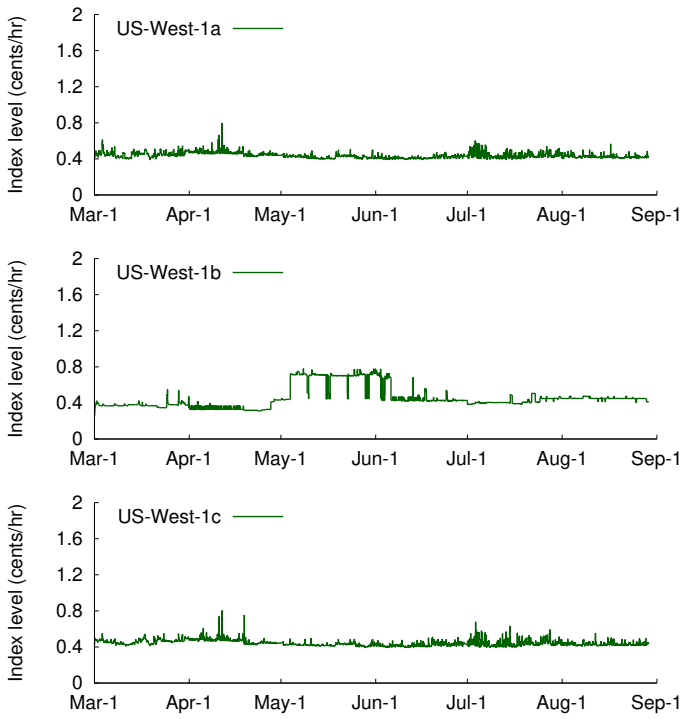


Figure 5.2: Indices for the three US-West-1 datacenters

regional index. As expected, it shows a higher level of stability and predictability compared to the zone-level indices, with price remaining at ∼16% of the on-demand price level.

Insight (on predictability): *Our analysis confirms that spot markets are remarkably stable and its prices are reliably predictable at aggregate levels. We see this behavior consistently at the global, regional, datacenter, and server-family levels.*

Given its generality, the index could be trivially extended to analyze the EC2 on-demand offerings. While on-demand prices are fixed within a region, they vary across regions as shown in Figure 5.5, which plots the index-level across all 14 of the EC2 regions. First off, we see that the price of compute varies substantially across regions with `SA-East-1` being 57% more expensive than `CA-Central-1` on average. Surprisingly, significant price differentials exist for geographically nearby regions as well. For example, index for `US-East-1` in Virginia is ∼20% higher than `US-East-2` in Ohio. While such disparities may be due to the regional economic factors including price of energy, availability of technical staff, and cli-

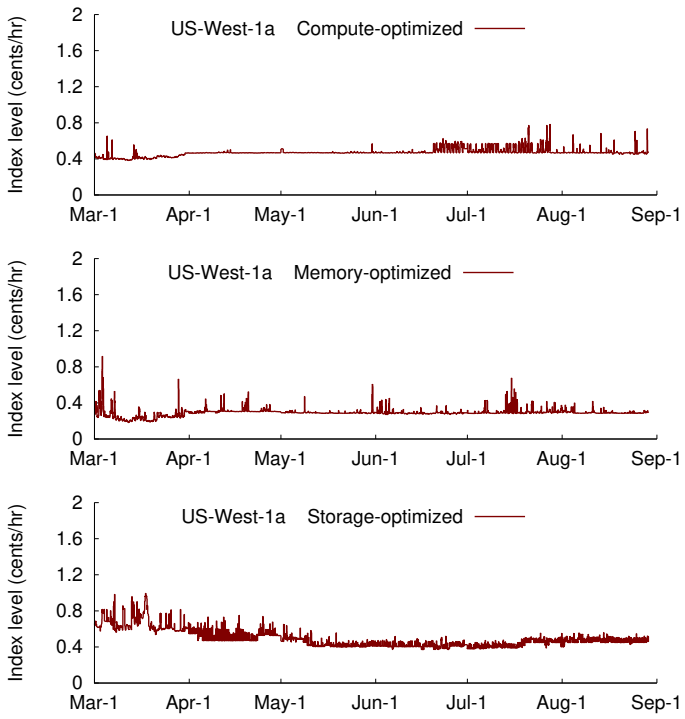


Figure 5.3: Indices of server families within a datacenter.

mate conditions, it does provide significant cost saving opportunities for flexible applications (even without using spot servers).

Since on-demand prices vary across regions, the magnitude of cost savings from choosing particular regional spot markets also varies widely. For example, while the index levels of `EU-West-1` and `EU-West-2` (not shown here) hover around 0.45 and 0.3 cents/hour respectively, indicating a 33% price differential, this is reflective of the ∼30% price differential that exists in their on-demand price levels. However, many price inversions do exist between on-demand and spot markets. For example, though `AP-Northeast-1` is slightly more expensive than `AP-Southeast-1` for on-demand servers, their spot market averages are flipped, with `AP-Northeast-1` offering 60% discount over `AP-Southeast-1` region as shown in Figure 5.6.

Insight (on inversions): *The market index allows users to readily identify systematic price differentials, inversions and arbitrage opportunities both within the spot markets and across different types of EC2 contracts.*

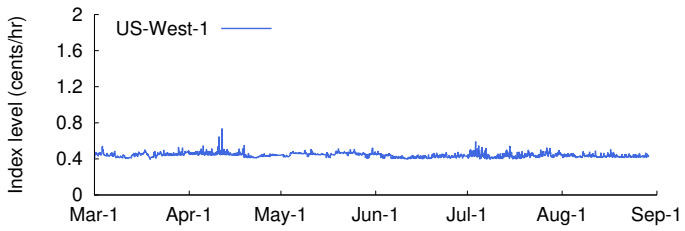


Figure 5.4: Index at the regional level

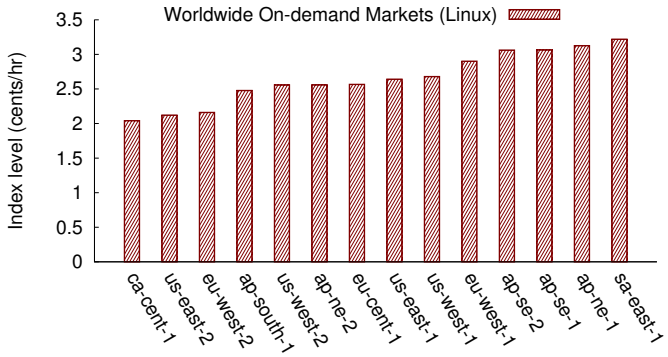


Figure 5.5: On-demand prices vary across regions.

5.3 Design of Index-tracking

Our system design goal is simple: run a given application on variable-price spot servers such that it incurs a predictable expense. Given that we have devised a market index that exhibits reliably predictable cost-efficiency for groups of spot markets, there is a trivial solution: build a cluster composed of one spot server from each of the constituent markets such that the overall cluster's cost-efficiency always matches that of the index. While trivial, this solution is not practical for generic applications. So, we aim to realize the performance of market indices without replicating its scale i.e., even single-node application should achieve cost-predictability. In this section, we design mechanisms and policies toward that goal.

We approach this in two steps: first, determine a broad set of candidate spot server markets that satisfy application's resource requirements. For this set, compute the cloud index to get the target cost-efficiency. Second, from amongst the candidate markets, select the best server (§5.3 outlines three policies for this selection) that meets the target level.

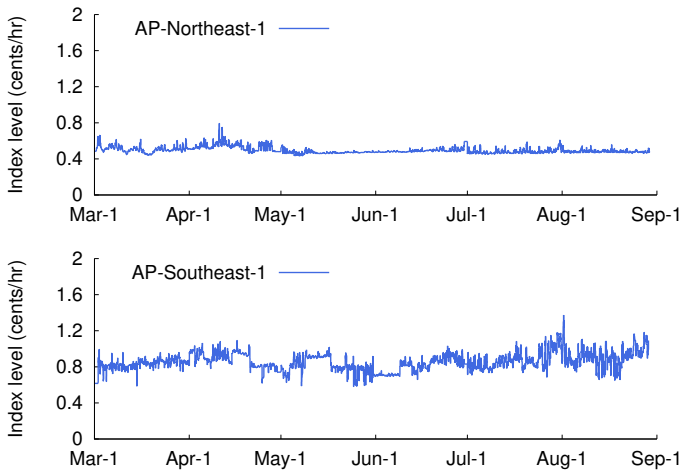


Figure 5.6: Indices showing price inversion across regions.

If changes in market conditions or application characteristics render the selected server no longer meeting the target, then transparently migrate the application to another server that does (§5.3 proves why such a market always exists). Our design draws inspiration from two techniques followed in the financial markets: (i) *index funds*, which are financial instruments constructed to track the performance of a reference market index, and (ii) *active trading*, the strategy of actively trading stocks and other instruments in the short-term in order to benefit from market volatility. However, there are significant differences between financial instruments and cloud servers such that these techniques are not applicable as is. Remainder of this section addresses these challenges.

5.3.1 Index Tracking by Server Hopping

Fundamentals of Tracking. Index tracking is a rule-based investment mechanism with a goal to match the financial returns from a portfolio to the performance of the market index it tracks. Originally conceived to quell the notion that *one cannot buy the averages*, index funds have grown to account for ∼20% of all the managed funds in the U.S. Their efficacy is rooted in the Efficient market hypothesis [33], which states that the stock prices fully reflect all available information such that the benefits of acting on information do not exceed the transaction costs. Simply, the hypothesis implies that one cannot consistently beat the market by predicting future prices of stocks. We have devised the cloud index on the same

principle that while it is hard to predict the future prices of individual spot markets, it is possible to reliably predict the behavior of certain groups of markets in aggregate.

Adapting to cloud servers. While our design retains the high-level goal of matching (or improving on) an index's performance, we are constrained by having a portfolio of one server at a time. To track the performance of the selected server i with respect to the reference cloud index $\mathbb{I}$, we define

$$\mathbb{G}ain(t_1, t_2) = \sum_{t=t_1}^{t_2} (\mathbb{I}(t) - \hat{P}_i(t)) \cdot \sqrt{C_i \cdot M_i} \tag{5.3}$$

where $\mathbb{G}ain(t_1, t_2)$ represents the gain on the index between times t_1 and t_2 that the server i was held, while P_i, C_i and M_i denote the server's price, CPU and memory capacity respectively. In order to keep the gain positive (i.e., maintain a cost-efficiency at or better than the index level), it may become necessary over time to migrate to a better server.

Server Hopping. This derives from the techniques such as day trading in financial markets, and loan refinancing in credit markets, where the objective is to benefit from favorable market conditions by actively trading one's assets or obligations. Recent advances in container virtualization, datacenter networking and per-second billing models have made it possible (and even attractive) to frequently migrate applications from one cloud server to another in response to real-time dynamics. For example, Supercloud [74] live migrates applications in response to geographically shifting workloads, and HotSpot [70] migrates applications to more cost-efficient servers in spot markets. Server hopping algorithms are designed as localized greedy optimizations: they incur upfront migration costs in the hopes of future benefits.

Our primary goal in adapting server hopping is to prevent the portfolio server from becoming cost inefficient with respect to the index level. But every hop reduces the already accrued gain on the index. To account for this, we consider the overheads of paying for two servers for the duration of migration, while making no progress on the application's work. Thus, $\mathbb{L}oss(i, j)$ quantifies the monetary loss of hopping from spot server i to j, with migration taking time T_m.

$$\mathbb{L}oss(i,j) = (P_i(t) + P_j(t)) \cdot T_m \tag{5.4}$$

By tracking an application's $\mathbb{G}ain$ and accounting for its $\mathbb{L}oss$ over the course of its lifetime, we can determine its overall cost-efficiency vis-a-vis the index. Next, we analyze the properties of these mechanisms and the market conditions under which the index-level cost-efficiency could be maintained.

5.3.2 Properties of Tracking-by-Hopping

There will always be a cost-efficient market to hop.

While this property is critical for the functionality of our algorithm, it is also the easiest to establish. In the base case, when the candidate set contains only one spot server market, the cost-efficiency of that market is the same as that of the market index (by definition 5.2). So, one can always hop back to the default spot market. Next, when the candidate set contains multiple markets, there needs to be at least one spot market whose cost-efficiency is better than or equal to that of the index level (this follows from the definition of market index, which is the average of the constituent market's efficiencies). Thus, there will always be a spot market whose cost-efficiency is better than or equal to the index level.

However, the mere existence of an efficient-cost market at all times does not imply that the overall cost-efficiency target would be met. In fact, it is trivial to prove the opposite: consider a set markets such that the cost-efficiency of one half of them are below the index-level and the other half is above the index-level. Let us also say that these markets are extremely volatile such that at every unit of time, the markets in each of these halves swap i.e., those with better than index efficiency become worse and vice-versa. Under such a volatile setup, the application continually ends up hopping, leaving itself no time to perform any actual work. Thus, for tracking-and-hopping algorithm to be viable, the negative impact of hopping overhead needs to be compensated by gains in index tracking.

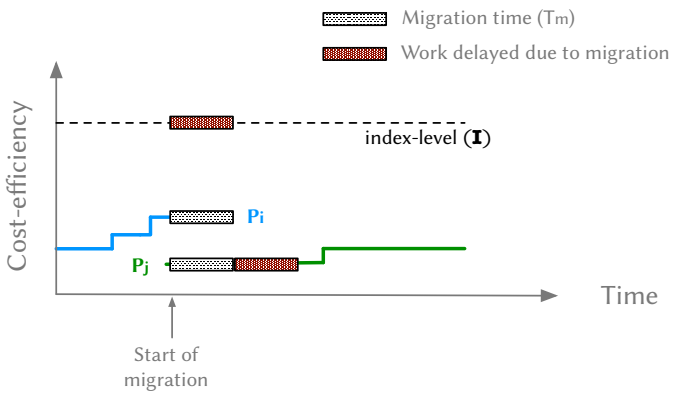


Figure 5.7: Illustrating the sufficiency condition to accommodate the overhead of migration.

Necessary and sufficient conditions.

It is trivial to establish the *necessary and sufficient condition* for tracking-by-hopping to be effective: If the aggregate gain on tracking exceeds the cumulative losses on hopping for the entire duration of hosting, the mechanism would have met the goal.

However, we derive a sufficiency condition that helps make localized greedy decisions instead of having to wait till the end of the execution to verify meeting the target cost-efficiency. Figure 5.7 illustrates an application migrating from server i to j with migration taking time T_m (shown by the gray area). In order to completely absorb the overhead of this migration, we need to account for not only the $\mathbb{L}oss(i,j)$ but also the actual work that has gotten delayed by migration (shown by the red area). Thus, if the cost-efficiencies of two spot markets satisfy the following condition with respect to the index-level, then hopping would not affect any previously accrued $\mathbb{G}ain$ on the index. Note that this condition is *sufficient but not necessary*.

$$\hat{P}_i(t) + (2 \cdot \hat{P}_j(t)) \leq \mathbb{I}(t) \tag{5.5}$$

On the efficacy of the mechanism.

Another advantage of tracking-by-hopping mechanism is that it *scales well with increased adaption*. As more users seek to achieve index-level cost-efficiency, we expect the market to become increasingly stable since everyone's target is the fair market value of the idle cloud

capacity. In turn, this should reduce the number of server hopping required to maintain the desired cost-efficiency, thereby leading to an increased application availability. This is in contrast with the behavior induced by HotSpot, where every application is actively trying to hop to the most cost-efficient server, which could exacerbate the market volatility.

5.3.3 Server Selection Policies

Now that we have established the existence of one or more spot markets that satisfy the target cost-efficiency levels at all times, we present three policies that enable a tradeoff between *lower cost* and *higher availability* while maintaining the target cost-efficiency.

Cost-centric Policy. The goal of this policy is to maximize cost savings by aggressively migrating to the *best-fit server* that is also cost-efficient. In order to determine this, we re-normalize the server's cost-efficiency from Eq-5.1 to take into account the actual resources utilized at time t, namely C_{util} and M_{util}.

$$\breve{P}_i(t) = \frac{P_i(t)}{\sqrt{C_{util} \cdot M_{util}}} \tag{5.6}$$

Thus, cost-aware policy chooses the spot market that provides the best $\breve{P}_i$ value at the given time. Since availability is a not concern, selections are triggered every time a better fit cost-efficient server emerges due to any changes in spot market or application behavior. However, in order to maintain the target cost-efficiency, only those migrations that satisfy Eq-5.5 are carried through.

Availability-aware Policy. The goal here is to maximize application's availability by selecting a stable server that also meets the cost-efficiency targets. To identify such a server, this policy computes the standard deviation of each market's price with respect to the index-level over a predefined window. Then, from amongst the spot markets, whose average is below the index-level, it picks the one with the least deviation. No further proactive selection is triggered until the chosen server's cost-efficiency crosses the group's index level.

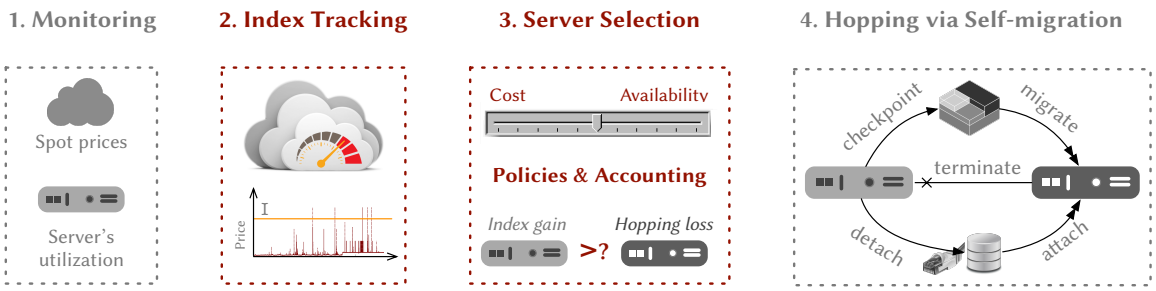


Figure 5.8: System architecture with HotSpot components boxed in gray and our extensions in red.

Balanced Policy. To mind the gap between the two extremes, we define a policy whose goal is to achieve a balance between higher cost efficiency and higher availability. We infer that higher a spot market's price variability, higher the risk of needing to migrate away. In order to balance this risk-reward tradeoff, we employ the *Sharpe ratio* [69], a statistical measure commonly used in finance to compute the risk-adjusted returns of an asset. We define balance factor as a variant of the Sharpe ratio,

$$\mathbb{S}_i(t) = \frac{\mathbb{I}_g(t) - \breve{P}_i(t)}{\sigma_i} \tag{5.7}$$

where $\mathbb{I}_g$ is the index level of the group, $\breve{P}_i$ is the server's average cost-efficiency over a small window, and σ_i is the standard deviation of the spot server's cost-efficiency with respect to the index level over the same window. While the numerator estimates the server's current "return" relative to the index-level, the denominator quantifies its expected "risk" of deviating from the return and thus needing to migrate again. In this policy, hopping is triggered only when the current server is no longer the one with highest balance factor, thus minimizing migrations while not sacrificing on cost-efficiency.

5.4 Implementation

Since we propose a middle ground between fully-predictive and fully-reactive approaches to spot server management, we had several options to build on the prior work. However, given the ease of adding a predictive component (i.e., index tracking) to an already

functional reactive system (that does server hopping), we chose HotSpot [70] as our base framework.

5.4.1 HotSpot Overview

HotSpot introduces a self-migrating server abstraction for containerized applications. It works by (i) continuously monitoring the spot market prices and application's resource utilization, (ii) periodically performing cost-benefit analysis to determine whether to stay or migrate to a cheaper server, and finally (iii) migrating the containerized application to the newly chosen server and shutting down the current one. Since this logic is embedded inside the Amazon Machine Image (AMI), any EC2 server booted with it becomes a self-migrating server (i.e., runs this control loop throughout its lifecycle). Thus, HotSpot is easily adaptable: it requires no application modifications nor any external infrastructure support.

Since we reuse the HotSpot framework, we also inherit some of its restrictions. First, due to LXC integration, we can only support stop-and-copy migration. Second, hopping semantics force applications to use remote storage and virtual network i.e., Elastic Block Storage (EBS) and Elastic Network Interface (ENI) respectively. Finally, HotSpot is architecturally decentralized i.e., each server manages itself without explicit coordination with others. Thus, efficient coordinated deployments in multi-node configurations may need a centralized orchestrator (we address this in evaluation).

5.4.2 Extending HotSpot

HotSpot is implemented in `Python` with additional integrations with EC2's `Boto3` library, `LXC` bindings and administrative `Shell` scripts. We retain the monitoring and hopping components but replace the cost-benefit analysis logic with index-tracking and server selection modules as depicted in Figure 5.8. We implement our extensions in ∼850 lines of `Python`.

First, we build a standalone cost estimator utility that takes in application's resource constraints, then computes the market index-level for the selected availability zone, and finally predicts the overall cost to be incurred. This enables users to know their expenses before starting the workload. A library version of this utility is integrated in the index-

tracking module, which in turn polls the monitoring engine once every five minutes to update the gain on index-tracking. This also triggers the server selection policy, which iterates through all the applicable spot markets to determine if server hopping is required. For the most populous zone (`US-East-1a` with 106 spot markets), this whole operation sequence of monitoring, tracking and server selection takes an average of ∼2 seconds. We configure the migration module for direct memory-to-memory transfer as it maximizes application's availability. We find that the migration latencies observed in HotSpot are still current: migration of up to 32GB RAM takes ∼30 seconds (as it is bottlenecked by the EBS/ENI transfer happening in parallel), and the latency for larger memory sizes (up to 128 GB) increases linearly at the rate of ∼1 second per GB.

5.5 Evaluation

We hypothesized that we could achieve cost-predictive server hosting by modeling the spot markets in aggregate via a *cloud index* and then by tracking it via server hopping. Additionally, we presented server selection policies that enable a tradeoff between higher availability vs. lower cost, while maintaining the index predicted cost-efficiency. Our evaluation investigates the validity of these claims by setting up experiments that answer two key questions: (i) How do server selection policies perform under different market and application conditions? (ii) How effective is index-tracking on EC2 spot markets, and how does it fare against fully-predictive and fully-reactive approaches? We quantify the former via prototype experiments, and the latter via simulations of jobs from Google clusters.

5.5.1 Prototype Experiments

In order to evaluate the server selection policies (described in 5.3), we have to be able to control application's and spot market's characteristics. Since the prototype is intended to run real workloads on EC2 instances with real-time prices, it limits our ability to control key parameters. Thus, in the following set of experiments, while our prototype is deployed and run on the EC2 platform, we stub out certain EC2 API calls (for e.g., real-time spot price querying). We also use an emulated job to better control application's CPU and memory

usage. Below, we describe these setup, establish a baseline performance and then quantify the effect of varying key parameters.

Application. To predictably control the application behavior, we emulate the job using lookbusy [25]. Our job runs for an hour on the reference server `m4.2xlarge` and has two distinct resource utilization phases. In the first phase (lasting 30 minutes), it consumes 4 vCPUs and 16 GB of memory while in the second phase (the next 30 minutes), it consumes 2 vCPUs and 8 GB of memory.

Spot Markets. To ensure identical market conditions over different runs, we generate synthetic spot price traces for four spot markets: `m4.large`, `m4.2xlarge`, `c4.2xlarge` and `r4.xlarge`. These are chosen as their vCPU varies between 2-8 and memory capacity between 8-32, which cover the entire spectrum of our application's resource utilization. We model their prices as follows: `m4.large` has an average price of 4.5 cents per hour and a standard deviation of 0.5, `m4.2xlarge` has the same standard deviation but an average price of 8.5 center per hour, while `c4.2xlarge` and `r4.xlarge` have identical average price of 6.5, the former has a standard deviation of 1, while the latter has 1.1. Figure 5.9(left) gives an illustrative representation for these markets. For our experiments, the instantaneous spot price is computed randomly such that their average and standard deviation characteristics hold good. We use the same per-second billing model that EC2 operates on.

Baseline Result. Figure 5.9(right) shows both the cost incurred and availability achieved by the three different policies. We normalize the cost to that of a reference server running at the index-level cost-efficiency. We observe that all three policies perform better than the index predicted levels. Also, expectedly the cost-centric policy realized the cheapest run, and the availability-aware policy attained the highest availability. The balanced policy managed to be within 12% of the lowest cost, and 1.7% of the highest availability.

Changing Application Behavior. Next, we modify the baseline configuration of the application to exhibit more diversity in its resource consumption as depicted in Figure 5.10 (left). The resource variations are such that the application could be executed on at least one of the four target spot markets at all loads. Figure 5.10 (right) shows the results of hosting the three different application configuration. First thing to notice is that the performance

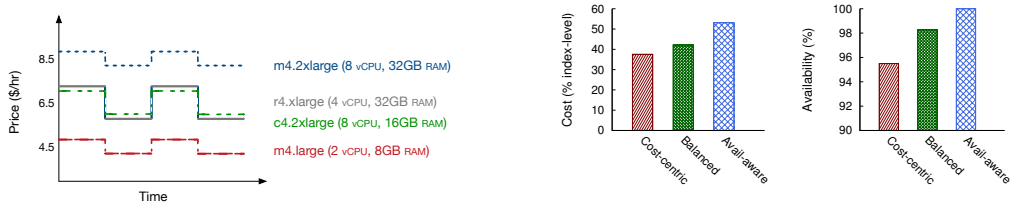


Figure 5.9: Spot market setup (left), and the performance tradeoffs (right) at the baseline configuration.

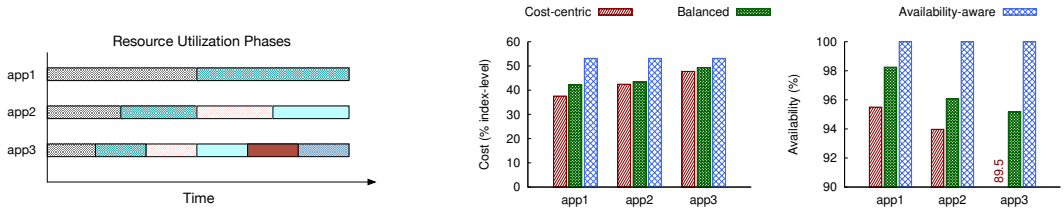


Figure 5.10: Performance of policies when application's resource utilization varies.

of the availability-aware policy does not change at all, which is not unexpected because this policy optimizes for stability and not savings. Next, we see that both balanced and cost-centric policies incur increasing overheads as the application's utilization gets bursty. As the markets have remained the same, the cost-efficiency gains of moving to a better fit server gets overtaken by the migration overheads. However, since the balanced policy does not react to changes as quickly as the cost-centric policy, its losses are less pronounced.

Changing Market Behavior. Finally, we vary the baseline market conditions to see its impact on the policies. We achieve this by changing the standard deviation. Figure 5.11 then shows how the policies perform under more volatile conditions. We plot the increase in market volatility (compared to the baseline) along the X-axis. The first graph shows that both cost-centric and balanced policies slightly improve their cost efficiencies when the market volatility increases. Interestingly, the availability policy, when forced to migrate under more volatile conditions, has managed to reduce its cost as a side effect of repeated migrations. However, the availability graph shows that all policies suffer, when markets are more volatile. Our experiments in §5.5 give a better sense of the current state of the EC2 markets as they use real spot price traces.

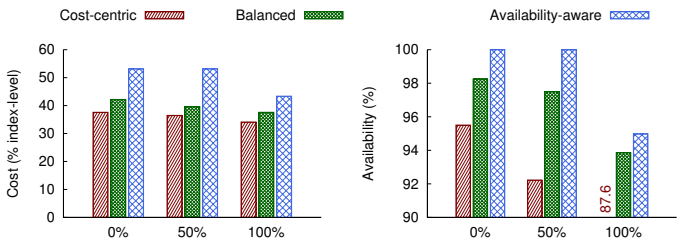


Figure 5.11: Policies under changing market volatility.

Summary. *The balanced policy using the Sharpe ratio consistently achieves better cost-efficiency tradeoffs than the extreme policies, under varying market conditions and application's resource utilization.*

5.5.2 Simulation Experiments

The goal of our simulation experiments is to quantify the efficacy of different spot server management techniques under realistic spot market conditions. We evaluate three approaches for two categories of applications using job traces from Google cluster and price traces from EC2. Below, we describe each of these as well as our experimental findings.

Spot Markets. For these set of experiments, we use EC2's spot price traces from the `US-West-1` region between 1-MAR-2017 and 31-AUG-2017. We run three separate trials, one for each of its three zones (`1a`, `1b`, `1c`), whose index levels are depicted in Figure 5.2. While each zone consists of 79 Linux markets, every jobs considers only the set of markets that meet its minimum resource requirements.

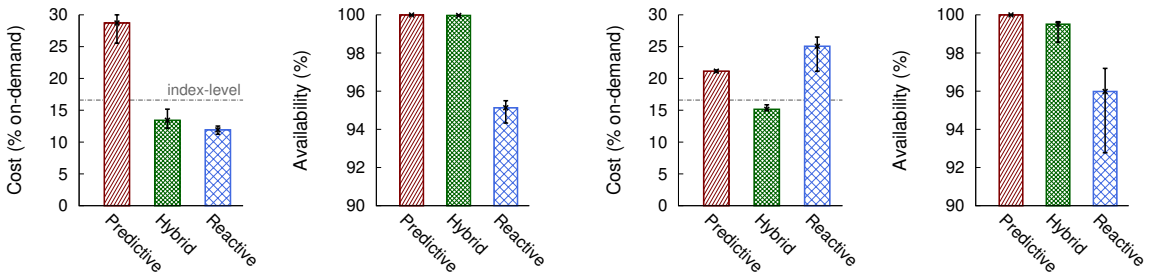
Server Management Techniques. The three approaches to spot server management that we evaluate are: (i) Fully-predictive, (ii) Fully-reactive, and (iii) Hybrid. A fully-predictive system employs static selection such that once a server is selected from amongst the available pool of spot servers, the application is continually hosted on it as long as the server is not revoked. If that happens, the selection process is repeated and the application is restarted on the new server. Amazon's `Spotfleet` tool is an example of this style. In contrast, the fully-reactive approach continually looks for better servers and as soon as one is found, it migrates the application. `HotSpot` belongs to this category. Finally, the hybrid

approach uses `index-tracking` as the predictive component and server hopping as the reactive component. We configure its server selection to be driven by the balanced policy.

5.5.2.1 Long-running Occasionally-interactive Applications

This emerging application category includes data sink servers for IoT sensors, cryptocurrency miners and peer-to-peer file trackers. While they are flexible in tolerating moderate downtimes and application restarts, they typically do not benefit from the classical fault-tolerance mechanisms like checkpointing or replication. Thus, we run the application without any fault-tolerance, and investigate how the three spot server management approaches manage its hosting. To do so, we simulate running the application for a duration of *6 months*. The simulator is seeded with the following characterization of the application's behavior. First, the application requires a minimum of 2 vCPUs and 10GB of memory. While its performance degrades below these levels, it does not scale up with additional resources. Second, the application could be transparently migrated with a stop-and-copy migration, and that it would incur a downtime of 30 seconds given its resource levels. Finally, an application restart following a server revocation would incur a downtime of 90 seconds (for acquiring a new spot server, and setting up EBS/ENI).

Figure 5.12a shows both the overall cost and availability of running the application over the 6-month window. To establish a baseline, we simulate running the application on the cheapest on-demand server that meets the resource constraints, which happens to be `r4.large`. Then, we normalize the running cost of all techniques to this level. First, we see that all three approaches are substantially cheaper than on-demand hosting. But the reactive and hybrid schemes not only manage to meet the index-level cost-efficiency but also achieve ∼50% cost reduction over the predictive approach. Next, in terms of availability, the predictive and hybrid schemes achieve three nines of availability whereas the reactive scheme manages only 95%. Under the hood, we observe that the predictive scheme experienced an average of 4.33 revocations, the reactive scheme migrated 4208 times with no revocations, and the hybrid scheme suffered 1 revocation and chose to perform 24.66 migrations.



(a) Long-running application

(b) Parallel applications

Figure 5.12: Comparing the fully-predictive, fully-reactive and hybrid server hosting systems on EC2 spot markets.

5.5.2.2 Parallel Synchronous Applications

A staple of high-performance scientific computing, these applications are deployed in multi-server configurations with all the servers working in lock step, often involving significant data exchanges and synchronizations. Thus, a downtime for one server negatively impacts all other servers interacting with it. In this experiment, we evaluate how decentralized server management approaches cope with this parallel setup. To do so, we simulate running 1000 randomly selected jobs from the Google cluster traces [61]. Internally, each of these jobs comprise of worker tasks (ranging from ∼10-500) that are run on separate servers but coordinate during their execution. Google traces report the CPU and memory consumption of every task at the granularity of 300 seconds. The run length of jobs vary between ∼10-720 minutes. We execute each job, at the time it arrives by choosing the best spot servers for each of its tasks as per the hosting technique. Within the simulator, we make three operational assumptions: (i) if a task gets revoked, the whole job is restarted, (ii) when a task is migrated to a new server, all other tasks of that job pause until the migration is fully completed, and (iii) we consider the job as completed only when all of its tasks are finished.

Figure 5.12b then shows the cost and availability of running these jobs. For the cost graph, we normalize the Y-axis to that of running all the jobs on the cheapest matching on-demand servers. We see that the hybrid scheme not only meets the index predicted cost levels but also comes out 30-40% cheaper than the other two schemes. The reactive scheme

suffers from asynchronous migrations, where a small number of hopping nodes hold a large number of communicating nodes frozen for the duration of migration, thereby increasing the overall cost. This problem was not as pronounced in the hybrid approach since its policy naturally encouraged synchronous (and fewer) migrations. On the other hand, the predictive scheme improved its performance (compared to prior experiment) as it benefited from having a large number short jobs that in turn reduced the probability of revocation and also increased its ability to find better matched servers repeatedly. Availability paints a similar picture as before: predictive achieved four 9s, hybrid managed three 9s, and reactive mustered ∼96%.

Summary. *For long-running as well as parallel applications, the hybrid approach meets the index predicted cost-efficiency. Not only that it also achieves the best combination of lower cost and higher availability compared to other approaches.*

5.6 Related Work

To the best of our knowledge, this is the first work to apply market indices to analyze cloud spot markets, as well as propose a mechanism for cost-predictive server hosting on variable-price spot markets. While a preliminary version of this work appeared in [71], the current treatment differs in three significant ways: (i) we validate our intuition for using cloud indices via infrastructure-level observations in public cloud datacenters, (ii) we have a more specific goal (on cost predictability) and do not consider global trading, and (iii) we present a rigorous treatment of the index composition, tracking-by-hopping mechanism, and server selection policies. Below, we describe the related work in detail.

Spot market predictions. The first category of related work comprises of modeling spot markets with a goal to predict its future behavior. The earliest work came from Ben-Yehuda et.al. [22] in 2013, and has been followed since then by a large body of work including [92, 93, 7, 41, 87]. These work mainly focus on predicting the behavior of individual server markets, whereas our work proposes to analyze markets in aggregate. In conjunction with prediction schemes, researchers have also developed bidding strategies [99, 76, 66] for

different types of applications. However, bidding policies are orthogonal to our work since our system migrates away from risky (i.e., cost-inefficient) markets naturally.

Financial concepts applied to spot markets. The next category is the application of financial and economic concepts to cloud spot markets. Prior efforts include creating an options market [82], adapting the modern portfolio theory [67], implementing active trading [70], proposing asset pricing [73], and composing derivative cloud markets [78, 68, 98]. However, none of these share our goal of realizing cost-predictive spot server hosting.

Market indices for non-cloud environments. Financial market indices have existed for a long time with the Standard and Poor's index dating back to 1923. Our work extends and adapts the index construction methodology of several indices including the Consumer Price Index [55], the S&P and Dow Jones [43]. Researchers have applied market indices to other spot markets like electricity [32]. However, the unique characteristics of compute-time namely, its state and the use-it-or-lose-it property make its application distinct from the prior indices.

System design aspects. Finally, our work derives several system design elements from HotSpot [70], SmartSpot [42], and Supercloud [74]. These include mechanisms and policies for container- and nested VM migration, automated server hopping, and decentralized server lifecycle management. However, the goals of these projects are distinct from ours. Both SmartSpot and Supercloud employ migration to lower access latency and improve resiliency but do not focus on spot market dynamics. While HotSpot uses automated server hopping to reduce server hosting costs, it makes no predictions on the resulting cost-efficiency.

5.7 Other Applications: Mitigating Spatial Price Risk

In this section, we extend the cloud index framework to address price risk. While Chapter 4 designed a solution to mitigate price risk, the focus was temporal in nature i.e., price risk arising in a given geogrpahical market over time. However, as the global footprint of the spot markets have expanded, the applications are exposed to another form of price risk: *spatial price risk* i.e., risk that a chosen server's price rises relative to servers in different geographical locations. We posit that enabling applications to operate in global

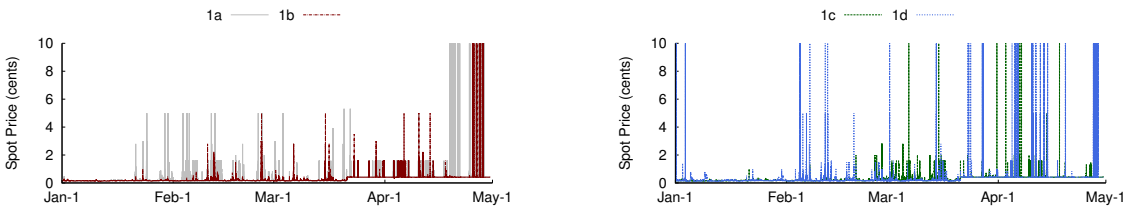


Figure 5.13: Price of a representative Linux server (r3.4xlarge) across four availability zones of the US-East-1 region.

spot markets, similar to how financial investors trade in global financial markets, provides additional cost-saving opportunities. In this section, we address the challenges of deploying applications in global spot markets, and demonstrate how cloud indices can help solve those.

5.7.1 Server Trading in Global Spot Markets

EC2's global footprint is massive and complex: it operates in 16 worldwide regions each of which comprise of 2-6 availability zones, and it has announced plans to add 6 new regions with 17 additional zones in the future [3]. EC2 spot markets, which Amazon uses to sell the unused compute capacity in its datacenters, have the exact same global footprint. Since EC2 sets a different dynamic spot price for each type of VM in each availability zone of each region, the global spot market currently includes more than 7600 separate server "listings". Notwithstanding the global footprint, the spot prices are hard to predict even for identical VMs within a region. For example, Figure- 5.13 shows the price of `r3.4xlarge` Linux server in four availability zones of `US-East-1` region. However, prior work have largely restricted their scope to either individual servers or those that are located within the same availability zone (or datacenter).

Understanding and modeling server migration (interchangeably referred to as *server trading*) is essential for any system that reacts to market changes in real-time especially since trading can be sticky. In EC2, the overhead of migrating an application and its data is governed by the geographical separation between the source and destination VMs, and the data size. Figure 5.14 shows the organization of EC2 spot markets into regions and availability zones, with each zone housing ∼150-200 spot markets. To keep our migration model consistent across geographical boundaries, we employ stop-and-copy migration,

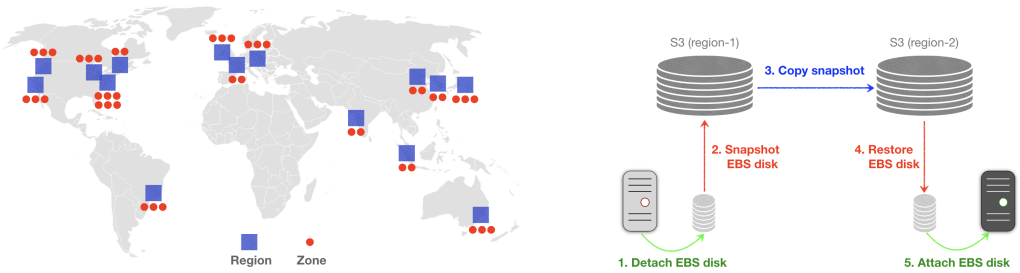


Figure 5.14: EC2's global infrastructure comprises of 44 availability zones or datacenters (shown in red) organized within 16 regions (shown in blue). Intra-zone migrations require only detaching and attaching of EBS disk (shown in green), inter-zone but intra-region migrations additionally require EBS to be snapshotted and restored (shown in red) and finally, inter-region migrations require copying snapshots across the regional data stores (shown in blue)

which unlike live migration, minimizes the migration costs in exchange for some application downtime. We also assume that applications use remote disks like EC2's Elastic Block Storage (EBS) that support built-in APIs for migration.

Figure 5.14 then depicts a simplified view of EC2's data migration model. If both the servers are located within the same zone, the EBS volume can simply be detached and remounted (as shown in green), resulting in a fixed overhead that is independent of the data size. Our microbenchmarks show these operations to take an average of ∼30 seconds. In contrast, when the servers are in different zones but within the same region, the EBS disk needs to be snapshotted and recreated fully (via a globally accessible datastore like S3) between the detach and remount operations (as shown in red). However, snapshot and restore operations need to be run serially, and their time to completion depends on the data size. Our microbenchmarks indicate an average rate of 1-2 minutes per GB of data. Finally, for inter region migrations, there is an additional operation of copying the EBS snapshot from source region's S3 bucket to that of the destination region (shown in blue). While the overhead of snapshot copying varies across regions, our microbenchmarks show it to be 15-60 seconds per GB. Putting these together, Figure 5.15 chronicles the overhead of migrating a 10GB EBS disk across and within regions. Thus, for an application with a data footprint of 32GB, migrating once per day results in an availability of 99.96% for intra-zone, 96.6% for intra-region and 95.2% for inter-region.

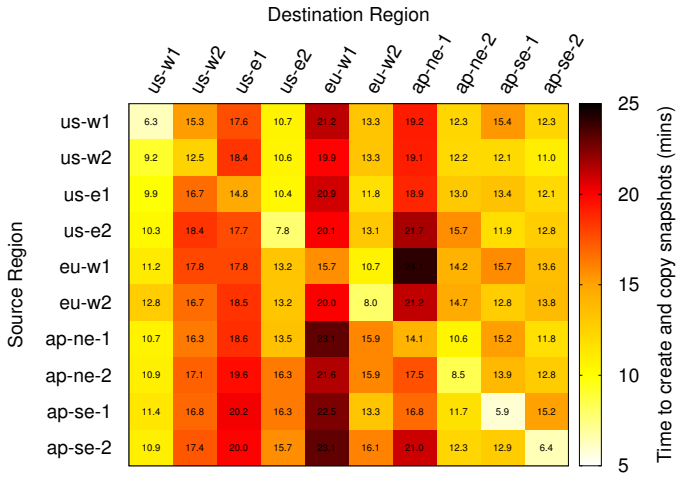


Figure 5.15: Global migration overheads for a 10GB snapshot.

While our analysis focused on the transfer of disk data, a stateful migration also requires trasfering the current CPU and memory state of the VM or the container. Since EC2 VMs support a network bandwidth of 25Gbps within a datacenter, a direct VM-to-VM transfer of the application state would not exceed the EBS migration overhead (of ∼30 sec). For transfers across zones and regions, the application state could be dumped on the EBS, so that it gets migrated along with the disk data.

Summary: *Compared to intra-zone migrations (which incur a fixed overhead), inter-zone and inter-region migrations consume an orders of magnitude more time and are not practical, given the current network bandwidth.*

5.7.2 Index-based Global Trading

We demonstrate the importance of index-based global trading using a generic long-running application in simulation. We assume our application i) has no geographical constraints, ii) is capable of consuming whatever resources are available, and iii) executes within a virtualized environment, such as a nested virtual machine or resource container, that makes it capable of trading servers via transparent systems-level migration. We also assume the application can gracefully handle IP address changes when crossing regions and AZs, and employs fault-tolerance mechanisms, such as replication or checkpointing, to make it robust to revocations. Solutions to enabling these assumptions are well-known, as prior work on superclouds has resolved many of these "plumbing" issues [74].

We simulate the application's behavior over the past two months using real spot market prices from EC2's 2287 Linux spot markets. To enable trading, we assume the application monitors spot prices in each of these Linux spot markets, and includes a trading policy that dynamically migrates as prices change to the server with the lowest current price per ECU. Our simulation accounts for the overhead of trading across AZs and regions based on Figure 5.14. We define multiple global and local trading policies, as outlined below.

- **Market-based No Trading** selects the individual spot server across the global market with the highest Sharpe ratio below, which is a standard measure for estimating an asset's risk-adjusted returns: for an asset i, it is the ratio of the expected difference between the asset's returns R_i and the risk-free returns R_{free} divided by the standard deviation of the returns σ_i. In this case, the on-demand price captures the risk-free returns. As in nearly all prior work on spot instances, this policy commits to its chosen server and never trades, regardless of price changes.

$$S_i = \frac{E[R_i - R_{free}]}{\sigma_i} \tag{5.8}$$

- **Market-based Local Trading** selects the individual spot server in the global market with the lowest price per ECU, and then actively trades *within that server's AZ* to ensure it always runs on the server with the lowest price per ECU. Thus, this policy avoids any trading overheads from crossing AZs and regions.
- **Market-based Global Trading** selects the spot server in the global market with the lowest price per ECU, and then actively trades *across the global market* to ensure it always runs on the server with the globally lowest price per ECU. The policy incurs the trading overheads from crossing AZs and regions.
- **Index-based Global Trading** first selects the AZ with an index having the highest Sharpe ratio, then selects the individual spot server in that AZ with the lowest price per ECU, and finally actively trades *within that server's AZ* to the server with the lowest price per ECU.

Note that incorporating risk, in this case using the Sharpe ratio, is most important when committing to a subset of markets (either the individual server market in the first bullet or

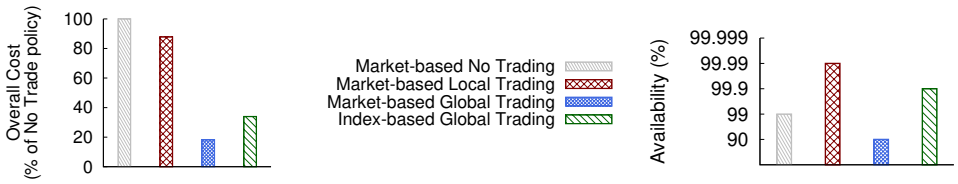


Figure 5.16: *Comparison of cost and availability of global trading policies.*

the AZ in the last bullet). Considering risk is much less important when actively trading, as the application often migrates before revocations ever occur.

Figure 5.16 shows both the overall cost (left) and availability (right) of running our generic application over the two month period. The cost is normalized as a percentage of the No Trading policy to illustrate the benefits of actively trading servers as market prices change. As mentioned above, this No Trading policy is similar to policies in prior work, which commit to spot servers and only select a new server after a revocation [65, 68, 78, 99]. Since spot prices for individual servers are generally stable, revocations are rare, and thus there are few opportunities for selecting a new server in prior work. However, given the large number of individual server markets, the lowest cost server (across the global market, region, or AZ) actually changes quite frequently. Thus, policies that actively trade servers can reduce their costs relative to policies that do not actively trade. As the figure shows, the No Trading policy incurs a higher cost than all of the active trading policies.

The figure also shows that the Market-based Local Trading policy incurs a much higher cost than the Market-based Global Trading policy. Since the local trading policy only trades within its own AZ to eliminate trading overhead, it cannot take advantage of low prices in other AZs. In general, the individual server in the global market that has the best combination of price and risk, as measured by the Sharpe ratio, is not necessarily contained in the AZ with the best combination. Overall, the Market-based Global Trading policy achieves the lowest cost, even when accounting for its high trading overhead, as it always actively migrates to the globally least-cost server. In comparison, the index-based global trading policy, which commits to the AZ with the highest Sharpe ratio, but then restricts itself to intra-AZ trading to mitigate trading overhead, incurs only a slightly higher cost than the Market-based Global Trading policy.

To quantify availability, we assume the application is unavailable when trading servers according to the benchmarks in Figure 5.15 with intra-AZ trades incurring an unavailability of two minutes. The figure shows that, while the Market-based Global Trading policy has the lowest cost, it also has the lowest availability (1 nine) due to the high trading overhead imposed by frequently crossing regions and AZs. While it is costly, the No Trading policy exhibits a slightly higher availability (2 nines), since it never trades and only experiences downtime when its spot price spikes. The Market-based Local Trading policy has the highest availability (4 nines), since it also restricts trades to within its AZ; by actively moving to the lowest-cost server it experiences few price spikes that cause unavailability. However, the policy incurs a high cost, since it selects an initial server based on its price characteristics and not AZ-level characteristics. Finally, in this case, our index-based policy achieves the best of both worlds—a high availability (3 nines) at a low cost—by selecting an AZ with an index price that has low magnitude and variability, and then actively trading within it. Of course, the best policy is application-dependent, and varies based on an application's footprint and other availability constraints.

5.8 Conclusion

Our work stems from the most prevalent deployment concern of the spot markets namely *cost uncertainty*, and how the diversity and span of EC2 spot markets have exacerbated this concern. We observe infrastructure-level realities from public cloud datacenters, and devise a novel index for cloud spot markets. The insights from these indices enable us to design a cost-predictive server hosting framework. We implement and evaluate it on EC2 spot markets.

Benchmarking. Though this work primarily uses cloud indices for cost-predictive server hosting, we believe in its broad applicability beyond this purpose. For example, by succinctly representing the aggregate behavior of spot markets, cloud indices establish a benchmark for comparing the performance of spot server management techniques. This is especially important as researchers and startups are designing sophisticated strategies such as

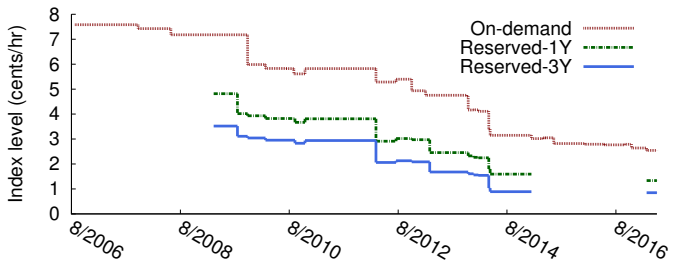


Figure 5.17: Index-levels of on-demand and reserved servers in the US-East-1 region, since EC2's inception.

portfolio diversification and derivative clouds, whose performances need to be vetted against a reference benchmark.

Beyond Spot Markets. In recent years, cloud computing platforms are rapidly evolving the IaaS offerings to cater to the diverse needs of cloud customers. While spot markets exhibit price and risk dynamism in short timescales of seconds and hours, other contract types like on-demand and reserved servers do so in terms of months and years. Cloud indices are a natural way to track this long-term evolution. For example, Figure 5.17 concisely represents the price trajectory of on-demand and reserved servers in the `US-East-1` region over the last decade. We posit that insights from cloud indices can drive informed investment decisions for cloud users.

CHAPTER 6

MINIMIZING VALUATION RISK VIA ASSET PRICING

> "Uncertainty is more stressful than knowing for sure something bad will happen."
>
> Archy de Berker et.al. [30]

A significant fraction of cloud capacity is idle [19] and providers are actively exploring ways to monetize it. The resulting transient server contracts are predominantly structured to provide flexiblity for cloud platforms to reclaim these servers any time. In this chapter, we outline the challenges consumers face in correctly valuing these transient servers, and demonstrate that even probabilistic information on transient server characteristics helps increase their utility dramatically, by more than 5×. We use this insight to design a new asset pricing abstraction, *transient guarantee*, which helps both providers (in setting prices correctly) and consumers (in determining if the prices are good).

6.1 Idle Cloud Pricing in the Wild

While on-demand servers are the most common cloud contract, they restrict a platform's control over its resources, as only users can decide how long they hold on-demand servers and when they release them. As a result, even though platforms guarantee high availability once an on-demand server has been allocated to a user, they do not guarantee obtainability [57] (i.e., requests for new on-demand servers can be rejected). Instead, for those customers who do not want to face this risk, cloud platforms offer *reserved servers*, which can be obtained anytime during the reservation periods of 1-3 years. To support reservations, platforms have only two options: *either keep physical resources idle or maintain a pool of resources they can reclaim to satisfy reserved requests.* Of course, keeping physical servers idle is highly inefficient, as it wastes their computational resources, as well as the capital and operational

expenses incurred to provide them. Thus, transient servers exist both to reduce this waste by enabling platforms to earn revenue from their idle capacity, and also to provide a pool of revocable resources to support reservations.

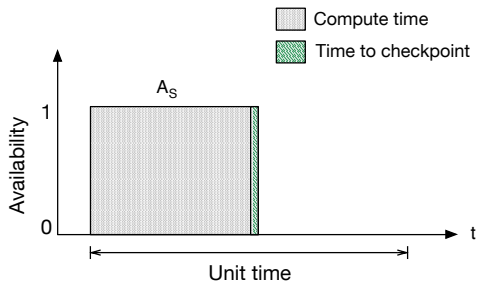
Since transient servers are a relatively new concept, there are not yet widely accepted standards for setting their terms and prices. EC2 offers its version of transient servers, called *spot instances*, via a market mechanism. Users place a bid for servers by specifying the maximum price they are willing to pay per unit time. EC2 then provisions the servers if the bid price is greater than the servers' current spot price, which is market-based and varies in real time. However, if the spot price rises above the user's bid price, EC2 revokes the servers. In contrast, GCE charges a fixed price for transient servers, called *preemptible instances*, such that it will always revoke them within 24 hours. Importantly, EC2 and GCE currently reserve the right to revoke transient servers *at any time*.

Our key insight is that *transient servers' revocation characteristics influence their performance relative to on-demand servers*, since these characteristics affect the overhead of the fault-tolerance mechanisms applications employ to handle revocations. As we show later in Section 6.3, knowing even probabilistic information about a transient server's revocation characteristics can increase its performance by enabling users to optimally configure fault-tolerance mechanisms. Unfortunately, the revocation characteristics for EC2 and GCE are unknown and unbounded. Due to the lack of information, EC2 and GCE users are also unable to accurately quantify transient server value. For example, while a transient server may be 50% the price of an on-demand server, its unknown revocation characteristics may result in a 50% performance overhead due to fault-tolerance. Thus, "cheaper" transient servers may actually offer no normalized discount relative to on-demand servers.

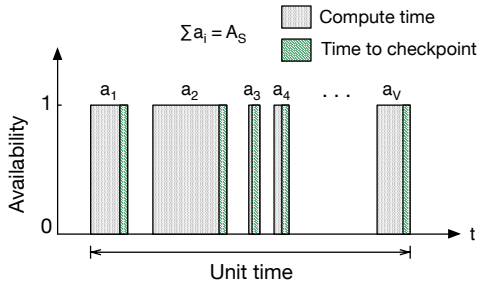
In the next section, we highlight three key metrics of transient servers, and how they affect the perceived application performance.

6.2 Transient Server Characteristics

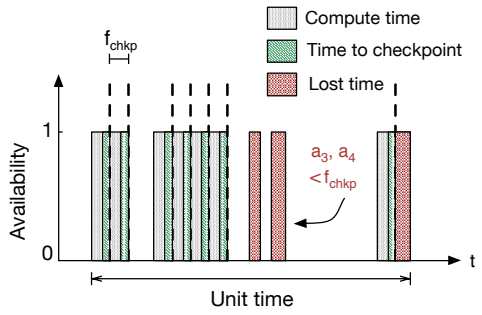
Transient servers in EC2 and GCE are significantly cheaper because they entail an unbounded risk of revocation, as platforms may revoke them at any time. Handling revocations not only introduces additional application complexity, but also additional performance



(a) Available, Not Volatile, Predictable



(b) Available, Volatile, Predictable



(c) Available, Volatile, Not Predictable

Figure 6.1: Availability, volatility, and predictability affect transient server performance

overheads, which decrease transient server performance. While applications can reduce this overhead, given sufficient knowledge of revocation characteristics, *they cannot eliminate it*. Thus, transient server performance is strictly less than on-demand performance. We distill the revocation characteristics that influence performance into three independent metrics: availability, volatility, and predictability. Below, we discuss these metrics in the context of EC2, as GCE releases no information on, and provides no control over, them.

- **Availability** is the percentage of time a transient server is available—in EC2, this translates to the percentage of time the spot price is below a user's bid price.
- **Volatility** is the frequency of transient server revocations—in EC2, this translates to the frequency at which the spot price rises above the user's bid price.
- **Predictability** is the stationarity in the distribution of revocations over time—in EC2, it is the frequency at which the mean and variance of spot price time-series changes.

Figure 6.1 illustrates, in the context of our simple batch job that these three metrics are distinct from, and independent of, each other. Figure 6.1(a) shows a time-series of transient server availability that is not volatile and highly predictable. In this case, there is only a single revocation at a well-known time. As a result, the application need only checkpoint immediately before the revocation occurs, thereby minimizing its overhead (in green) and maximizing the useful work it performs (in grey). In contrast, Figure 6.1(b) shows a similar time-series with the same availability over time, but with a higher volatility that includes many revocations. In this scenario, the application incurs more overhead (in green) than before because it needs to checkpoint much more frequently. However, since the time of each revocation remains well-known and predictable, it still need only checkpoint immediately prior to each revocation. Finally, Figure 6.1(c) shows a time-series again with the same availability, but with a high volatility and low predictability. Here, the application incurs a higher checkpointing overhead (in green), since it does not know precisely when revocations will occur, and must instead periodically checkpoint at a fixed interval. In this case, the application also incurs some recomputation overhead (in red) when it loses work after an unexpected revocation.

Our simple example illustrates that volatility and predictability affect transient server overhead and performance much more than availability. Despite this, prior work focuses largely on optimizing availability in EC2—by determining the bid that minimizes cost, while allowing an application to satisfy a performance target, e.g., a deadline [99, 49, 96, 81] or specified availability [35]. However, we contend that *there is no reason to ever wait for a particular transient server to become available*, since cloud platforms are large enough that resources are effectively always available somewhere (at some price). This has been corroborated by recent work [68, 37, 78, 42, 66] as well.

As a result, volatility and predictability—and not availability—are the critical metrics that affect transient server overhead and performance. Prior work likely does not focus on these metrics because EC2's current spot market is predictable and not volatile—prices generally remain low and stable for long periods. However, as more users exploit the spot market's arbitrage opportunities (by using spot instances when the spot price is low, and migrating to on-demand instances when it rises), spot prices will not only rise, but also

become more volatile and less predictable. This will ultimately decrease the performance and value of using spot instances [79].

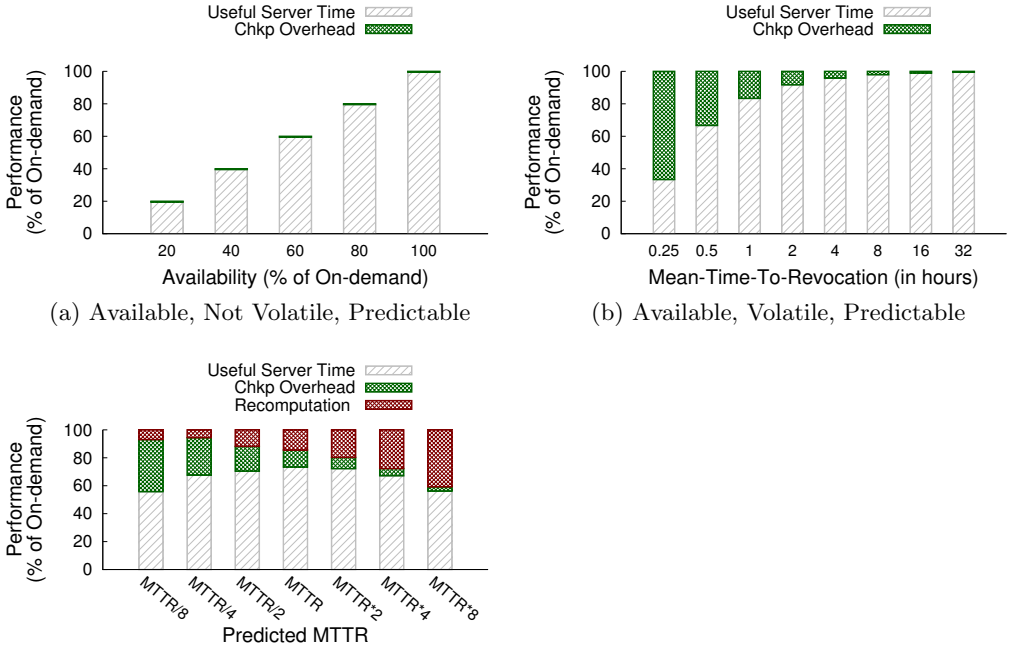
6.2.1 Quantifying the Performance Impact.

We quantify the performance impact of revocation characteristics for HPC-oriented batch applications that use checkpointing to handle server revocations. We focus on batch applications, since these are commonly run on transient servers [60, 83, 58]. In addition, we assume these applications have non-trivial memory footprints that prevent dynamically checkpointing memory state after a platform notifies a server of impending revocation, but before server termination. Current revocation warnings for transient servers range from thirty seconds (on GCE) to two minutes (on EC2), which prevents such dynamic checkpointing once memory footprints exceed 1GB to 4GB, respectively. Note that the current trend is towards shorter revocation warning times, as platform's are placing a higher priority on short re-provisioning and booting times [18]. Platform's revoke transient servers to reallocate them to satisfy other higher-priority requests, e.g., for on-demand and reserved instances. As a result the re-provisioning and boot times for on-demand and reserved instances must be greater than the warning time. Thus, platform's cannot arbitrarily increase their warning time to accommodate the dynamic migration of applications after a warning without increasing the boot times for high-priority on-demand and reserved instances.

Thus, applications with non-trivial memory footprints must employ checkpointing to ensure forward progress and prevent restarting from the beginning after each revocation. Based on prior work [29, 65], the optimal checkpointing interval that minimizes application running time when accounting for the overhead of recomputation and checkpointing is below.

$$t_{opt} \sim \sqrt{2 * \delta * MTTR} \tag{6.1}$$

Here, δ is the time to write each checkpoint and MTTR is the mean-time-to-revocation (assuming that the inter-arrival time of revocations is exponentially distributed). Thus, every t_{opt} interval, the application must pause and spend δ time writing a checkpoint of its memory state to a remote disk.



(a) Available, Not Volatile, Predictable

(b) Available, Volatile, Predictable

(c) Available, Volatile, Not Predictable

Figure 6.2: Impact on transient server performance when (a) varying availability, (b) varying volatility at a given level of availability, and (c) varying predictability at a given level of availability and volatility.

Figure 6.2 then shows how the performance of transient servers (as a fraction of on-demand server performance) varies based on availability, volatility, and predictability. Here we model the transient server revocations as following a Poisson process with a specified MTTR over a two week period. We then simulate running a batch job on a transient server with a 16GB memory footprint, which incurs a checkpointing overhead of ∼10 minutes on EC2 using an EBS magnetic disk. We consider 16GB a medium-sized memory footprint, as we assume the use of systems-level mechanisms that checkpoint the memory of an entire server. Currently, 28 of the 40 instance types offered by EC2 have >=16GB memory. Of course, a larger memory footprint would increase the checkpointing overhead resulting in a larger δ. In this example, since we use system-level checkpointing of the entire memory footprint, we also assume that applications are well-matched to the transient server's memory size. If an application uses significantly less memory than a transient server offers, it should acquire a smaller (and cheaper) server. In addition, each revocation in EC2 would also incur an additional two minute overhead to acquire and boot a replacement server. As

before, we show the useful server-time in grey, the checkpointing overhead in green, and the recomputation overhead in red.

Figure 6.2(a) shows that, as expected, the percentage of time a transient server is available is linearly related to its rate of computation: if the server is only available 50% of the time, its rate of computation is at most 50% that of an on-demand server. However, as we discuss, the availability of any single transient server is not an important metric, as cloud platforms are large enough that servers of some type are nearly always available. In contrast, Figure 6.2(b) demonstrates the impact of volatility on performance over a range of MTTRs. As in Figure 6.1(b), this figure assumes revocation times are entirely predictable, and thus represents the minimum overhead of transience at each MTTR. As the figure shows, transient server performance is 35%-70% less than an on-demand server with MTTRs of 0.25-1 hour even for a modest-sized 16GB server.

Finally, Figure 6.2(c) shows the impact of unpredictability on performance. Here, we fix the MTTR at four hours, but assume the precise revocation times are not known. We then plot the overhead due to checkpointing and recomputation for different estimates of the unknown MTTR. As the figure shows, even with a correct MTTR, the overhead increases by more than 6× compared to Figure 6.2(b) where the revocation times are known (from ∼4% overhead in (b) to ∼27% overhead in (c)). Thus, unexpected server revocations can incur substantial overhead. The figure also shows that inaccurate MTTR estimates further increase this overhead. Note that GCE releases no information on MTTR to guide users, while EC2's price history (which indirectly reveals historical MTTRs at different bid levels) does not necessarily guarantee future performance.

6.3 Transient Guarantees

A transient guarantee is the simple idea of providing users a probabilistic assurance on a transient server's availability, volatility, and predictability. While many variants of transient guarantee are possible, we propose a variant that provides a probabilistic guarantee by specifying a transient server's MTTR. Providing a probabilistic guarantee on the MTTR has two advantages: (i) it does not impose strict limits on a platform's freedom to revoke servers, as the MTTR need only converge to a particular value across many requests (ii)

the overhead of many fault-tolerance mechanisms, including the optimal checkpointing [29], is typically defined with respect to MTTR.

Of course, we could define much stronger transient guarantees to enable even higher performance. For example, EC2 introduced spot block instances in October 2015, which guarantee access to a transient server for a fixed block of time between 1 and 6 hours. Thus, EC2 promises a spot block instance will be revoked with 100% probability at the end of each block but not before. While spot (and preemptible) instances are 50-90% cheaper (in an absolute sense) than on-demand servers, spot blocks are typically only 30-45% cheaper [14]. Based on our analysis in Figure 6.1(a), since spot block revocations are predictable, they require less fault-tolerance overhead (and have higher performance) than spot instances, as applications need to checkpoint only once, immediately prior to the revocation. However, spot blocks also impose greater restrictions on a platform's freedom to revoke.

In effect, the more accurately a platform can predict the future supply of its idle capacity, the stronger the transient guarantees it can offer users. For example, if EC2 could perfectly predict the precise times of all future requests for on-demand and reserved servers (and when users would release them), it could simply offer spot block instances to exactly fill any idle time. As mentioned, these spot block instances are much more valuable (and cost more) than current spot instances. Of course, platforms are likely not able to predict their future demand with such precision. As a result, EC2 typically offers only a small number of spot block instances (for short time windows), likely when they are near 100% certain they will not need to revoke them within the window. However, we expect platforms are better able to forecast the statistical attributes of their future demand, e.g., its distribution, mean, and variance. For example, recent work develops prediction techniques to accurately forecast the percentage of idle server capacity, i.e., not allocated to on-demand or reserved servers, over multi-month periods for multiple production Google clusters [26]. Transient guarantees assume that platforms can accurately estimate the MTTR of transient servers based on the distribution of demand for on-demand (and reserved) servers.

We envision platforms offering transient servers with transient guarantees for a fixed price, similar to GCE's model for preemptible instances. Note that platform's could also offer servers with transient guarantees for a variable spot price. However, fixed pricing is

	Volatility	Predictability	Pricing
GCE Preemptible	Unknown	None	Fixed
EC2 Spot	Unbounded	Weak	Market-based
Transient Guarantees	Probabilistic	Probabilistic	Fixed

Table 6.1: Approaches to selling idle cloud capacity

simpler for users to budget than EC2's variable priced bidding model because users know the actual price in advance (and not just the maximum possible price). Since EC2 charges users based on the variable spot price, and not their bid price, users do not know the cost of transient servers *a priori*. Fixed pricing also makes decision-making for users much simpler, as they do not have to monitor, analyze, and predict prices across thousands of markets to select an optimal market and determine an optimal bid. Table 6.1 summarizes the differences between our MTTR-based transient guarantees, GCE preemptible, and EC2 spot instances.

6.3.1 Equilibrium Price.

An advantage of transient guarantees is that they enable users to quantify transient server value. We define a transient server's maximum value in relation to its amortized performance compared to on-demand servers after accounting for the overhead of revocations, e.g., checkpointing, migration, and recomputation. That is, if a transient server with high volatility and low predictability incurs a 25% overhead for checkpointing, migration, and recomputation, then we say its value is 25% less than an equivalent on-demand server. We call this the transient server's *equilibrium price*: where the price per unit of useful time (modulo overhead) between a transient server and an on-demand server is equal. Rational users should never pay more than it for a transient server, as it provides no discount.

Based on our analysis, we can derive the equilibrium price for a batch application in terms of its volatility, checkpointing overhead, and the price of an equivalent on-demand server. The expected completion time $E[T_j]$ for an application j with running time T_j on a transient server is below. Here, the first term is the application's actual running time, the second term is the additional overhead from checkpointing (at the optimal frequency),

which incurs δ overhead at every checkpoint interval, and the last term is the expected recomputation overhead across all revocations (assuming the probability of revocation at any time during each interval is equal). Thus, if an equivalent on-demand server costs p_o, then the transient server's equilibrium price p_{eq} is:

$$\begin{aligned} E[T_j] &= T_j + \frac{T_j}{t_{opt}}\delta + \frac{T_j}{MTTR} * \frac{t_{opt}}{2} \\ p_{eq} &= p_o * \frac{T_j}{E[T_j]} \end{aligned} \tag{6.2}$$

Note that platforms should offer transient servers for a discounted price that is strictly less than their equilibrium price, as the equilibrium price reflects the point at which transient servers offer no savings relative to on-demand servers. The magnitude of the discount represents the size of the arbitrage opportunity that exists for using transient servers.

6.3.2 Transient Classes.

Current platforms not only provide no guarantees on revocation characteristics, they also offer only a single class of transient servers. Transient guarantees permit platforms to define multiple service classes with different strength guarantees. Offering multiple service classes has two advantages:

- **Multiple Choices**. It offers users multiple choices at different price and performance/risk levels. For example, important, but non-critical, applications might be willing to tolerate a few interruptions, e.g., with a high MTTR, in return for a slightly lower price compared to on-demand servers. However, less important background tasks may be willing to tolerate more frequent interruptions, e.g., with a low MTTR, in return for a much lower price.
- **Accurate Revocation Characteristics.** It also enables platforms to reduce the aggregate overhead incurred by transient servers (and increase their aggregate value) by more accurately specifying the revocation characteristics in each class, enabling users to better tune fault-tolerance mechanisms for servers in each specific class.

To illustrate, consider Figure 6.3, which depicts the idle capacity over time after servicing on-demand requests for a platform with a total capacity of N servers. This idle capacity

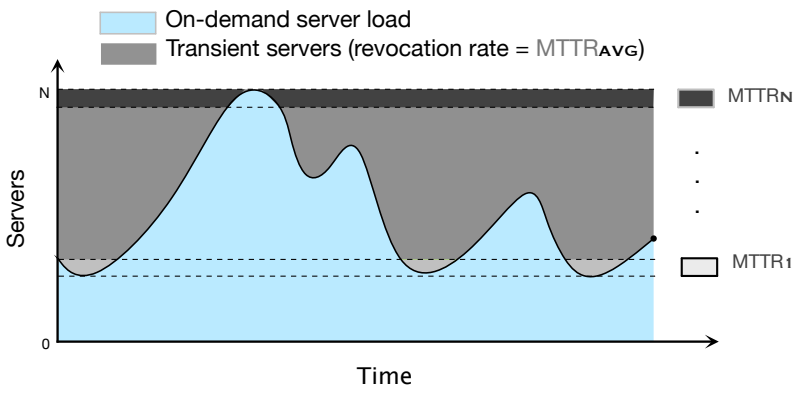


Figure 6.3: Platforms may offer their idle capacity as multiple transient classes with different transient guarantees

can be offered as a single class of transient servers with $MTTR_{avg}$ based on the average revocation characteristics across all idle servers. However, notice that if we allocate on-demand requests with servers 0 to N in order, higher ranked servers experience fewer revocations than lower ranked servers. Thus, a platform could carve out the top N^{th} server to be allocated and offer it as a separate class with $MTTR_N \gg MTTR_{avg}$. Since the N^{th} server's MTTR is much longer than the average, it is more valuable to applications: it experiences fewer revocations, incurs less overhead, exhibits higher performance, and has a higher equilibrium price. In contrast, offering the N^{th} server as part of a single class with $MTTR_{avg}$ significantly undersells its true value, since its actual revocation characteristics are much better than average.

In the extreme, to maximize aggregate transient server performance and value, platforms would offer each individual server as a separate class with a unique MTTR that precisely captures its revocation characteristics. As per Equation 1, this minimizes the fault-tolerance overhead incurred by each transient server, thereby maximizing the performance and value of the entire transient server pool. Of course, this extreme is not feasible, as it would result in thousands of classes, each composed of a single server. However, offering only a single class reduces aggregate performance by treating the most available servers similarly to the least available ones. Thus, to mind this gap, we define a small number of classes such that they approach the optimal performance and value. In addition, defining a small number

of transient classes, each consisting of many servers, enables providers to more accurately estimate the aggregate MTTR of each class [26].

We assume the number of idle servers ranges from $[0, N]$ at any time t. The platform then partitions N servers into k classes with each class having M_j servers, such that $\sum_{j=0}^{k} M_j = N$. As in Figure 6.3, we assume a strict ordering of servers with server N having the fewest revocations and server 0 having the most. Thus, higher numbered classes have higher performance than lower numbered classes. We then offer each class with a transient guarantee specifying the average MTTR for transient servers within that class. Since composing the optimal set of k classes that minimize overhead requires examining all $\binom{N}{k}$ combinations classes (with complexity $O(n^k)$), we define two simple heuristic policies.

- **Equal-Split Policy.** Our equal-split policy naïvely divides the idle server capacity into k equal-sized classes, such that each class has the same number of N/k servers.
- **Greedy-Split Policy.** In contrast, our greedy-split policy iteratively composes classes as follows. The policy starts with only the most available server N in the first class. The policy then proceeds iteratively by adding the next most available server $N-1$ to the first class, and then determines whether the addition of the server increases the aggregate value across all servers in the class. The aggregate value is computed as the number of servers M_j in the class multiplied by the equilibrium price p_e of servers in the class, where the equilibrium price is computed based on the average MTTR across all servers in the class.

Thus, a tradeoff exists when greedily adding each additional server (with lesser availability) to a class: adding a new server increases the total number of servers M_j in the class, but it decreases the class' aggregate MTTR and thus the equilibrium price of all servers p_e in the class. As a result, the greedy-split policy proceeds by adding servers to the first class in order of their availability (from most to least) until adding the next server decreases the overall value of the class. The greedy-split policy then defines a new class, and proceeds in the same fashion. The policy stops once it has defined k classes, or there are no more servers to add.

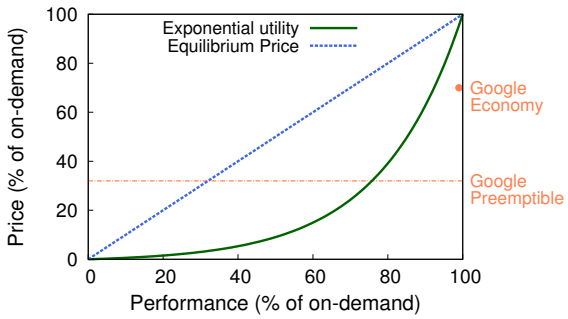


Figure 6.4: Utility functions that specify an offered price for a transient server with a transient guarantee.

6.3.3 Transient Server Pricing.

We assume that transient servers are priced such that they are never idle, i.e., their price is always low enough to attract saturating demand. Note that our analysis in Section 6.3.1 only derives an equilibrium price, which represents the *maximum* amount a user should be willing to pay for a transient server. In practice, transient servers should be discounted from this maximum price. For example, recent work defines an economy class of on-demand servers for Google that are >98.9% available (instead of near 100%), but proposes selling them for only 70% of the on-demand price [26]. Thus, they discount these servers 30% due to only a slight reduction in availability.

Prior work on defining utility functions for real users indicates a similar steep dropoff in utility for the initial degradation in performance [47, 90]. Thus, we adopt a similar exponential utility function for estimating the offered price of transient servers with different MTTRs. Figure 6.4 plots our exponential utility function, where the offered price (as a percentage of the on-demand price) on the y-axis is a function of transient server performance on the x-axis (based on the MTTR). The exponential utility function[1] captures the steep drop-off in price once servers are not 100% available. We also plot the current price-points for the Google economy class described above and GCE preemptible instances, which cost ∼70% of the on-demand price for all servers. Finally, we plot the equilibrium price, which is

[1]Exponential utility is derived from the function $y = 101^{(x/100)} - 1$.

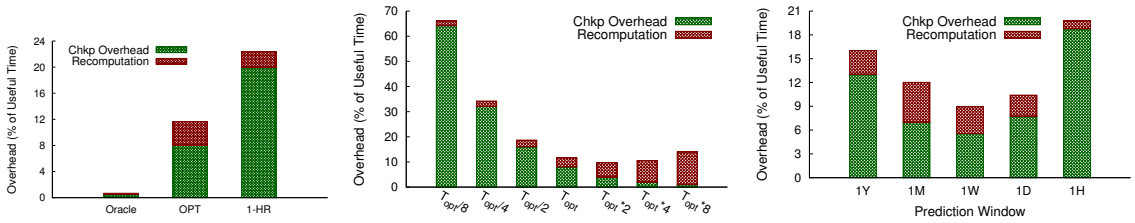
simply the line y=x. The difference between our utility function and the equilibrium price is the normalized discount from using transient servers.

We use the utility function above in the Evaluation section to estimate the potential revenue from offering transient guarantees. Note that, to verify transient guarantees, large-scale users can average their performance across a large number of requests. Ensuring small-scale users can verify transient guarantees poses a more challenging problem. While such verification is outside the scope of this paper, crowd-sourced techniques [86] are a promising direction.

6.4 Implementation

We implemented a cluster simulator in python to evaluate the performance improvements that transient guarantees offer. The simulator takes a fixed server capacity as input (where each server has a specified memory size), as well as a trace of requests for on-demand servers. Each request specifies a job to run that includes the number of servers the job requires, job submission time, and job duration. We assume any excess capacity is then allocated to the pool of transient servers. Each transient server incurs an overhead by periodically checkpointing its memory state. We assume that transient guarantees specify an MTTR, which transient servers use to compute the optimal checkpointing interval (based on t_{opt}). Our simulator implements the equal-split and greedy-split algorithms.

The simulator is designed to operate on publicly-available job traces from a production Google cluster [61]. The trace contains job characteristics, server configurations, and scheduling decisions on a cluster of 12.5k servers over a 29-day period. We make a few simplifying assumptions using this job trace in our evaluation. First, we normalize the heterogeneous servers in the trace based on the smallest server type, and assume a cluster with homogeneous servers. If a server runs multiple jobs concurrently, we assume it runs multiple VMs. We also rank cluster nodes from 1 to N, and schedule jobs in rank order (as opposed to following Google's scheduling decisions), such that if k jobs are active, they occupy servers 1 to k. These assumptions enable the simulator to avoid trace-specific scheduling, bin-packing, and cluster management decisions that are not central to evaluating transient guarantees.



(a) Baseline Policies (b) Checkpoint Interval (c) Inaccurate Predictions

Figure 6.5: Impact on spot server performance due to incorrect MTTR characterization

6.5 Evaluation

The goal of our evaluation is three fold. First, we examine the overhead and performance degradation due to revocations in EC2 spot markets. Next, we analyze the demand pattern of a production Google cluster trace [61] to quantify the characteristics of its idle capacity. Finally, we compare the benefits of transient guarantees vis-a-vis the current approaches of EC2 and GCE in pricing and valuing the resulting idle capacity.

6.5.1 EC2 Spot Instance Performance.

We evaluate spot instance performance for a representative EC2 spot market—the `m1.large` instance type running Linux in one availability zone of the U.S. East region over 2014. Note that we focus on EC2, since GCE offers no information on the revocation characteristics of preemptible instances. The `m1.large` market represents one of the most popular configurations of the most popular instance types in the most popular region of EC2. In this analysis, we assume a bid equal to the on-demand price since users can nearly always switch to using on-demand servers if their spot instances are revoked. Based on spot price data from 2014, we observe 555 revocations over the year with an MTTR of ∼15 hours. We assume a modest-sized memory footprint of 16GB, which takes ∼10 minutes to checkpoint and restore based on our benchmarks using EBS magnetic disks.

Figure 6.5a plots the overhead of spot instances in the `m1.large` market for three different fault-tolerance policies: an oracle policy that minimizes overhead by checkpointing immediately before each revocation, a periodic policy that checkpoints at the optimal (OPT) periodic interval [29] ($t_{opt} \sim 2$ hours in this case) assuming the MTTR is known, and a static

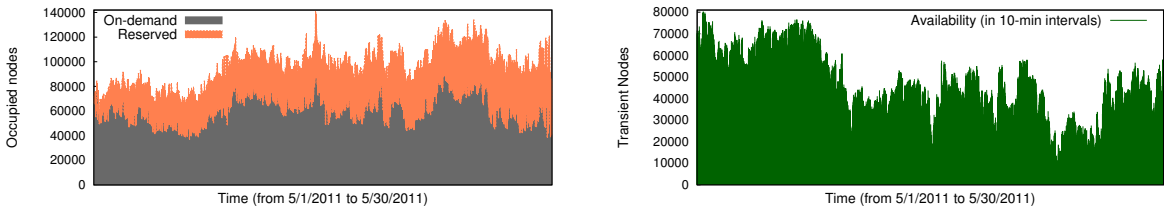


Figure 6.6: Transient servers resulting from the idle capacity in Google cluster traces

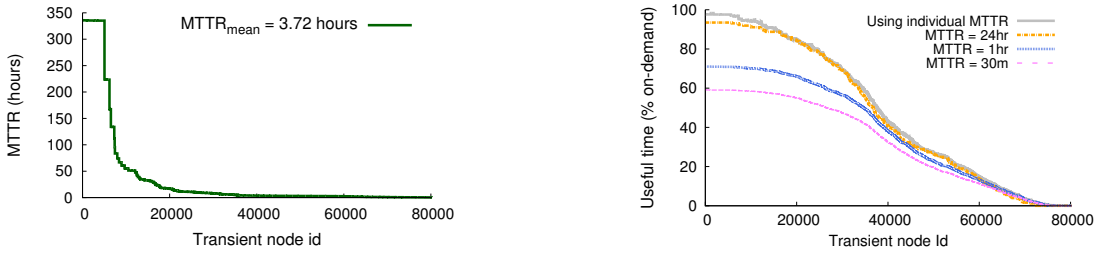
policy that checkpoints once each hour. The latter policy is proposed in prior work, since EC2 bills on an hourly basis [84]. The figure shows that the static per-hour checkpointing consumes 24% of the useful server-time, and the optimal periodic policy consumes 12% of the useful server-time. While the oracle consumes less than 1% overhead, it is not viable in practice as future demand is not precisely known.

Figure 6.5b then shows the impact on overhead of incorrectly setting the checkpointing frequency when computing the optimal periodic checkpointing interval from Figure 6.5a. In this case, the optimal checkpointing frequency is near $2 * t_{opt}$, since the optimal formula is only a first-order approximation and incorrectly assumes revocation interarrival times are Poisson distributed. However, recall that with EC2, users actually do not know future revocation characteristics. The graph shows that selecting a checkpointing frequency too short can result in significant additional overhead that further reduces performance. Finally, Figure 6.5c shows the impact of mispredicting the revocation rate on overhead. In this case, we use the optimal periodic checkpointing interval, but where we compute the MTTR based on spot price history over different size past windows, e.g., the last hour, day, week, month, and year. The graph shows that overhead varies widely (from 9% to 21%) depending on the prediction window we select.

Result. *Though the current EC2 spot markets are highly stable, their limited volatility already reduces the performance of spot instances by 9-24% relative to on-demand servers.*

6.5.2 Origin and Characteristics of Idle Cloud.

Using the Google cluster traces, Figure 6.6(left) plots the server allocation pattern for two classes of jobs – high-priority (akin to requests for reserved instances) and low-priority (akin to requests for on-demand instances). The peak server capacity required for executing



(a) MTTR Distribution

(b) Performance

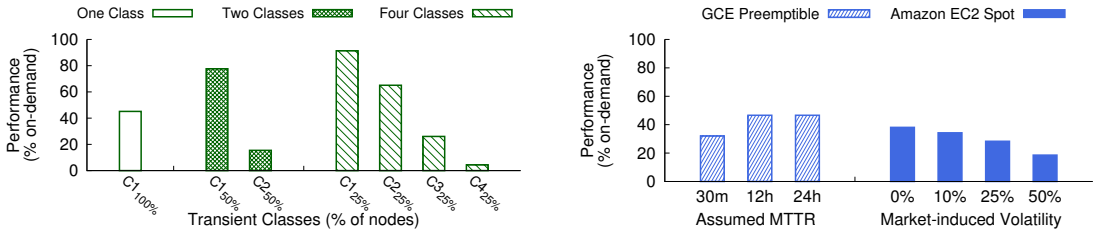
Figure 6.7: Revocation and performance characteristics of transient servers in 6.6

both job classes is ∼141k VMs. Thus, the white space at the top of the graph indicates the varying amount of idle server capacity that is available to offer as transient servers. Figure 6.6(right) then plots the idle server capacity over time, where only those servers that are unused for at least 10 minutes are considered. The availability of idle capacity ranges from 0 to ∼80k VMs, where we assume each server has 16GB of memory.

Figure 6.7a then shows the mean-time-to-revocation (MTTR) for each transient server in the cluster. While the average MTTR across all transient servers is 3.72 hours, we note that the top 10% of servers have an MTTR >84 hours and the top 50% have an MTTR >17 hours. Thus, as discussed earlier, a large fraction of transient servers experience much longer periods of availability than reflected in the average MTTR.

Next, we derive the maximum amount of useful server-time (modulo fault-tolerance overhead) for each transient server, assuming that we offer each server based on its own unique MTTR. We also plot the useful server-time assuming an MTTR of 24 hours, 1 hour and 30 minutes across all transient servers. Figure 6.7b shows that, in this case, using a MTTR of 24 hours achieves near the optimal useful server-time, while using a MTTR of 30 mins reduces the useful server-time by ∼40% across all servers in the cluster. Since only 7% of transient servers have an MTTR in the range of 30 minutes, this results in excessive checkpointing overhead for the vast majority of servers.

Result: *Since idle capacity varies over time, transient servers exhibit a wide range of characteristics. Checkpointing transient servers based on incorrect revocation characteristics*



(a) Transient Guarantees

(b) GCE (left) and EC2 (right)

Figure 6.8: Performance of transient servers under different pricing models

results in significant performance losses (up to ∼40%), especially for the least volatile (and most valuable) servers.

6.5.3 Transient Guarantees.

Figure 6.8a next quantifies the benefit of partitioning transient servers into multiple classes with different transient guarantees. This graph employs the equal-split policy to partition transient servers into 1, 2, and 4 classes. In each case, the y-axis quantifies the average useful time of a server in each class (modulo checkpointing overhead). The graph demonstrates how separating transient servers into different classes enables platforms to offer differentiated quality-of-service for different transient servers. We see that, as we offer more transient classes, the increase in the performance of higher classes is significantly more than the decrease in the performance of lower classes, thereby increasing the overall performance of the cluster. For example, moving from a single class configuration to a two class configuration results in an overall decrease in fault-tolerance overhead across all transient servers of 13.5%. This reduction in overhead is due to more accurately specifying revocation characteristics in multiple classes.

Figure 6.8b then compares the performance of transient guarantees with the current approaches used by GCE and EC2. Since GCE does not reveal any information about preemptible instances, we consider three distinct fixed-interval checkpointing policies that assumes a MTTR of 30 minutes, 12 hours, and 24 hours. For EC2, we use the commonly employed one hour checkpointing strategy [84]. Since EC2's spot market is strictly more

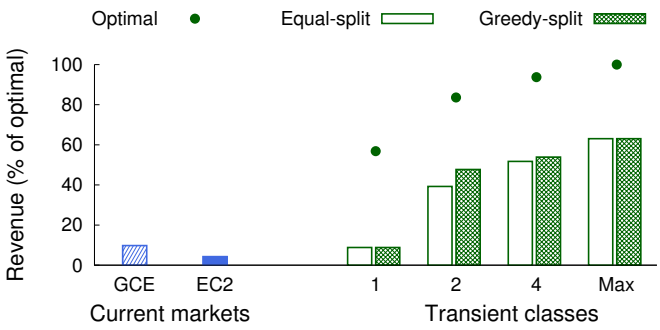


Figure 6.9: Revenue comparison from selling transient servers

volatile than our approach, we add differing levels of market-induced volatility to the Google job trace for EC2. The graph shows that the performance of GCE and EC2 is less than that with transient guarantees. GCE's performance (assuming a 24 hour MTTR) is near that of offering a single class with transient guarantees, but has 13% higher overhead than offering two separate classes. EC2 also performs worse than a single class with transient guarantees, largely because of the additional market-induced volatility. For example with an additional 25% volatility, its performance is 28% less than using transient guarantees with a single class.

Finally, we evaluate the potential increase in revenue from offering multiple classes of servers with transient guarantees compared to GCE and EC2. For this graph, we use the exponential utility function [73] to assign prices to transient servers based on their performance. Figure 6.9 then shows the aggregate revenue from selling GCE preemptible instances, EC2 spot instances, and using transient guarantees at these prices. The dot represents the optimal revenue if transient servers were sold at their equilibrium price. In all cases, we assume saturating demand, such that all transient servers are sold. The y-axis quantifies the revenue as a % of the maximum, where every transient server is priced at its equilibrium price, for each approach on the x-axis. The maximum number of transient classes represents the optimal where each transient server has its own class and revocation characteristics. For GCE and EC2 we select the top performing configuration from Figure 6.8. The figure shows the potential revenue of offering multiple classes of transient servers.

In this case, using the optimal maximum number of transient classes achieves 6.5× more revenue than GCE and 14× more revenue than EC2. In addition, partitioning transient servers into only two and four classes brings the revenue to within 25% and 15% of the optimal maximum, respectively. This result stems from the fact that most of the value of transient servers derives from the servers with the lowest volatility. Thus, selling them separately at a higher price yields significant gains. Thus, while offering each transient server as its own class is not viable, offering two or four classes of transient servers is reasonable and can offer significant benefits. In addition, the greedy-split partitioning policy yields slightly better revenue that the equal-split policy in all cases, e.g., adding 20% revenue when offering two classes.

Result: *Partitioning servers into just four classes increases revenue from transient servers by ∼6.5× compared to GCE and EC2, and comes within 15% of the optimal revenue.*

6.6 Related Work

Prior work focuses on optimizing existing offerings of transient servers from a user's perspective. For example, there is substantial prior work on analyzing EC2 spot price characteristics [22, 41], designing optimal bidding policies [96, 81, 99], and modifying particular applications to gracefully handle transient servers using fault-tolerance mechanisms, such as checkpointing [84, 78]. Our work differs from these work in that we propose a new service contract that maximizes the performance of transient servers for users, while still allowing platforms the freedom to revoke transient servers when necessary.

Related work that takes a similar platform-centric perspective proposes offering a new economy class of on-demand servers that have slightly lower availability (>98.9%) [26]. The work analyzes data from multiple Google production clusters and shows that a large fraction of servers (6.7-17.3%) are idle with high probability for long multi-month time periods. On similar lines, Amazon introduced Spotblocks [14], a new contract type for servers that come with a predefined fixed block of time (1-6 hours) for higher price than regular spot servers. Our work generalizes and expands upon these ideas by defining the concept of a transient guarantee, and then showing how to partition idle capacity into an arbitrary number of transient server classes to maximize the performance of transient servers. Importantly, we

also identify the relationship between a transient server's performance and its volatility and predictability, and define its equilibrium price to capture its value relative to an on-demand server.

Finally, researchers at Princeton have proposed (i) Availability-knob [63], an incentive scheme where users can specify the desired availability and be charged accordingly, and (ii) Graceful degradation [62], a self-adaptive technique to enable cloud applications to respond to supply variations at the cloud provider. While our work shares the high-level goal of increasing the datacenter utilization, it differs at both system and policy levels. For example, transient guarantees do not require any modifications to the user applications; nor do they require real-time interactions between users and providers.

6.7 Conclusion

Since transient servers are a new concept and are not widely used, there remains an opportunity to experiment with their terms and pricing. We show that the current terms offered by EC2 and GCE limit the useful performance that users can extract from transient servers. We propose transient guarantees to maximize their performance and value, while still allowing platforms to revoke servers when necessary. We analyze the performance and cost benefits of transient guarantees for batch applications. We show that the aggregate revenue could increase by up to $\sim$6.5$\times$ when selling transient servers through transient guarantees than through the current market mechanisms of EC2 and GCE. Thus, transient guarantees may represent a better way to offer and consume transient servers.

Status. Transient guarantees have been evaluated on Google cluster traces via simulation. Additional details on its design, implementation and evaluation are in [73].

CHAPTER 7

CONCLUSIONS

> "How lucky I am to have something that makes saying goodbye so hard."
>
> Winnie-the-Pooh

Cloud platforms sell computing capabilities to applictions for a price. However, by finely controlling the characteristics of their offerings, the cloud providers expose applications to several risks. These cloud risks incur implict costs and potential losses for applications running on the cloud. Our work elevates risk management in cloud computing plaforms to a first-class design principle. In this dissertation, we identify and quantify these risks, and then develop system-level support to transparently optimize their costs to the applications. In what follows, we summarize our contributions, and explore avenues for future work.

7.1 Summary of Contributions

Our goal is to identify the cloud risks, and build system support to manage them. In doing so, we observe its similarities to the risks encounted in the financial and commodity markets, which in turn enable us to adapt and extend risk management concepts from these domains to cloud computing. Guided by this new approach dubbed *financializing cloud computing*, we make four significant contributions.

First, we mitigate the revocation risk with insurance. Revocations pose a new fault-model that undermines the benefit of using transient cloud servers. Unlike the classical hardware failure, transient server revocations are intentional, frequent and come with advanced warning. In SpotOn [78], we investigate how to insure against revocation risk without incurring huge premiums.

We identify that the cost of insurance against transient server revocations is a function of application's footprint, transient server's market characteristics, and fault-tolerance mechanism's overhead. In order to make this cost-efficient, we (i) extend the classical fault-tolerance mechanisms of migration, checkpointing, and replication to the new fault scenario and model their overheads; and then (ii) design a greedy insurance policy that dynamically selects a combination of transient server and fault-tolerance mechanism that results in the lowest premium. We implement these in a service called SpotOn, which executes unmodified batch applications on EC2 spot servers. Evaluations on Amazon EC2 show that SpotOn is able to achieve near on-demand level performance while also realizing ∼91% cost savings.

Second, we reduce the price risk through active trading. Transient cloud servers sold in variable priced markets, like EC2 spot markets, exhibit price variations, inversions and arbitrages. These lead to price risk, or the risk that a chosen server's price will increase relative to others. In HotSpot [70], we explore how to avoid price risk for unmodified cloud applications.

Through market analysis, we observe (i) that price risk is ∼500x more frequent than revocation risk, and (ii) that servers with high discount also tend to have low revocation risk (which is reflective of the supply-demand dynamics). With these key observations, we hypothesize that by employing active trading (i.e., hopping from the current server to a better one), a flexible application can reduce its costs without increasing its revocation risk. Then, we design a server hopping mechanism at the system level (via a self-migrating container) such that unmodified applications can utilize it. HotSpot, our prototype on Amazon EC2 demonstrates that active trading results in ∼50% savings relative to insurance based approaches.

Third, we eliminate the uncertainty risk by index tracking. Applications that run on variable-priced cloud servers suffer from cost uncertainty. Since the server prices are market-based, and could vary considerably (up to 10×), customers find it difficult to plan their IT expenses. In [72], we design an index-tracked cloud server to eliminate the uncertainty risk.

While prior approaches have tried to model and predict individual transient server markets, they have had limited success due to the proliferation of EC2 spot markets. We propose

an alternative solution based on two key insights: (i) making price predictions at aggregate market level is more reliable than at individual server level, and (ii) knowing the benchmark for cost estimates a priori enables reactive server management systems to achieve the target cost-efficiency without sacrificing availability. Towards eliminating cost uncertainty, we introduce a market-based cloud index, and design a mechanism for index-tracking via server hopping. We implement and evaluate this system on Amazon EC2 spot markets, and demonstrate that it can reliably achieve the predicted cost-efficiency for a broad class of flexible applications.

Finally, we minimize the valuation risk via asset pricing. For transient cloud server offerings, the providers do not reveal precise transiency information as it makes their administration challenging. However, this opacity makes it difficult for consumers to gauge their true value. In Transient Guarantees [73], we explore how to minimize the valuation risk.

By distilling transient server characteristics into three orthogonal axes of availability, volatility, and predictability, we introduce the notion of equilibrium price—i.e., the price beyond which the utility of a transient server (modulo its fault-tolerance overhead) is no better than an equivalent on-demand server. While equilibrium price is only applicable in retrospect, it helps consumers determine how their transient server fared. Interestingly, our market analysis using equilibrium price reveals that Amazon and Google transient server contracts do not maximize the server's value for either providers or consumers. To address these problems, we design a new asset-pricing abstraction called transient guarantee that offers probabilistic assurances on transiency characteristics. Through modeling and evaluation, we show that transient guarantees not only help users in determining the value of transient servers upfront but also enable providers to increase their revenue by up to 5× without sacrificing their ability to revoke transient servers.

7.2 Directions for Future Research

Our dissertation focused on four key cloud risks, and on transparently managing them for specific cloud applications. Here, we identify several directions to extend our work.

- **Managing Additional Risks.** A natural direction for future work would be to identify additional risks and extend the system support framework to incorporate those. Chapter-2 identified a number of risks including rejection-, utilization-, and performance-risks. This line of enquiry leads to several interesting yet unanswered questions: (i) Is it possible to *hedge* risks by combining cloud servers of different contract types? (ii) How to extend the risk management framework to newer types of risks arising from an evolving set of contracts?
- **Handling Different Applications.** Another intuitive set of extensions would be to support different classes of applications. The main challenge here is to automatically infer the behavior and utility function of application categories such that cloud risks could be quantified at the system-level. For example, in chapter 6, we designed *transient guarantees* with a focus on batch applications that could be checkpointed. However, the mechanism of deriving equilibrium price can be extended to other applications with different characteristics for e.g., latency-senstive applications, distributed frameworks, and replicated services.
- **Risk management beyond IaaS.** We observe that the diversity and asymmetry in the creation and consumption of compute resources is not only on the rise but also getting more explicit. So, while our focus here has been on the IaaS cloud servers, risks exist at different levels of cloud services. For example, the emerging interface of serverless computing or Function-as-a-Service poses risks in a different context. On the provider side, it exemplifies the utilization risk since a fixed set of servers are required to execute a variable workload subject to performance constraints. On the user side, it exacerbates valuation- and rejection-risks since cloud resources are even more abstracted out compared to IaaS.
- **Emerging Computing Platforms.** Cloud datacenters are experiencing many technological advancements in the recent years including disaggregated datacenters [34, 64]. While current datacenters are built as racks of individual servers, each of which tightly integrate small amounts of compute, memory, storage, and network resources, the disaggregated datacenters are built as racks of standalone resources interconnected via fast

networks. This paradigm shift will force a rethinking of system software, and we believe that *financialized system design* serves a valuable reference point in the debate.

BIBLIOGRAPHY

[1] Financialization. https://en.wikipedia.org/wiki/Financialization.

[2] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/, Accessed August 2017.

[3] AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/, Accessed October 2017.

[4] Cmpute inc. `http://www.cmpute.io`, Accessed October 2017.

[5] Linux Containers. `http://linuxcontainers.org`, Accessed October 2017.

[6] Spot Instance Product Details. https://aws.amazon.com/ec2/spot/details/, Accessed August 2017.

[7] Sara Arevalos, Fabio Lopez-Pires, and Benjamin Baran. A Comparative Evaluation of Algorithms for Auction-based Cloud Pricing Prediction. In *IC2E*, April 2016.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[9] Alvin AuYoung, Laura Grit, Janet Wiener, and John Wilkes. Service contracts and aggregate utility functions. In *HPDC*, June 2006.

[10] Charles Babcock. Amazon's 'Virtual CPU'? You Figure it Out, in *InformationWeek*, December 23rd 2015.

[11] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI*, February 1999.

[12] Jeff Barr. Amazon EC2 Beta. AWS Official Blog, August 2006.

[13] Jeff Barr. EC2 Spot Instance Termination Notices. `https://aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/`, January 2015.

[14] Jeff Barr. New - EC2 Spot Blocks for Defined-Duration Workloads. AWS Blog, October 6th 2015.

[15] Jeff Barr. Experiment that Discovered the Higgs Boson Uses AWS to Probe Nature. `https://aws.amazon.com/blogs/aws/experiment-that-discovered-the-higgs-boson-uses-aws-to-probe-nature/`, March 2016.

[16] Jeff Barr. Natural Language Processing at Clemson University - 1.1 million vcpus and EC2 Spot Instances. https://aws.amazon.com/blogs/aws/natural-language-processing-at-clemson-university-1-1-million-vcpus-ec2-spot-instances/, September 2017.

[17] Jeff Barr. Per-Second Billing for EC2 Instances. `https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/`, 2017.

[18] Alex Barrett. The Need for Speed: This Week on Google Platform. Google Cloud Platform Blog, June 2016.

[19] Luiz Barroso, Jimmy Clidaras, and Urs Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*. Morgan and Claypool Publishers, 2009.

[20] Muli Ben-Yehuda, Michael Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *OSDI*, October 2010.

[21] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. In *CloudCom*, November 2011.

[22] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM TEAC*, 1(3), 2013.

[23] Johnathan Boutelle. What to do when Amazon's spot prices spike, in *Gigaom*, December 27th 2011.

[24] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14(1), 2016.

[25] Devin Carraway. Lookbusy - A Synthetic Load Generator. http://www.devin.com/lookbusy/, Accessed October 2017.

[26] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term slos for reclaimed cloud computing resources. In *SoCC*, 2014.

[27] Louis Columbus. Roundup of Cloud Computing Forecasts 2017. Forbes, April 29th 2017.

[28] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP*, 2017.

[29] John Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. In *Future Generation Computer Systems*, volume 22, 2006.

[30] Archy de Berker, Robb Rutledge, Christoph Mathys, Louise Marshall, Gemma Cross, Raymond Dolan, and Sven Bestmann. Computations of Uncertainty Mediate Acute Stress Responses in Humans. *Nature communications*, 7, 2016.

[31] Gerald Epstein. *Financialization and the World Economy*. Edward Elgar Publishing, 2005.

[32] Paolo Falbo, Marco Fattore, and Silvana Stefani. A new index for electricity spot markets. *Energy Policy*, 38(6), 2010.

[33] Eugene Fama. Efficient capital markets: A review of theory and empirical work. *The Journal of Finance*, 25(2), 1970.

[34] Peter Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, 2016.

[35] Weichao Guo, Kang Chen, Yongwei Wu, and Weimin Zheng. Bidding for Highly Available Services with Low Price in Spot Instance Market. In *HPDC*, June 2015.

[36] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory Ganger, and Phillip Gibbons. Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets. In *EuroSys*, April 2017.

[37] Xin He, Ramesh Sitaraman, Prashant Shenoy, and David Irwin. Cutting the Cost of Hosting Online Internet Services using Cloud Spot Markets. In *HPDC*, June 2015.

[38] Botong Huang, Nicholas Jarrett, Shivnath Babu, Sayan Mukherjee, and Jun Yang. Cumulon: Matrix-Based Data Analytics in the Cloud with Spot Instances. *PVLDB*, 9(3), November 2015.

[39] David Irwin, Jeff Chase, Laura Grit, and Aydan Yumerefendi. Self-recharging virtual currency. In *EconP2P*, August 2005.

[40] David Irwin, Laura Grit, and Jeff Chase. Balancing risk and reward in a market-based task service. In *HPDC*, June 2004.

[41] Bahman Javadi, Ruppa Thulasiramy, and Rajkumar Buyya. Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In *UCC*, December 2011.

[42] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Smart Spot Instances for the Supercloud. In *CrossCloud*, 2016.

[43] S&P Dow Jones. Index Mathematics Methodology, 2014.

[44] Sunirmal Khatua and Nandini Mukherjee. Application-centric Resource Provisioning for Amazon EC2 Spot Instances. In *EuroPar*, August 2013.

[45] Cinar Kilcioglu, Justin Rao, Aadharsh Kannan, and Preston McAfee. Usage patterns and the economics of the public cloud. In *WWW*, 2017.

[46] Frederic Lardinois. Spotinst, which helps you buy AWS spot instances, raises $2m Series A. TechCrunch, March 8th 2016.

[47] Cynthia Lee and Allan Snavely. Precise and Realistic Utility Functions for User Centric Performance Analysis of Schedulers. In *HPDC*, June 2007.

[48] Qianlin Liang, Cheng Wang, and Bhuvan Urgaonkar. Spot Characterization: What are the Right Features to Model? In *SAC*, June 2016.

[49] Aniruddha Marathe, Rachel Harris, David Lowenthal, Bronis de Supinski, Barry Rountree, and Martin Schulz. Exploiting Redundancy for Cost-effective, Time-constrained Execution of HPC Applications. In *HPDC*, June 2014.

[50] Michele Mazzucco and Marlon Dumas. Achieving Performance and Availability Guarantees with Spot Instances. In *HPCC*, September 2011.

[51] Ishai Menache, Ohad Shamir, and Navendu Jain. On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud. In *ICAC*, 2014.

[52] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[53] Paul Nash. Extending per second billing in google cloud. `https://cloudplatform.googleblog.com/2017/09/extending-per-second-billing-in-google.html`, 2017.

[54] Jordan Novet. Amazon pays $20M-$50M for ClusterK, the startup that can run apps on AWS at 10% of the regular price, April 29th 2015.

[55] Bureau of Labor Statistics. The Consumer Price Index. `https://www.bls.gov/opub/hom/pdf/homch17.pdf`, 2015.

[56] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *OSDI*, December 2002.

[57] Xue Ouyang, David Irwin, and Prashant Shenoy. SpotLight: An Information Service for the Cloud. In *ICDCS*, June 2016.

[58] David Pellerin, Dougal Ballantyne, and Adam Boeglin. An Introduction to High Performance Computing on AWS. Amazon Whitepaper, September 2015.

[59] Florentina Popovici and John Wilkes. Profitable services in an uncertain world. In *SC*, November 2005.

[60] Tipu Qureshi. Cost-effective Batch Processing with Amazon EC2 Spot. AWS Compute Blog, August 2015.

[61] Charles Reiss, John Wilkes, and Joseph Hellerstein. Google Cluster-usage Traces: Format + Schema. Technical report, Google Inc., November 2011.

[62] Mohammad Shahrad, Cristian Klein, Liang Zheng, Mung Chiang, Erik Elmroth, and David Wentzlaff. Incentivizing self-capping to increase cloud utilization. In *SoCC*, September 2017.

[63] Mohammad Shahrad and David Wentzlaff. Availability knob: Flexible user-defined availability in the cloud. In *SoCC*, October 2016.

[64] Yizhou Shan, Sumukh Hallymysore, Yutong Huang, Yilun Chen, and Yiying Zhang. Disaggregated Operating System. In *SoCC*, 2017.

[65] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *EuroSys*, April 2016.

[66] Prateek Sharma, David Irwin, and Prashant Shenoy. How Not to Bid the Cloud. In *HotCloud*, June 2016.

[67] Prateek Sharma, David Irwin, and Prashant Shenoy. Portfolio-driven Resource Management for Transient Cloud Servers. In *SIGMETRICS*, June 2017.

[68] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *EuroSys*, April 2015.

[69] William Sharpe. The Sharpe Ratio. *The Journal of Portfolio Management*, 21(1), 1994.

[70] Supreeth Shastri and David Irwin. HotSpot: Automated Server Hopping in Cloud Spot Markets. In *SoCC*, September 2017.

[71] Supreeth Shastri and David Irwin. Towards Index-based Global Trading in Cloud Markets. In *HotCloud*, June 2017.

[72] Supreeth Shastri and David Irwin. Cloud index tracking: Enabling predictable costs in cloud spot markets. In *SoCC*, October 2018.

[73] Supreeth Shastri, Amr Rizk, and David Irwin. Transient Guarantees: Maximizing the Value of Idle Cloud Capacity. In *SC*, November 2016.

[74] Zhiming Shen, Qin Jia, Gur-Eyal Sela, Ben Rainero, Weijia Song, Robert van Renesse, and Hakim Weatherspoon. Follow the Sun through the Clouds: Application Migration for Geographically Shifting Workloads. In *SoCC*, October 2016.

[75] Jeffrey Shneidman, Chaki Ng, David Parkes, Alvin AuYoung, Alex Snoeren, Amin Vahdat, and Brent Chun. Why Markets Could (but don't currently) Solve Resource Allocation Problems in Systems. In *HotOS*, June 2005.

[76] Yang Song, Murtaza Zafer, and Kang-Won Lee. Optimal Bidding in Spot Instance Market. In *Infocom*, March 2012.

[77] Ian Stoica, Hussein Abdel-Wahab, and Alex Pothen. A microeconomic scheduler for parallel computers. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSPP)*, April 1995.

[78] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. SpotOn: A Batch Computing Service for the Spot Market. In *SoCC*, August 2015.

[79] Supreeth Subramanya, Amr Rizk, and David Irwin. Cloud Spot Markets are Not Sustainable: The Case for Transient Guarantees. In *HotCloud*, June 2016.

[80] Ivan Sutherland. A Futures Market in Computer Time. *CACM*, 11(6), June 1968.

[81] ShaoJie Tang, Jing Yuan, and Xiang-Yang Li. Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In *CLOUD*, June 2012.

[82] Adel Toosi, Ruppa Thulasiramy, and Rajkumar Buyya. Financial Option Market Model for Federated Cloud Environments. In *UCC*, November 2012.

[83] Tiffany Trader. Amazon Web Services Spotlights HPC Options. HPCWire, August 2015.

[84] William Voorsluys and Rajkumar Buyya. Reliable Provisioning of Spot Instances for Compute-Intensive Applications. In *AINA*, 2012.

[85] Carl Waldspurger, Tad Hogg, Bernardo Huberman, Jeffrey Kephart, and Scott Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[86] Kevin Walsh and Emin Gun Sirer. Experience with an Object Reputation System for Peer-to-Peer Filesharing. In *NSDI*, May 2006.

[87] Cheng Wang, Qianlin Liang, and Bhuvan Urgaonkar. An Empirical Analysis of Amazon EC2 Spot Instance Features Affecting Cost-effective Resource Procurement. In *ICPE*, April 2017.

[88] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting Spot and Burstable Instances for Improving the Cost-efficacy of In-Memory Caches on the Public Cloud. In *EuroSys*, April 2017.

[89] Josh Whitney and Delforge Pierre. Data Center Efficiency Assessment. Technical report, Natural Resource Defense Council, August 2014.

[90] John Wilkes. Utility Functions, Prices, and Negotiation. Technical report, Hewlett-Packard, July 2008.

[91] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *EuroSys*, 2012.

[92] Rich Wolski and John Brevik. Providing Statistical Reliability Guarantees in the AWS Spot Tier. In *HPC*, April 2016.

[93] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic guarantees of execution duration for amazon spot instances. In *SC*, 2017.

[94] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. Blending On-Demand and Spot Instances to Lower Costs for In-Memory Storage. In *International Conference on Computer Communications (Infocom)*, July 2016.

[95] Sangho Yi, Derrick Kondo, and Artur Andrzejak. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *CLOUD*, July 2010.

[96] Murtaza Zafer, Yang Song, and Kang-Won Lee. Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs. In *CLOUD*, 2012.

[97] Sharrukh Zaman and Daniel Grosu. Efficient Bidding for Virtual Machine Instances in Clouds. In *CLOUD*, July 2011.

[98] Liang Zheng, Carlee Joe-Wong, Christopher Brinton, Chee Tan, Sangtae Ha, and Mung Chiang. On the viability of a cloud virtual service provider. *ACM SIGMETRICS Performance Evaluation Review*, 44(1), 2016.

[99] Liang Zheng, Carlee Joe-Wong, Chee Tan, Mung Chiang, and Xinyu Wang. How to Bid the Cloud. In *SIGCOMM*, August 2015.