FINANCIALIZING CLOUD COMPUTING

A Dissertation Outline 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

July 2017

Electrical and Computer Engineering

FINANCIALIZING CLOUD COMPUTING

A Dissertation Outline 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

ABSTRACT

FINANCIALIZING CLOUD COMPUTING

JULY 2017

SUPREETH SHASTRI

M.S., COLUMBIA UNIVERSITY

Directed by: Professor David Irwin

As cloud computing rapidly expands to become the de-facto computing platform of our society, cloud providers are embracing this opportunity by aggressively scaling up their investments in infrastructure. However, providers face competitive pressure to (i) offer purchasing options tailored to the requirements of a wide range of users, and (ii) increase the utilization and thus revenue from their expanding fleet of infrastructure. In response to these dynamics, providers have continued to evolve Infrastructure-as-a-Service (IaaS) clouds from simple fixed-price server rentals into full-fledged marketplaces. For example, Amazon EC2 servers come in 76 different hardware configurations, each available under 10 different contracts that vary in their pricing model, time commitment, resource guarantees, and risk exposure.

While this expanded choice in server selection provides opportunities for cost savings, current generation system software and frameworks are ill-prepared for it. For example, modern big data frameworks such as Hadoop and Spark do not have support distinguish underlying servers by their contract types. Thus, without adequate system-level support, the burden of managing this newfound complexity in cloud contracts falls on cloud application developers, who may not have the necessary background for it. As a result, many applications simply resort to using the default on-demand servers despite their high costs.

In this thesis, we address the challenges of managing and benefiting from the contract diversity in infrastructure clouds. Our insights come from a key observation: cloud has evolved to resemble financial and commodity markets but the current software systems do not treat it like one. To address this disconnect, we design system-level abstractions, mechanisms, and policies that derive and extend concepts from finance and economics, to manage cloud market's complexity. To this end, we make four contributions (i) devise an *asset-pricing model* for transient servers, (ii) demonstrate *cost-efficient insurance* against revocation risks, (iii) design *active server trading* as an alternative risk-management mechanism to the prior insurance schemes, and (iv) create a *market-based index* for making better decisions on cloud investments, benchamrking and trading.

We demonstrate that by "*financializing cloud computing*"—i.e., treating cloud resources as investments, rationalizing about their risk-reward tradeoffs, and managing them in a way that adapts to changes in market price—we enable cloud applications to significantly reduce their compute costs while limiting their risks.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Idle Cloud is Provider's Workshop

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 was Amazon's Elastic Cloud Compute (EC2), which offered Infrastructure-as-a-Service (IaaS) virtual machines starting in August 2006 [12]. IaaS cloud platforms provide numerous benefits, including an on-demand access to compute resources, a pay-as-you-go billing model, and the illusion of near-infinite scalability, all of which come without a significant upfront capital investment. As a result, cloud computing has quickly become the foundation of our information-based economy, providing large-scale computing power to nearly every segment of our society, including science, communication, entertainment, finance, energy, and healthcare. Propelled by such widespread adaptation, cloud computing is projected to represent more than half of the $1.5 trillion worldwide IT spending by 2021 [22].

To accommodate this projected growth, and to maintain the cloud's illusion of unlimited compute capacity, major cloud providers namely Amazon EC2, Microsoft Azure, and Google Compute Engine, are rapidly expanding their datacenter footprint. However, this poses a significant challenge: maintaining high resource utilization, and thus revenue. This is challenging because large datacenters typically have utilizations around 10-50% [14]. To mitigate this problem, providers have started evolving their cloud platforms from simplistic fixed-price server rentals into a full-fledged marketplaces that offer a wide variety of service contracts. These contracts enable providers increase the temporal and spatial multiplexing of their underlying hardware resources, while also offering users customized options tailored to their needs.

The sophistication and diversity of contracts in modern IaaS clouds are starting to resemble those in financial and commodity markets, with service level objectives (SLOs) varying in pricing model, time commitment, resource guarantees, and risk exposure. For example, Amazon's reserverd instances and Google's committed-use VMs are akin to *futures* contracts, where users commit to 1-3 years of continuous server usage, in exchange for guaranteed access and deeply discounted pricing. Similarly, platforms created *transient server* contracts to sell their idle capacity. For example, Amazon's spot instances and Google's preemptible VMs, where customers can acquire cloud servers for heavily discounted prices but with the caveat that they may be unilaterally revoked by the platform anytime. As of this writing, Amazon offers 10 contracts and Google offers 4 SLO options for their infrastructure clouds.

While having a multitude of options in server selection does provide cost saving opportunities, the current generation of system software frameworks are ill-prepared for it. This can be traced back to decades of practice, 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 characterisitics was laregly irrelevant. As a result, system softwares and applications have evolved to treat all servers alike expect for their hardware configurations. Unfortunately, this evolution has continued to influence the design of modern system software like Hadoop, Spark, Mesos and Kubernetes.

Consequently, the burden of managing the newfound diversity in cloud resource contracts has passed on to application developers. But without adequate support from OS and system software, applications lack the necessary mechanisms and policies to maximize efficiencies in cost and performance. The current state of cloud computing is already an advanced marketplace but consumers lack the sophistication and tools to operate efficiently. In this thesis, we propose to *financialize cloud computing* to enable applications to operate efficiently in this market.

1.2 Financialization

Financialization [4] 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 [27] to convey the importance of financial markets, financial motives, financial institutions, and financial elites in the operation of the economy and its 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 tradeoffs, and manage them in a way that adapts to changes in market prices and risk. As the cloud continues to commoditize compute, and as cloud contracts resemble those in commodity markets like fuel, electricity and metals, the case for financially-aware 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.

While the cloud's evolution into a sophisticated marketplace has necessitated financialization, another significant technological trend is making it achievable: *system and network virtualization*. Recent advances in resource containers [5, 11, 47] and nested virtualization [17, 73, 58] are eliminating many of the implicit dependencies that bind applications to a specific underlying server. At the same time, advances in datacenter networking are reducing the time and performance penalties associated with migrating applications across servers. Together, these trends are turning compute time into a fungible resource, thereby enabling applications to "move with the market."

Though computing on the cloud is not (yet) an open market, and in fact, evolving to be dominated by a small number of large, monopolistic providers, there is ample internal diversity in each of the market silos to benefit from financializing cloud applications.

Cloud Market Problem	Proposed Solution	Result
Significant portions of cloud capacity is idle but difficult for providers to price and consumers to value	Asset pricing via *Transient Guarantees*	6.5x increase in value
Spot revocations pose a new fault-model that undermines the cost benefit of using transient servers	Cost-efficient insurance via *SpotOn*	91% cost-savings
Applications are not designed to benefit from the real-time price dynamics of cloud markets	Active Server trading via *Automaton*	50% lower cost 2-5x less revocations
Cloud markets are too diverse to understand at the individual server level	Index-based Decision-making via *Cloud Indices*	New paradigm for cloud investing and benchmarking

Table 1.1: Financializing cloud computing

1.3 Thesis Contributions

This thesis addresses widespread performance inefficiencies, price inversions, contract arbitrages, and risk proliferations that have emerged due to an increased diversity of service contracts in cloud markets. As part of our thesis, we identify four significant problems and design system-level solutions that help cloud applications, transparently and automatically, overcome these challenges. We highlight these contributions in Table 1.1, and describe them below.

1.3.1 Asset Pricing

Cloud providers are actively devising contracts to sell their idle cloud capacity that cannot otherwise be turned off. The resulting servers, which can be reclaimed by the provider at any time, form a new category of servers called transient servers [60, 61]. While transient servers are typically inexpensive, the value a customer derives from them is a function of the server's characteristics namely, its availability, volatility and predictability. These three characteristics collectively quantify the risk that a transient server poses vis-a-vis an on-demand server. Since providers want to retain their flexibility in reclaiming allocated transient servers, they do not provide users any assurances over these transiency characteristics. Unfortunately, this makes it difficult for users to value the servers correctly as well as optimally configure fault-tolerance mechansims.

To clarify the impact of this hidden transiency information, we define the notion of an *equilibrium price* i.e., the price above which the utility derived out of a transient server (modulo its volatility overhead) is no better than an equivalent on-demand server. Using this notion, we demonstrate that the current transient server offerings of Amazon and Google do not maximize the value for either customers or providers. To address this problem, we propose a new asset-pricing model called *transient guarantee* that offers probablistic assurances on transient server characteristics. Our evaluation shows that transient guarantees not only helps users in quantifying server performance and thereby valuing their utility correctly, but also enables providers to increase their revenue by up to ∼6.5X without sacrificing their ability to revoke servers anytime.

1.3.2 Insurance

Cloud spot markets enable users to bid for transient servers such that the cloud platform may revoke them unilaterally, if the market price rises above the bid. Due to their inherent risk, spot servers are often significantly cheaper (by as much as 90%) than the equivalent on-demand servers. While delay- and disruption-tolerant applications are a natural fit to exploit these price dynamics, they now face a new failure model— spot revocations, unlike hardware failures, are intentional, frequent and come with a short warning. Thus, applications need to employ fault-tolerance mechanisms to preserve their progress. Using fault-tolerance is akin to buying insurance, where the mechanism's cost and performance overhead is the "premium" and its ability to reduce cost and improve performance in the event of a revocation is the "payout." However, selecting a cost-optimal insurance scheme is non-trivial due of the scale of the spot markets, the diversity of revocation characteristics, and variations in application's resource usage.

To address this problem, we design a batch computing service called *SpotOn* that executes unmodified user applications on spot servers by dynamically determining the best insurance policy. SpotOn's goal is to achieve performances similar to that of on-demand servers but at a price near that of spot servers. By focusing exclusively on batch jobs, SpotOn has the freedom to choose from a wide set of fault-tolerance mechanisms as well as to exploit favorable servers from amongst 2400+ global spot markets.

In designing SpotOn, we make two contributions: first, we extend the classical fault-tolerance mechanisms of migration, checkpointing, and replication to the new fault scenario and model their overheads. Second, we derive a greedy cost-aware policy that selects a combination of spot market and fault-tolerance mechanism that results in lowest insurance premium (i.e. fault-tolerance overhead). We implement SpotOn on Amazon EC2 by leveraging the advances in Linux container technology, and our evaluations using Google cluster traces indicate up to 91.9% cost savings compared to using on-demand servers.

1.3.3 Active Server Trading

Cloud spot markets offer servers for a variable price, but expose applications to the risk of server revocation if the price rises. SpotOn and other recent approaches manage this risk by treating revocations as failures, and then optimizing the use of fault-tolerance mechanisms. The problem with this insurance mode, is that if market conditions change, applications may end up paying high premiums for expensive servers.

To address the problem, we present *Automaton*, a cloud server that actively "trades" resources—by dynamically selecting and self-migrating to new hosts—as market conditions change. Automaton defines a trading policy that determines when to execute a trade by balancing a potential trade's transaction costs (from vacating a server early and briefly double paying) with its benefits (based on its expected savings).

In designing Automaton, we make two contributions: first, we provide an empirical analysis of Amazon EC2 spot markets to show that the best server (for many definitions of the best including most discounted, most resource-efficient etc) changes frequently, and thus make a case for active server trading. Next, we define a system-level mechanism for supporting active server trading, such that unmodified applications can leverage this abstraction. We implement Automaton on EC2 and show via our evaluations that Automaton's active server trading consistently outperforms prior insurance-based approaches in all of cost, performance and availability, under varying market conditions.

1.3.4 Index-aware Decision-making (Proposed)

As the adaption of cloud computing rapidly expands into diverse sectors and global geographies, so do the cloud spot markets. For example, Amazon EC2 operates 7600+ markets globally where users can rent transient servers for a variable price. To exploit these inexpensive servers, while mitigating the risk of price spikes and revocations, many researchers and startups have developed techniques for modeling and predicting prices to optimize spot server selection. However, these approaches focus largely on predicting individual server prices, which is akin to predicting the price of a single stock.

To address this problem, we introduce broad market-based indices for cloud, and empirically demonstrate that predictions based on aggregate market behavior are remarkably accurate and easier to make. Building on this insight, we propose (i) index-based trading for flexible applications, (ii) index-based benchmarking to rigorously compare spot-based systems, and (iii) index-based investing to enable users to rationalize their cloud investments over time. Our evaluations show the utility of such *index-aware cloud computing* in extracting better tradeoffs compared to current approaches.

1.4 Proposal Overview

We organize the rest of the proposal as follows. Chapter 2 provides the necessary background on cloud platforms and service contracts. Chapter 3 covers Transient Guarantees, our asset pricing model, and illustrates its effectiveness in increasing the value of idle cloud capacity. Chapter 4 desribes SpotOn, a batch computing service that designs a cost-efficient insurance policy against revocation risks of transient cloud servers. Chapter 5 details the design and implementation of Automaton and lays the groundwork for active server trading. Chapter 6 introduces our proposed work on market-based cloud index, and argues for the benefits of index-aware cloud computing. Finally, chapter 7 concludes the completed work and outlines the milestones for the proposed works.

CHAPTER 2

BACKGROUND

In this chapter, we provide background required for various aspects of this dissertation. We compare and contrast service contracts offered by cloud providers with those used in the commodity and financial markets. We describe the make up and composition of contract types in a representative cloud provider (Amazon EC2), share the recent progress towards commoditizing compute-time on cloud, and finally survey the related work to set a context for our contributions.

2.1 Cloud Contracts

Cloud contracts bear some resemblences to those in the conventional commodity markets namely, spot contracts and futures contracts. Spot contracts offer commodities for immediate delivery, while futures contracts offer commodities for delivery at some future date. These contracts are structured so that companies that rely on commodities can buy and sell them based on their expectations of future workload and future market prices. For example, an airline might purchase fuel 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 the expected demand at any point, it might sell or buy, respectively, spot contracts (or shorter term futures contracts) to ensure they match their real-time supply with their demand.

However, there are significant differences between compute-time and fungible commodities like electricity and oil, which influence the composition of the contract. In particular, while the investment strategies in conventional commodity markets consist of pure financial transactions, any trading of cloud resources must directly integrate with the systems and applications running on those resources. For example, to use a newly-allocated server, users must first initialize an application on it, which introduces system and application complexity and incurs a performance penalty. Cloud resources are also not a homogenous

pool like other commodities, but are instead divided into discrete resource bundles, e.g., of CPU, memory, disk, etc., based on the underlying server hardware with a distinct price. However, compute-time is similar to electricity in that it must be used in real time, or it is wasted.

In this section, we explore the components of the cloud contracts that define the service level objectives a provider is willing to offer to their customers. Based on the contracts offered by major cloud providers, we group these into four broad categories.

2.1.1 Composition of Cloud Contracts

- **Pricing Model**

  Amongst all the components of the cloud contracts, the pricing model exhibits the most diversity. While cloud computing started with fixed per-hour pricing, it has expanded to include different granularities (per-hour, per-minute, per-year), payment frequencies (monthly, all upfront, partial upfront with monthly), pricing mechanisms (fixed, variable, market-based, custom), and pricing levels (inexpensive, expensive, discounted by usage level).

- **Time Committment**

  Along time committment axis, contracts exhibit variations in the duration of the rental: fixed-length until the contract expires, variable-length until the customer requires, or variable-length until the contract is revoked by the provider.

- **Risk Exposure**

  Risk exists along two metrics: obtainability (whether the server could be obtained at a time of customer's choosing), and availability (if an obtained server could be used without being revoked until a time of customer's choosing).

- **Resource Guarantees**

  Resource guarantees vary from fully-dedicated hardware (i.e., every VM is hosted on a dedicated physical server), fully-shared hardware (i.e., customer's VM is hosted on a physical server that also has VMs by other customers) and partially-dedicated hardware (i.e., VMs from a particluar customer are multiplexed on a bigger underlying physical

Contract	Pricing Model	Time Committment	Risk Exposure
On-demand	expensive, fixed pricing, charged per hour	flexible	not always obtainable, always available
Reserved	inexpensive, fixed pricing, charged per 1-3 year(s)	fixed	always obtainable, always available
Reserved Market	inexpensive, fixed pricing, charged for leftover contract	fixed	always obtainable, always available
Spot	inexpensive, variable pricing, charged per hour	flexible	not always obtainable, any time revocable
Spot Block	inexpensive, variable pricing, charged for hold duration	fixed (1-6 hours)	not always obtainable, always available
Burstable	inexpensive, fixed pricing, charged per hour	flexible	not always obtainable, always available

Table 2.1: Summary of EC2 contracts under shared resource category. All contracts except for Burstable are offered under dedicated-resource category but with different cost-risk-committment tradeoffs

server). Resource guarantees directly affect the performance levels observed by the user applications (predictable, variable or burstable).

2.2 Amazon EC2

Table 2.1 summarizes seven distinct contract types offered by Amazon EC2. These contracts represent different tradeoff points between cost, risk and performance. We note that the table offers only a coarse approximation of each contract's term and composition, as they are too complex to precisely capture in a single table.

On-demand servers are the simplest contract, incurring a fixed price per unit time, e.g., every hour or minute, and enabling users to request and relinquish them at anytime. Reserved servers differ from on-demand servers in that they incur a much higher upfront cost for a 1- or 3-year term, and have no rejection risk. By contrast, EC2 may periodically reject new requests for on-demand servers [48]. However, reserved servers eliminate much of the elasticity benefits of using the cloud for variable workloads, and may significantly increase costs if not highly utilized. To provide some elasticity for reserved servers, EC2 also operate reserved server marketplaces, where users can recoup their upfront costs—if they are not utilizing reserved servers as expected—by selling the remaining term of their reservations. The price of reserved servers in this market varies based on supply and demand, and is similar to a futures market, where users lock in a price for future resources.

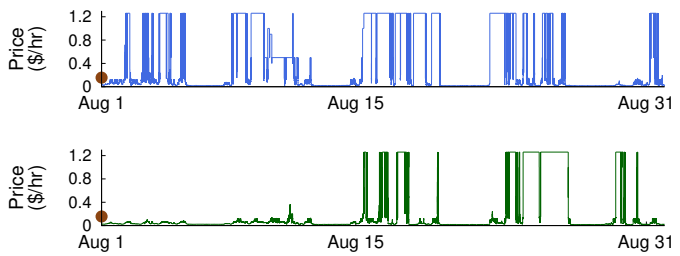


Figure 2.1: The spot price of the `m4.large` server varies across two AZs of EC2's US-East region over Aug 2016. The red dot indicates the on-demand price

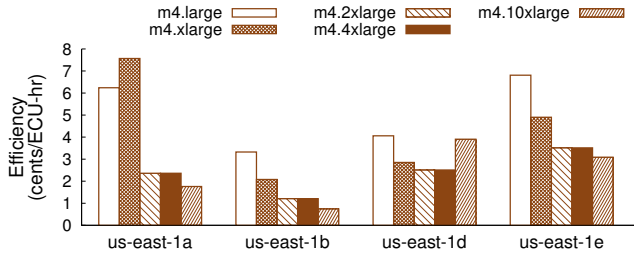


Figure 2.2: The normalized efficiency in \$/ECU of the `m4` family of servers varies across spot markets within each AZ of the US-East-1 region over Aug 2016

Spot servers are perhaps the most direct example of dynamism in the cloud market. Spot servers generally offer low prices (50-90% less than on-demand) but expose users to revocation risks, since a price spikes can occur at anytime. The global spot market is massive, as EC2 operates a separate spot market with its own dynamic spot price for each server type and configuration in each Availability Zone (AZ) of each region. Importantly, there is often a difference in the price of the same server in different AZs, and the normalized price per unit of resource of different servers in the same AZ. Such price differentials in the market represent a potential arbitrage opportunity that financially-aware applications can exploit to lower their costs. For example, Figure 2.1 shows that the `m4.large` server's price in two AZs of EC2's US-East-1 region over August 2016 differs by up to 30×. Similarly, Figure 2.2 shows that the average normalized price per EC2 Compute Unit (ECU) for the five different server types in the `m4` server family of the same AZ differs by up to 4× over the same period.

EC2 spot servers and Google preemptible servers represent a type of transient server that is available for an uncertain amount of time, and may be revoked at anytime. Since their introduction [60, 61], there has been significant research on optimizing system performance on transient servers in many contexts [66, 57, 76, 77, 30, 28]. As we discuss in Chapter 3, transient servers require the use of fault-tolerance mechanisms to gracefully handle revocations. Since these fault-tolerance mechanisms incur an overhead, the performance and value of transient servers is inherently less than that of stable on-demand servers. Spot-block contracts were recently introduced by EC2 to mitigate the impact of revocations on performance by enabling users to bid for fixed blocks of time between one and six hours. If allocated, the platform guarantees to revoke spot-block servers only at the end of their time block but not before. Since spot-block servers have no revocation risk over their time block, they are worth more than spot servers, e.g., costing 50%, and do not require the continuous use of high- overhead fault-tolerance mechanisms.

Finally, users may select from shared or dedicated server contracts, such that they have either high or low performance variability, respectively [25]. The latter allocates cores to VMs and exhibits low performance variability (only due to cross-talk among co-located VMs), while the former allows VMs to fairly share cores, such that performance varies based on the utilization of co-located VMs. In general, since performance variability is a form of risk, the more variable the performance the lower the cost.

2.3 Towards Commoditizing Compute-time

While the roots of our work (i.e., financializing cloud computing) lies in cloud's evolution into a marketplace, sophisticated policies and mechanisms that benefit from this approach require compute-time on cloud to be a fungible good i.e., a commodity. In this section, we address the challenges and opportunities towards that end.

For a long time, both researchers and practitioners have called for an open cloud commodity markets. However, such open cloud commodity markets have not yet materialized due to key differences between cloud resources and other commodities. In particular, the relationship between applications and their underlying server resources is fundamentally different and more complex than other commodities. For example, servers are stateful such

that migrating between servers, or experiencing a server revocation in the spot market, incurs a performance penalty and may affect application correctness. In addition, applications have numerous, often implicit, dependencies on a specific platform based on its underlying hardware, software, and network, which restricts the flexibility of applications to "move with the market" in order to exploit the lowest cost resources. Finally, there are security and privacy concerns with running applications in the cloud, requiring the user and provider to trust each other.

As a result, server-time has not yet to become a truly fungible resource, such that a server from one provider can freely substitute for a server from another provider. The complexity above also creates significant barriers to entering the market, resulting in commodity clouds currently being dominated by a small number of large, monopolistic providers, e.g., Amazon, Google, and Microsoft. Thus far, only these few providers have been able to amass the technical expertise, trust, and scale required to offer cloud servers as a commodity.

However, continuing advances in system and network virtualization are increasingly making server-time more fungible. For example, recent work on nested virtualization and "superclouds" is eliminating many of the implicit dependencies that bind applications to a specific cloud or hardware platform [58]; advances in data center and wide-area networking are reducing the time and performance penalty from migrating state to take advantage of low prices; and new security and privacy mechanisms, such as Intel SGX, are decreasing the trust applications must place in cloud providers and vice versa [16, 9]. In addition, the emergence of large-scale co-location data centers, which host cloud servers on behalf of customers, are enabling small providers to take advantage of the operational efficiencies and economies-of-scale of large data centers [26].

As servers become more fungible, we expect cloud server resources to evolve into a true commodity market where a wide range of providers can offer server resources under a variety of pricing models. That said, the scale of todays monopolistic cloud platforms has already spawned rich internal cloud markets for resources, as indicated by the large set of complex contracts that are now offered.

2.4 Related Work

There is a long history of prior research on market-based resource allocation prior to the emergence of cloud computing, e.g., [70, 67, 64, 33, 34, 49, 10, 59]. 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.

More recently, there has been a variety of work on exploiting arbitrage in the spot market by both researchers [32, 66, 75, 51, 36, 69, 44, 38, 21, 40, 78, 81, 18] and industry [15, 45, 39]. However, this prior work only focuses on spot servers and does not leverage the wide range of other contracts offered by cloud platforms. Thus, this prior work does not consider the full range of investment strategies available to users. In addition, by focusing on spot servers, the allocation policies are inherently short-term and do not consider longer term investment strategies, e.g., using reserved servers. Prior work also often focuses on optimizing the use of spot servers for only narrow classes of applications [32, 75, 66] that necessitates complex application-specific modifications [51, 76, 21, 40, 29].

Our is broader than the work above in that it focuses on developing generalized platforms and tools to benefit a wide range of financially-aware applications, which leverage the full range of contracts offered by cloud platforms. While some recent work examines other cloud contracts, it focuses on gaming their inefficiencies and not simplifying the development of market-aware applications. For example, Zheng et al. [82] exploit GCEs billing model, which offers a sustained-use discount that reduces a servers per-hour price the longer a user holds it, by enabling users to hold servers indefinitely and resell their resources to other consumers. Likewise, HCloud attempts to select contracts based on predictions of servers performance variability [25].

CHAPTER 3

PRICING IDLE CLOUD CAPACITY

> "Uncertainty is more stressful than knowing for sure something bad will happen."
>
> Archy de Berker et.al., Nature communications [24]

A significant fraction of cloud capacity is idle [14] 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 6x). 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).

3.1 Idle Cloud Capacity 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 [48] (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.

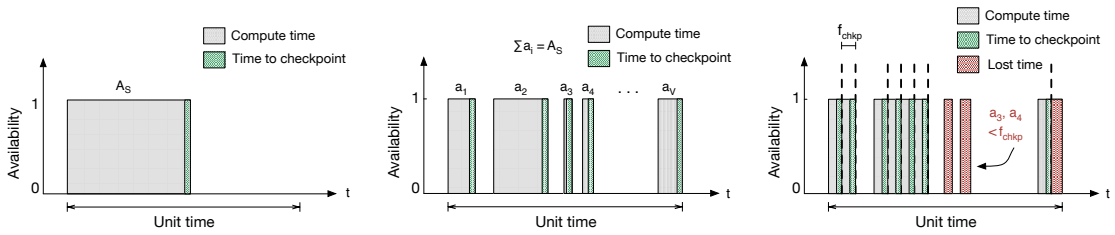
Since transient servers are a relatively new concept, there are not yet widely accepted standards for setting their terms and prices. EC2 offers its version of transient servers, called *spot instances*, via a market mechanism. Users place a bid for servers by specifying the maximum price they are willing 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 3.3, 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.

3.2 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 3.1: 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 3.1 illustrates, in the context of our simple batch job that these three metrics are distinct from, and independent of, each other. Figure 3.1(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 3.1(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

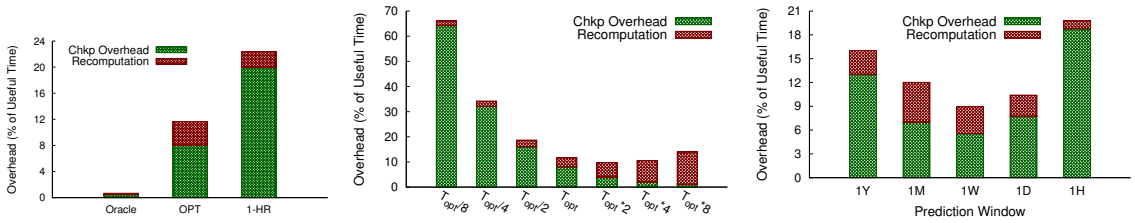
before because it needs to checkpoint much more frequently. However, since the time of each revocation remains well-known and predictable, it still need only checkpoint immediately prior to each revocation. Finally, Figure 3.1(c) shows a time-series again with the same availability, but with a high volatility 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 [83, 41, 79, 68] or specified availability [28]. 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 [54, 31, 66, 36, 53] 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 [65].

3.2.1 Impact on Application Performance.

To validate the utility of our transient server characterization, we simulate a long-running batch job in a representative EC2 spot market—`m1.large` instance running Linux in US-East-1 over the period Jan-1-2014 to Dec-31-2014. In this analysis, we assume a bid equal to the on-demand price, and we observe 555 revocations over the year with an



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

Figure 3.2: Impact on spot server performance due to incorrect MTTR characterization

MTTR of ∼15 hours. We assume a modest-sized memory footprint of 16GB, which takes ∼10 minutes to checkpoint based on our benchmarks using EBS magnetic disks.

Figure 3.2a 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 [23] ($t_{opt} \sim 2$ hours in this case) assuming the MTTR is known, and a static policy that checkpoints once each hour. The latter policy is proposed in prior work, since EC2 bills on an hourly basis [69]. 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 3.2b then shows the impact on overhead of incorrectly setting the checkpointing frequency when computing the optimal periodic checkpointing interval from Figure 3.2a. 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 3.2c 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.*

3.3 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 [23], 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 [13]. Based on our analysis in Figure 3.1(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.

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 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

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

Table 3.1: Approaches to selling idle cloud capacity

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 3.1 summarizes the differences between our MTTR-based transient guarantees, GCE preemptible, and EC2 spot instances.

3.3.1 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

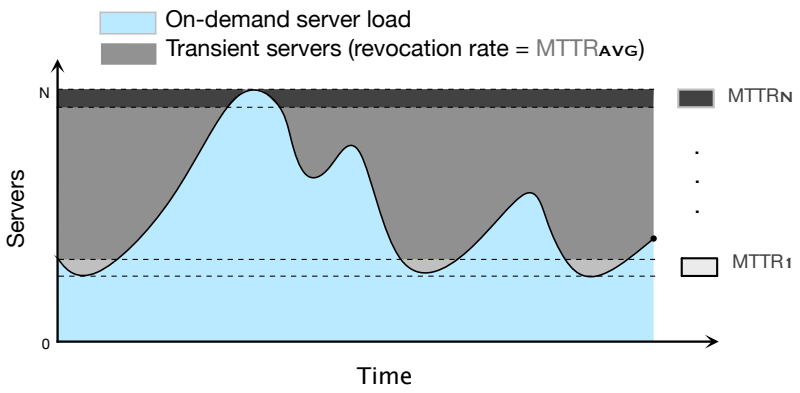


Figure 3.3: Platforms may offer their idle capacity as multiple transient classes with different transient guarantees

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:

$$\begin{aligned} E[T_j] &= T_j + \frac{T_j}{t_{opt}}\delta + \frac{T_j}{MTTR} * \frac{t_{opt}}{2} \\ p_{eq} &= p_o * \frac{T_j}{E[T_j]} \end{aligned} \tag{3.1}$$

3.2.2 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: (i) multiple choices at different price/risk levels allows customers to select servers that match their workload requirements closely, and (ii) more accurate the revocation characteristics, lesser the fault-tolerance overhead and higher cost efficiency.

To illustrate, consider Figure 3.3, 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 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

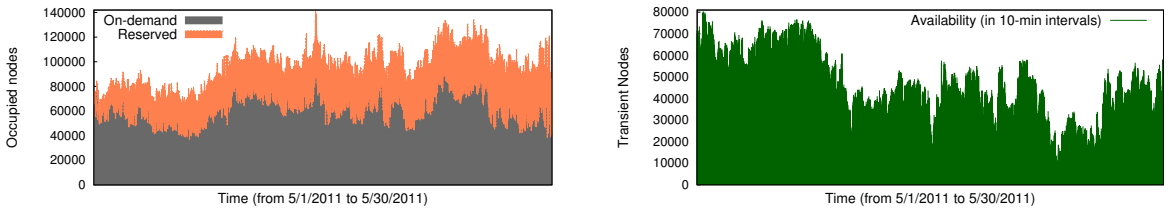


Figure 3.4: Transient servers resulting from the idle capacity in Google cluster traces

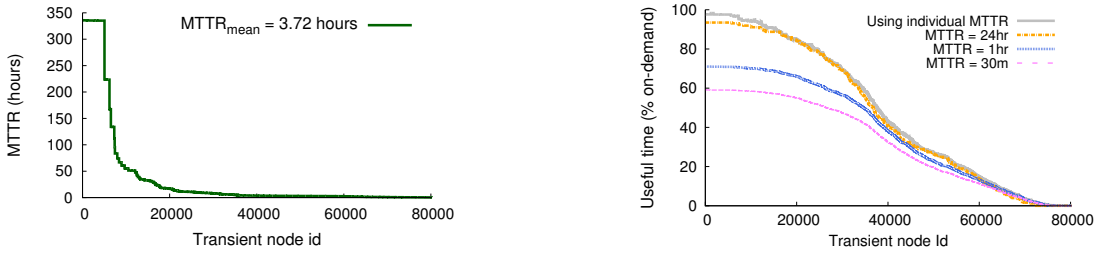
revocations than lower ranked servers. Thus, a platform could carve out the top N^{th} server to be allocated and offer it as a separate class with $MTTR_N \gg MTTR_{avg}$. Since the N^{th} server's MTTR is much longer than the average, it is more valuable to applications: it experiences fewer revocations, 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. 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. To partition N servers into k classes, we propose two heuristic policies (i) Equal-split, where every class gets N/k servers (ii) Greedy-split, a greedy approach to increasing the number of servers in higher classes. We evaluate the benefits of transient classes in the next section.

3.4 Evaluation

The goal of our evaluation is two-fold. First, we analyze the demand pattern of a production Google cluster trace [50] over a month-long period to quantify the characteristics of its idle capacity. Next, 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.

3.4.1 Origin and Characteristics of Idle Cloud.



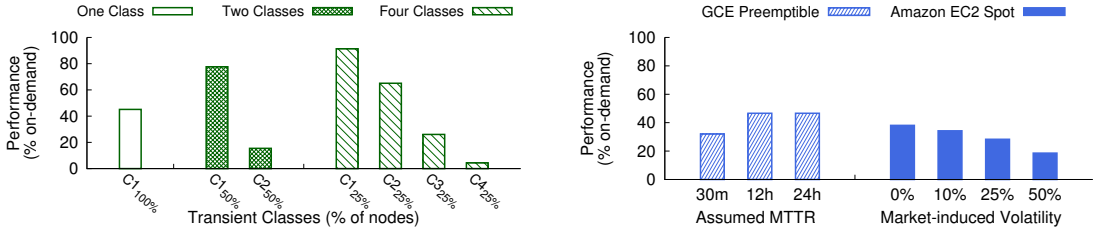
(a) MTTR Distribution (b) Performance

Figure 3.5: Revocation and performance characteristics of transient servers in 3.4

Using the Google cluster traces, Figure 3.4(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 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 3.4(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 3.5a 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 3.5b 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.



(a) Transient Guarantees

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

Figure 3.6: Performance of transient servers under different pricing models

Result: *Since idle capacity varies over time, transient servers exhibit a wide range of characteristics. Checkpointing transient servers based on incorrect revocation characteristics results in significant performance losses (up to ∼40%), especially for the least volatile (and most valuable) servers.*

3.4.2 Transient Guarantees.

Figure 3.6a 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 3.6b 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

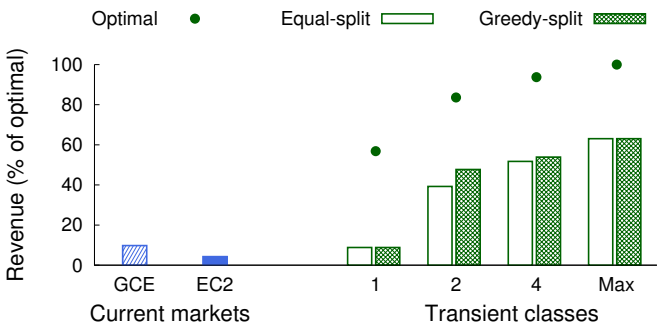


Figure 3.7: Revenue comparison from selling transient servers

assumes a MTTR of 30 minutes, 12 hours, and 24 hours. For EC2, we use the commonly employed one hour checkpointing strategy [69]. Since EC2's spot market is strictly more 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 [57] to assign prices to transient servers based on their performance. Figure 3.7 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 3.6. 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.*

3.5 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 [18, 35], designing optimal bidding policies [79, 68, 83], and modifying particular applications to gracefully handle transient servers using fault-tolerance mechanisms, such as checkpointing [69, 66]. 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%) [20]. 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 [13], 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.

3.6 Conclusion and Status

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 [57].

CHAPTER 4

INSURING AGAINST REVOCATION RISKS

"Certainty belongs to mathematics, not to markets."

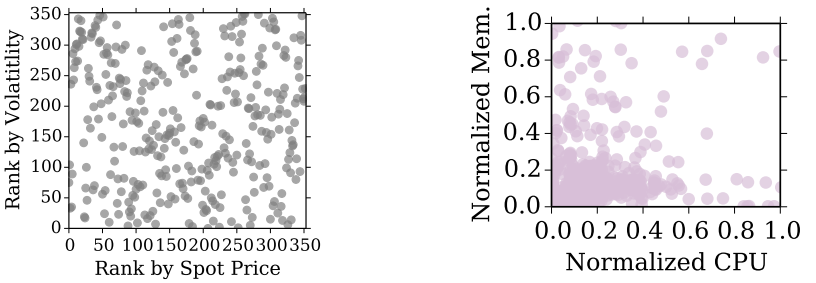
Bill Miller, Investor and portfolio manager

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.

4.1 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. To illustrate the former, we



(a) Diversity in Spot markets

(b) Diversity in Google cluster jobs

Figure 4.1: Spot markets and jobs exhibit a wide range of characteristics.

analyze all the 353 spot markets in the US-East-1 region over Dec-2014 to Mar-2015, as well as a random sample of 1000 jobs from Google cluster traces [50]. Figure 4.1a then shows a scatterplot of the rank of the spot markets in terms of their average spot price and volatility, which demonstrates that markets with lowest price need not always be least volatile. Similarly, figure 4.1b shows a scatterplot of CPU, memory, and I/O resource usage for Google cluster trace jobs (normalized to the 99th percentile value). We discuss the fault-tolerance mechanisms in Section 4.2.

Architecture. We depict SpotOn's architecture in figure 4.2. 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

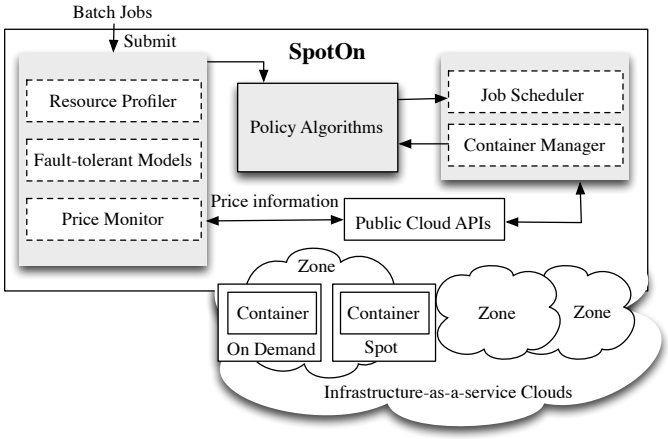


Figure 4.2: SpotOn's Architecture

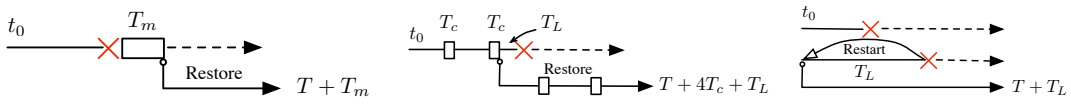
time. SpotOn also chooses whether the job should use locally-attached or remote storage, e.g., via EBS. After making these decisions, SpotOn acquires the chosen instance from the underlying IaaS platform, configures the selected fault-tolerance mechanism, and executes the job within a container on the 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.

4.2 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 4.3 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



(a) Reactive Migration. (b) Proactive Checkpointing. (c) Replicating Computation.

Figure 4.3: 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.

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.

$$T_m = \frac{M + D}{R_b^s} + \frac{M + D}{R_b^r} \tag{4.1}$$

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.

$$T_c = \frac{M + D}{R_b^s} \tag{4.2}$$

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.

4.3 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 $\bar{C}^k_{sp}$, 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 [23]. 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 4.1.

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

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

Table 4.1: Expected runtime and cost under different fault-tolerance mechanisms

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 [53] 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.

4.4 Evaluation

The goal of our evaluation is to quantify the benefit of SpotOn's cost-aware insurance policy that chooses the fault-tolerance mechanism and spot market to minimize costs, while mitigating the impact of revocations on job completion time. We conduct experiments using our prototype and in simulation.

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 create a baseline batch job that runs for an hour, has a memory footprint, i.e., working set size, of 8GB, and has a CPU:I/O ratio of 1:1. 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).

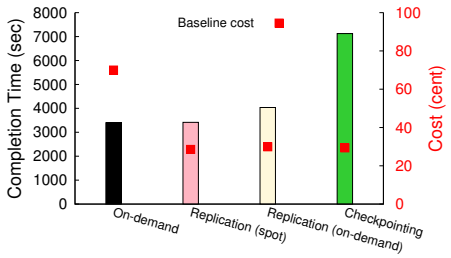
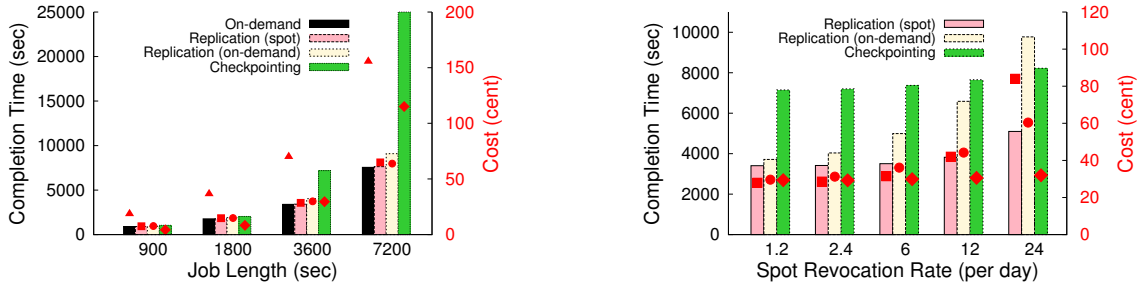


Figure 4.4: Baseline job's performance and cost when running on an on-demand instance versus running on spot instances using different fault-tolerance mechanisms.

We execute the job on a `r3.2xlarge` instance type, which costs 70 cents per hour, and measure its average completion time across multiple runs to be 3399s.

Figure 4.4 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. 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 incurs periodic overheads, and also unable to use local disks as they are prohibitively expensive for migration. Thus, replication benefits from the I/O intensity of our baseline job.

Figure 4.5a 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 4.5b 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



(a) Job Duration

(b) Revocation Rate

Figure 4.5: Baseline job's performance and cost as its job length (a) and spot market characteristics (b) vary, while keeping other attributes constant.

contrast, checkpointing's cost and performance is more 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.

Policy Results

We build a policy simulator and use it 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. Then, we randomly select 1000 tasks from Google cluster traces [50] and compare the cost of SpotOn's greedy cost-aware policy and running the jobs on the user-specified on-demand instance. Our results show that SpotOn's cost-aware policy not only achieves 91.9% savings over the on-demand option but also decreases the aggregate running time by 13.7%. This decrease was because of SpotOn's ability to often select faster and more resourceful instances for a cheaper price than those specified by the user.

Summary: *The insurance overhead 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. By always selecting the combination of spot market and fault-tolerance mechanism that yields minimum insurance overhead, SpotOn's greedy cost-aware policy achieves performance of on-demand servers at the price of spot servers.

4.5 Related Work

SpotOn is similar to recent startup companies, such as ClusterK [45] and Spotinst [39], 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 [44, 80, 62, 79, 68, 42] and checkpointing [69, 38, 78] 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 [69], 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 [54], map-reduce jobs [83], in-memory cache [72], in-memory storage [75] and machine learning [30]. 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.

4.6 Conclusion and Status

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 [66].

CHAPTER 5

ACTIVE SERVER TRADING

"Every man lives by exchanging."

Adam Smith, Wealth of Nations (1776)

While cloud spot markets offer inexpensive servers, they expose applications to the risk of server revocations and unexpected price spikes. SpotOn and other recent approaches manage these risks by employing fault-tolerance mechanisms or by making predictions on the market behavior. While, fault-tolerance mechanisms provide "insurance" against failures, applications may end up paying high premiums for expensive servers, if market conditions change. In this chapter, we explore an alternative risk management technique based on *active server trading*. We provide empirical evidence on its feasibility using Amazon EC2 spot markets, and design a policy that adapts and extends concepts from finance to manage cost and risk in real-time. Below, we present the design and evaluation of *Automaton*, a cloud server that transparently trades resources in response to market changes.

5.1 Why Insure When You Can Trade

As cloud spot markets have continued to expand their footprint [1], many researchers [30, 32, 51, 52, 54, 60, 61, 66, 76, 77] and startups [15, 39, 45] are actively working to exploit their low price, high risk transient servers. All prior works manage the revocation risk in the same general way: they treat revocations as failures, and then optimize the use of fault-tolerance mechanisms, particularly checkpointing, to handle them. This method of managing risk is akin to buying insurance, where the mechanism's cost and performance overhead is the "premium" and its ability to continue executing after a revocation is the "payout." As we discuss below, there are multiple problems with leveraging fault-tolerance mechanisms as "insurance" for transient servers.

Most importantly, if market or workload conditions change, applications may incorrectly configure their fault-tolerance mechanism, e.g., checkpointing too frequently, causing them to "pay" non-optimal premiums. The benefits of employing fault-tolerance are also inherently 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. Finally, fault-tolerance-based approaches only address revocation risk and not *price risk*: the risk that the price of the current server will rise, or that of another server will fall.

As we summarize below, active server trading addresses many of the problems with fault-tolerance-based mechanisms.

- **Lower Overhead.** Trading does not incur unnecessary fault-tolerance overhead based on probabilistic information. While each trade incurs an overhead, it also serves as a natural checkpoint that applications only pay if the expected savings outweigh the costs.
- **More Deterministic.** Since trading decisions are based on current cost, risk, and performance information, they are generally more deterministic than decisions on how to configure fault-tolerance mechanisms, which are based on probabilistic expectations of future cost, risk, and performance. As we discuss, a trade's transaction costs are largely deterministic, while its savings are based on near-term expectations.
- **More Transparent.** Trading can be implemented transparently at the systems-level, e.g., using live migration of nested VMs or resource containers, and thus does not require complex application modifications.
- **Lower Risk.** Trading enables applications to reduce both revocation and price risk by continuously migrating to servers that offer the best combination of both.

Thus, we hypothesize that risk management based on active trading is simpler, more cost-effective, and less risky than previous approaches based on using fault-tolerance mechanisms. In the next section, we present an empirical analysis of EC2 spot markets towards validating this hypothesis.

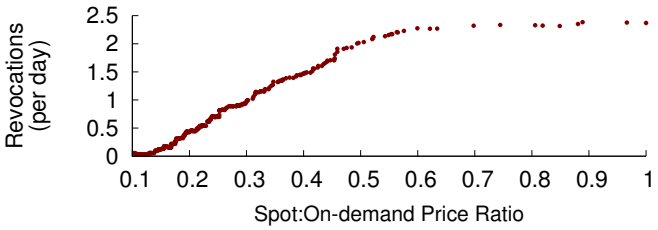


Figure 5.1: The revocation rate increases as the spot price increases relative to the on-demand price

5.2 Active Server Trading in EC2

To empirically validating our hypothesis on active server trading, we quantify two key spot market metrics: risk and cost.

Risk Metric.

We quantify the revocation risk of a spot server k based on the ratio of its current spot price P_k^{Sp} relative to its on-demand price P_k^{Od}. The lower this ratio the further away the supply/demand balance is from being constrained, and thus causing prices to spike and a revocation to occur. That is, the lower this ratio the greater the change in the supply and demand required for the price to rise above the on-demand price. We use a server's on-demand price as the baseline because it represents the *risk-free price*. Thus, we define a spot server's "return" or savings as

$$Return_k = 1 - (P_k^{Sp} / \ P_k^{Od}) \tag{5.1}$$

Counterintuitively, servers with the highest return have the lowest demand (relative to their supply), and thus have the lowest risk of revocation. The relationship between revocation risk and a spot server's returns is a fundamental market property: the higher the returns the lower the revocation risk. Trading based on such fundamental market relationships is common in finance. For example, the fact that bond prices always move in the opposite direction as interest rates is a fundamental relationship in finance. Thus, investors inevitably shift more assets to bonds as interest rates drop, since their price is guaranteed to rise (and vice versa). In contrast, prior work on spot servers often attempts to model and predict future prices and volatility [72, 71], which is much less reliable. In

Spot Market	Revocations (per day)
c3.2xlarge.vpc.1a	22.6
g2.2xlarge.vpc.1e	21.8
g2.8xlarge.1a	18.0
m4.large.vpc.1d	17.6
m3.xlarge.vpc.1d	17.6
m4.xlarge.vpc.1a	16.4
r3.4xlarge.1a	15.8
g2.2xlarge.vpc.1a	15.7
c3.2xlarge.vpc.1e	15.7
c3.xlarge.1a	15.2
Automaton	**7.5**

Table 5.1: Automaton achieves a lower revocation rate than any server in the primary market by migrating to the server with the current lowest risk of revocation.

particular, in a mature market, consistently accurate predictions are not possible, as the efficient market hypothesis states that predicting future prices cannot "beat the market," as current prices reflect all available information [57].

Figure 5.1 shows this relationship for all the 340 Linux spot markets in EC2's US-East-1 region over August 2016. On the x-axis is the spot price, as a percentage of the on-demand price, which represents P_k^{Sp}/P_k^{Od}. The y-axis then shows the corresponding revocation rate across the entire market when the spot price is less than or equal to the value on the x-axis. The result validates our basic intuition above, namely that as a market's average returns increase, its average revocation rate decreases. This insight is directly crafted into Automaton's trading policy.

Table 5.1 then quantifies the benefits of this trading policy, even when we are restricted to the top-10 most volatile markets over the same period. The table shows the average revocation rate for each of these volatile servers over August 2016, as well as the average revocation rate Automaton can achieve by continuously migrating to lowest-risk market within this subset as price fluctuates. Here, Automaton defines a secondary cloud server with a revocation rate 2× less than the server with the absolute lowest revocation rate.

Cost Metric.

We define our cost metric as the price incurred per unit of resource utilized by the application. While the pricing of a server encompasses its compute and memory capacity,

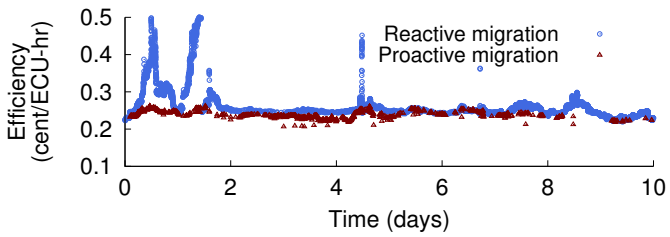


Figure 5.2: The most-efficient server constantly changes as spot prices vary

network connectivity and disk capability, we focus only on the compute part to simplify our analysis. For example, if a host with 2 ECUs and a price of 10¢/hour has an average utilization of 25% across its vCPUs, we compute its cost-efficiency as 10¢/(2*0.25) = 20¢ per ECUs utilized per hour.

Since all EC2 hosts have an ECU rating, we approximate ECU utilization on each potential host by proportionally scaling the ECU utilization of the current host across all vCPUs on the new host i. For example, consider a workload that has an average utilization of 25% on a host with 2 ECUs. Automaton would estimate that a host with half the ECU rating would have a utilization of 50%, and a host with twice the ECUs would have a utilization of 12.5%.

$$Util_i = \frac{Util_{current}}{ECU_i/ECU_{current}} \tag{5.2}$$

Given the estimated utilization of a new host i, Automaton calculates the new host's expected efficiency.

$$Efficiency_i = \frac{Price_i}{ECU_i * Util_i} \tag{5.3}$$

Figure 5.2 shows the benefit of proactively migrating solely to minimize cost. We compare Automaton with an approach that initially selects the most cost-efficient server but only migrates reactively when that server is revoked due to a price spike. As the figure shows, Automaton's proactive migrations serve as a strict lower bound on cost-efficiency with the most cost-efficient server often more than 2× cheaper than the server that was the most cost-efficient at the outset of the 10 day period.

Spot Market	Efficiency (¢/ECU-hr)
m1.small.1d	44.00
m1.small.vpc.1d	43.99
m1.large.vpc.1d	43.75
m1.large.1d	43.75
m1.xlarge.vpc.1d	43.75
m1.xlarge.1d	43.74
g2.8xlarge.vpc.1d	24.99
m3.medium.1c	23.33
m3.medium.vpc.1c	23.33
d2.2xlarge.1b	22.42
Automaton	**10.89**

Table 5.2: Even when restricted to top-10 least efficient markets, Automaton achieves a higher cost-efficiency (by >2×) than any single server

Table 5.2 shows how Automaton is able to create a secondary server that is significantly more cost-efficient than any individual server. The table lists the cost of a subset of servers in the US-East-1 region over August 2016, along with Automaton's cost-efficiency when migrating in real-time to the current most cost-efficient of these servers. Interestingly, the spot servers in this subset are much more expensive and less cost-efficient than their corresponding on-demand servers. Again, Automaton is able to achieve an average cost that is 2× less than the cheapest server in the primary cloud market.

5.3 Design of Automaton

Automaton takes a simple approach to optimizing for cloud markets: it automatically self-migrates to new cloud hosts as market conditions and application resource usage change. Automaton migrations are transparent, and leverage existing systems-level migration mechanisms for virtualized cloud servers offered by containers [47, 11] and nested virtualization platforms [17, 73]. Importantly, unlike other approaches, Automaton requires no centralized infrastructure or coordination, as its migration functions and policies are entirely embedded into each individual server. Thus, applications need only execute on an Automaton-enabled server image to exploit cloud markets.

Automaton's design separates the host's management plane from the application's data plane, as depicted in Figure 5.3. The management plane, which is largely hidden from

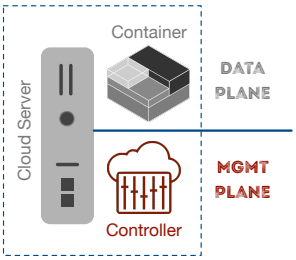


Figure 5.3: Automaton's controller runs in the management plane, while applications execute inside virtualized environment within the data plane

applications, runs a controller daemon, which executes Automaton's monitoring, migration, and trading policy functions. Applications then run independently within the data plane, which consists of a virtualized environment, e.g., a resource container, capable of transparent systems-level migration. By default, Automaton runs only a single container per host that has access to all the host's resources. Figure 5.4 depicts Automaton's control loop, which (1) monitors real-time market prices and application resource usage, (2) uses the information to determine when and where to migrate based on trading policy objectives, and then (3) executes the trade by making the migration. To migrate, the source controller directly requests and spawns a new server via cloud APIs and then migrates the container to it and transfers control to the controller running on the new host. Finally, once the migration completes, the new controller terminates the source server via cloud APIs.

When requesting a spot server, Automaton must determine a bid. Since EC2's current spot market requires applications to pay the spot price and *not* their bid price, Automaton does not require a sophisticated bidding strategy, as it actively migrates to a new host if the spot price rises. Thus, Automaton adopts a simple bidding strategy: since EC2 offers equivalent risk-free on-demand servers at a fixed-price, it always bids the on-demand price, as it can always migrate to the equivalent on-demand server at that price. The specific bidding policy is orthogonal to Automaton'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, Automaton could support a different bidding policy without altering any of its other functions.

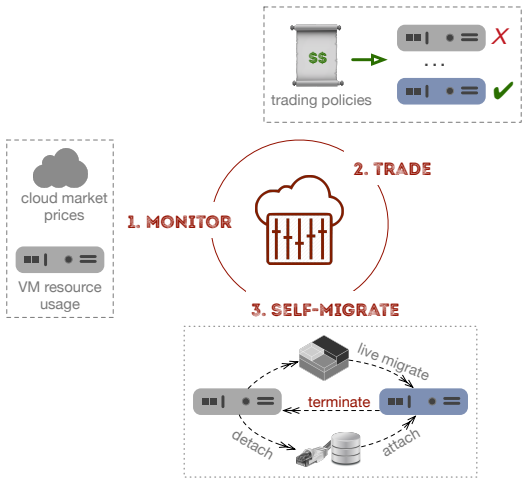


Figure 5.4: A depiction of Automaton's basic control loop

5.4 Trading Policy

Automaton's trading policy is built on the market insights of section 5.2. At its core, the trading policy maintains a real-time ranking of all the spot markets and continually evaluates the cost-benefit tradeoff of migrating to a better server.

Server Ranking. While a number of measures could be used to rank the servers based on their cost and risk, Automaton employs *Sharpe Ratio*. Sharpe ratio is a standard measure in finance, 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.

$$\mathbb{S}_i = \frac{E[R_i - R_{free}]}{\sigma_i} \tag{5.4}$$

For a spot market i, R_i is the spot price per ECU-hr and R_{free} is the on-demand price per ECU-hr. We can compute σ_i, the standard deviation of spot prices over a previous window. The numerator estimates our "return" relative to a risk free investment, while the denominator quantifies our "risk". Given two markets with an equal numerator, we should prefer the one with a lower risk. Automaton, via its monitoring engine, maintains a Sharpe ratio based real-time ranking of all the applicable spot markets.

Migration Cost. While there are many migration variants, a simple stop-and-copy migration is the optimal approach for batch jobs that permit downtime during migration. 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 $R_b = min(B, IOPS)$ to represent the bottlenecked data transfer rate.

$$T_m \propto \frac{M + D}{R_b} \tag{5.5}$$

Since the job is paused over T_m, the migration time also represents the downtime (or overhead) associated with each migration. Note that each migration incurs a cost based on T_m, since Automaton must pay for resources during this time but the job does no useful work.

Cost-benefit Analysis. As the market changes, our current server may no longer be the top-ranked server in our ranking. To guide the decision on whether or not to migrate to the new best server, the policy engine performs the following cost-benefit analysis: *if the trading benefit outweighs the trading cost, then migrate.*

$$\begin{aligned} Cost &= P_{new}^{Sp} * T_m + P_{cur}^{Sp} * T_{cur} \\ E[Benefit] &= (P_{cur}^{Sp} - P_{new}^{Sp}) * E[T_{new}] \end{aligned} \tag{5.6}$$

Here, P_{cur}^{Sp} and P_{new}^{Sp} are the real-time prices of the current and new spot servers. T_m is the migration cost, and T_{cur} is the remaining time to the end of billing period on the current server. The only term that is not deterministic is $E[T_{new}]$, the time we commit to the new server. In other words, this is the time until there is a new top-ranked server that is more attractive than our new server. Thus, $E[T_{new}]$ could be computed over a prior window of spot prices.

Using this formuation of cost and benefit, Automaton determines whether or not to migrate at a given time. This tradeoff is akin to the break even period on a mortgage refinance: one has to stay in the house so long before refinancing becomes beneficial. The

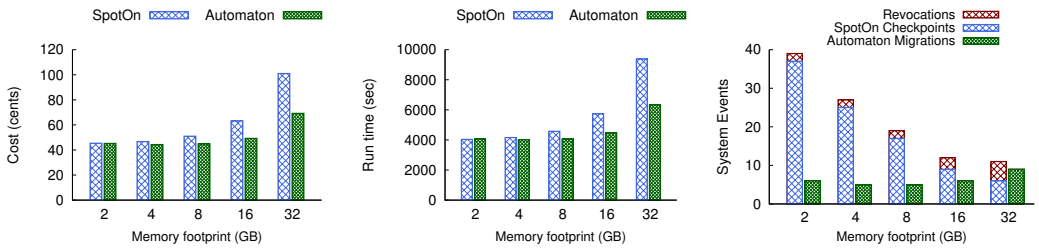


Figure 5.5: System performance when application's memory footprint changes.

risk of refinancing at any point is that if the interest rates drop further, we may not be able to take advantage of it because we are committed.

5.5 Evaluation

The goal of our evaluation is to quantify the effectiveness of active trading vis-a-vis current approaches. We pick SpotOn [66] as the representative of the insurance-based systems, and Amazon's SpotFleet as the representative of no-fault-tolerance approach. We conduct our evaluation in two phases. First, using real prototypes of Automaton and SpotOn on Amazon EC2, and second, using a simulator that runs jobs from Google cluster traces using EC2 spot price traces. The focus of the prototype experiments is to analyze how these systems respond to changes in application characteristics and market behavior, while the simulator experiments demonstrate the validity of our hypothesis at large scales over long periods.

Trading vs. Insuring.

We create a lookbusy [19] based batch job that runs for an hour on the reference server *m4.4xlarge* with a memory footprint of 16GB and CPU:IO time ratio of 1:1. We containerize it using LXC to transparently execute it. To predictably control market dynamics, we define five spot markets for *m4.4xlarge* servers, whose prices vary independently as randomly chosen points on sinusoidal waves of different frequencies. Both Automaton and SpotOn market monitoring engines have been modified to read price traces form these simulated markets and then go on to perform the actual operation (i.e. acquire, release, migrate) on real EC2 markets of *us-west-1c*.

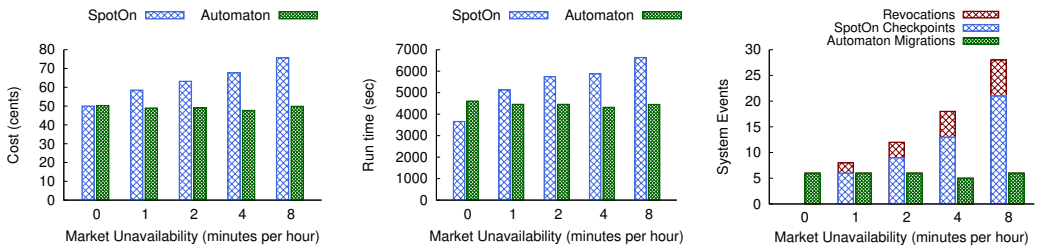


Figure 5.6: System performance when market conditions change.

For the baseline experiment, we set the average spot price of all five markets at 50% of on-demand price (our reference server *m4.4xlarge* costs $1.005 per hour in *us-west-1*) but allow a price volatility of ±25%. All the markets experience a price spike of >100% on-demand price for an average of two minutes every hour, or equivalently they experience two revocations per hour. We assume the servers are charged per minute of usage, and there is no minimum holding time.

Figure 5.5 shows how a change in application's footprint affects the cost and running time. Using the same configuration as the baseline experiment, we vary the application's active memory set from 2 to 32GB. At lower footprints, SpotOn and Automaton exhibit similar performances, however when the footprint increases, their performances diverge. At 32GB, Automaton finishes 32% faster than SpotOn while also costing 30% less. The System events subgraph indicates that at higher sizes, SpotOn reduces its checkpointing rate resulting in an increased impact of revocations. On the other hand, Automaton being reactive to market changes, is able to alter its migration pattern to avoid disruptive revocations.

Next, we vary the market condition in Figure 5.6 by inducing revocations via spot price increases beyond the bid level. The X-axis plots the resulting unavailability aggregated across all five markets. First, we see that when the market experiences no revocation (and when the systems are informed about it apriori), SpotOn outperforms Automaton albeit narrowly. However, as the markets become more volatile, SpotOn's performance degrades proportioanlly. On the contrary, Automaton demonstrates a steady performance at all levels of unavailability. This is primarily because Automaton is reactive to market changes and it actively migrates away from markets that experience higher than average price irrespective of whether or not it results in revocations.

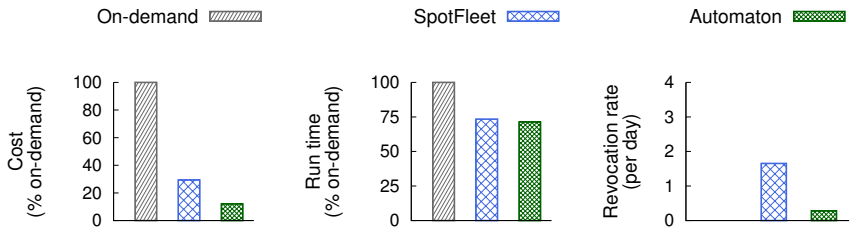


Figure 5.7: Cost, performance and risk comparison of using on-demand, SpotFleet, and Automaton when using all 340 spot markets of the US-East-1 region

Result: *Through active trading, Automaton is able to consistently outperform SpotOn, especially when application and market conditions are changing.*

Trading vs. No Trading.

Next, we compare Automaton's performance with two other approaches: running on-demand servers and using EC2's SpotFleet tool, which initially selects an efficient market but only switches to a new server on a revocation (i.e, no trading and no insurance). We select 1000 jobs randomly from the Google cluster trace [50], and simulate running them on EC2 using the spot price traces from August 2016. Since the Google cluster trace normalizes their server characteristics, we assume that the jobs were executed with a baseline number of 250 ECUs. For these experiments, we also include a performance penalty associated with revocation by requiring each job to restart after a revocation. Note that this is conservative, as Automaton could perform better by restarting the job from the previously migrated state.

Figure 5.7 compares Automaton's cost, performance, and risk with the competing systems using all 340 Linux spot markets of the US-East-1 region for the period between 8/1/2016 and 8/31/2016. The workload consists of 1000 randomly chosen tasks from Google cluster trace, such that they all have a minumum run length of 24 hours. Figure 5.7(left) shows that Automaton is able to achieve a cost-efficiency through its trading policy that is lower than both on-demand (by ∼90%) and SpotFleet (by ∼50%). Given the relative stability of the current spot markets, both spot-fleet and Automaton are able to complete their jobs faster than on-demand by making use of more efficient servers at lower costs. As shown in Figure 5.7(middle), the running time of both spot-fleet and Automaton are ∼30% less than using on-demand servers. Note that the running time for spot-fleet and

Automaton includes the additional time to restart any job that experiences a revocation. Finally, Figure 5.7(right) shows the relative risk of using each approach. This result shows how Automaton is able to cut the revocation rate relative to SpotFleet's approach by nearly 6× by considering a server's risk when migrating.

Result: *Automaton is able to achieve a lower cost (by ∼50%) and higher availability (by 2×-5× less revocations) than Amazon's SpotFleet.*

5.6 Related Work

Many researchers have recognized the opportunity to reduce costs by leveraging spot servers. However, there is little prior work similar to Automaton that supports proactive migration as market conditions change. Instead, the focus is on selecting the "optimal" server and spot market to execute a workload based on its expected resource usage and the market's expected future prices [66, 32, 51, 83]. Since prior work commits to running on a particular server until a revocation occurs and revocations incur a performance penalty, 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 that focuses on spot server bidding strategies [83, 71, 80, 62, 79, 68, 42, 74]. In contrast, the bidding strategy is not as important to Automaton, as it proactively migrates as market conditions change. Automaton never commits to a particular server, and often migrates before price spikes that cause revocations.

At a high level, derivative cloud platforms like SpotCheck [54] and Smart Spot Instances [36] may bear some resemblance to Automaton's ability to create abstract servers that have different cost-risk characteristics than the primary servers offered by the providers. However, Automaton is different in multiple ways. Unlike SpotCheck, which acts as a middleman in reselling resources to users, Automaton enables users to exploit cloud markets directly by simply selecting and running within an Automaton-enabled server image. Unlike Smart Spot, which focuses on centralized scheduling and optimal packing of nested VMs, thereby leaving fault-tolerance and risk management to individual servers, Automaton transparently handles these at the server level. Finally, both SpotCheck and Smart

Spot are centralized systems, while Automaton takes a decentralized approach to reduce revocation risk through migration at the individual server level.

5.7 Conclusion and Status

This chapter presents Automaton, a cloud server that actively "trades" resources—by dynamically selecting and self-migrating to new hosts—to exploit cloud markets as price and resource usage fluctuate. We demonstrate the benefits of active server trading in the cloud markets, and its effectiveness as a risk management technique. Using concepts from finance, Automaton's trading policy outlines a model for cost-benefit analysis towards executing profitable trades. We build a real implementation of Automaton on Amazon EC2, and release it publicly. We evaluate Automaton against SpotOn and SpotFleet, in prototype and via simulation, to show that active trading consistently outperforms insurance-based approaches in all of cost, running time and availability metrics.

Status. Automaton's initial implementation on EC2 has been released for public use at [2]. Additional details on its design, implementation and evaluation are in [55].

CHAPTER 6

INDEX-AWARE CLOUD COMPUTING

> "Always remember, a bird's eye view is way different from a worm's eye view, when in fact, they're looking at the exact same thing."
>
> Paula Peralejo

Infrastructure clouds have evolved from simple fixed-price server rentals into full–fledged marketplaces that offer a wide variety of service contracts differing in their pricing model, time commitment, resource guarantees, and risk exposure. While this allows cloud users to better match their workloads to the IaaS servers they purchase, the diversity of contracts and server choices make it a challenging endeavor. The problem is acute in Amazon EC2's spot markets, which operate more than 7600 server markets globally. Many researchers and startups have developed techniques for modeling and predicting individual server prices, which is akin to predicting the price of a single stock.

In this chapter, we introduce broad market-based indices for cloud, and empirically demonstrate that predictions based on aggregate market behavior are remarkably accurate and easier to make. Building on this insight, we propose a new approach to cloud computing called *index-aware decision-making*, where cloud users and applications make their investment, trading and benchmarking decisions, in part, based on the characteristics of cloud indices. We hypothesize that index-aware cloud computing helps users extract better tradeoffs compared to current approaches.

6.1 The Case for a Cloud Index

As cloud spot markets continue to expand, the price variability across different servers and locations presents a new optimization opportunity to select servers based on their dynamic price characteristics. Prior work leverages this optimization to select servers for

various applications that offer the best risk-adjusted returns, which takes into account the cost of performance penalties due to server revocations [51, 52, 66]. While this work offers the potential for significant cost savings, the magnitude of these savings is not guaranteed but is based on future prices, and could ultimately be negative if prices change.

Thus, accurately predicting future server prices is important in both estimating the potential savings for different servers and in selecting the optimal server. As a result, a number of researchers and startups [15, 39, 45] have proposed more sophisticated techniques for modeling and predicting spot market prices. For example, in a recent whitepaper [63], Spotinst [39] claims to use an "...in-house prediction algorithm..." to "[choose] the most effective and most likely available EC2 Spot instance." Researchers have proposed numerous similar modeling and prediction techniques for EC2 spot prices [6, 18, 35, **?**, 7, 43, 71, 74]. As one example, DrAFTS is an online service that, given a bid price and duration, returns the expected probability of acquiring an individual spot instance for that duration [74, 6].

In general, prior spot prediction techniques have focused on predicting prices in *individual server markets*, which dictate a dynamic price for each OS configuration of each instance type in each availability zone (AZ) of each region of EC2. For example, an `m4.large` running Linux in AZ `a` of the us-east-1 region has its own dynamic spot price, which is distinct from other server configurations and types in other AZs and regions. In aggregate, there are ∼7600 individual server markets across EC2's global platform. Modeling and predicting the behavior of each of these markets presents multiple challenges. In particular, unlike prices in electricity spot markets, which correlate with weather metrics, such as temperature, and other routine behavioral patterns, e.g., days versus nights, it is less clear if server spot prices correlate with any easily-measured external variables. As a result, price prediction techniques are inherently limited, as the primary information they leverage for prediction is historical prices. In addition, there is no guarantee a one-size-fits-all model exists, as price characteristics are based on local supply/demand conditions that may differ across individual server markets, just as individual stock prices may exhibit widely different characteristics.

Investors face similar issues in financial markets when making investment decisions. Since predicting individual stock prices is challenging, investors base investment decisions,

in part, on the characteristics of broader market indices, such as the Dow Jones Industrial, the S&P 500, and the NASDAQ. With cloud computing platforms evolving into full-fledged markets, technology-focused companies face themselves in a similar situation. Since *compute-time is a core investment* for any technology-enabled comapny, a company that significantly reduces its compute-time costs, while limiting its risks, will gain a competitive advantage. We argue that, in order to effectively manage a cloud server portfolio, cloud users need broad market-based indices.

6.2 Crafting a Cloud Index

6.2.1 A Primer on Market Indices

Index, as used in economics and finance, is a statistical measure of changes in the value of a collection of items. As such, an index is a time-series capturing the movement of a closely related set of variables. 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 monetary policies on wages and taxes, interest rates and cost of living adjustments among others. However, the more visible indices are those of the stock market namely Dow Jones Industrial Average (DJIA), the Standard and Poor's 500 (S&P) and the NASDAQ Composite. Stock market indices are used to guide investment decisions in mutual funds, and are generally considered as broad indicators of a country's economy.

The S&P [37], started in 1923, is a stock market index comprising of 500 large publicly traded companies (with market capitalization $\geq \$6.1B$) that are broadly representative of the U.S economy. The S&P is calculated using the formula:

$$S\&P\ index = \frac{\sum_{\forall i} P_i * Q_i}{divisor} \tag{6.1}$$

Here, P_i is the price of stock i, and Q_i is its float-adjusted number of stocks, which excludes closely held portion of the stocks unavailable to investors. Thus, the numerator represents the weight adjusted value of the complete portfolio, which is likely in trillions of dollars. The *divisor* serves many functions, the simplest of which is to scale down the portfolio value to a more manageable number. But more importantly, it serves to maintains the index level consistent when the portfolio properties are changing. For instance, S&P can decide to replace a portfolio stock with a new one, or the company may split the stocks or spin-off a division. Thus, the divisor plays a critical role in providing a measure of the market that can be comapred across different points in time, solving what is known as *the index number problem* [37].

6.2.2 Methodology for Cloud Index

We compose a cloud index, on similar lines as other economic and financial indices, to represent the behavior of a chosen set of cloud markets. In this section, we describe our index construction methodology including its granularity, weighting, and sensitivity.

Granularity.

We define the index at three different granularities (i) global, (ii) regional, (iii) zonal, which correspond to the geographical location of the servers. Granularity determines the composition of the set of servers that go into computing an index. Choice of granularity is a function of application requirements and market characteristics.

Weighting.

Index weighting determines the relative impact that each constituent item has on the final index value. The commonly used weighting mechanisms include (i) *equal weighting*, where every items contributes equally. Example of this is DJIA, which refers to it as price-weighting. (ii) *size-proportional weighting*, where each item contributes proportional to its market size. For example, in S&P, the weight of each stock is the number of that stock available in the public market, and in CPI, the weight of each item is the number of that item contained in the market basket (iii) *attribute weighting*, where each item is weighted

as per the score its gets for its attributes. An example is the S&P 900 Growth index, which weights its stocks based on their growth prospects.

For cloud servers, size-proportional weighting is non trivial as neither the overall cloud pool capacity nor the available capacity are published. We use a hybrid of attribute weighting and equal weighting for computing CSI. In the first phase, we use attribute weighting to normalize the spot prices across a broad set of servers with varying capacities, and in the second, we combine these normalized prices using equal weighting. This approach is similar to the two-stage composition of CPI [46], where ther first phase accounts for the regional price differences and the seoncd stage aggregates the market basket prices across regions.

Normalizing.

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. Even then, cloud servers come in several dozen different configurations in order to cater to different classes of applications. 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. The choice of geometric mean is influenced by its ability to compare items that have different ranges and properties.

$$P_i^{norm} = \frac{P_i}{\sqrt{C_i * M_i}} \tag{6.2}$$

Here, P_i^{norm} 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. This quantity allows us to fairly compare two servers with different resource vectors. Once the prices are normalized to the server capacity, we take an arithmetic mean across all the servers to compose the index level. Thus in essense, each server contributes equally to the index level but only in proportion to its resource

efficiency. Putting it all together, we define the index to represents the unit price of renting (measured in cents/hour) a given set of servers.

$$Index - level = \frac{\sum_{i=1}^{N} P_i^{norm}}{N} \tag{6.3}$$

Consistency.

Given the definition of our index, it is trivial to incorporate market changes like introduction of new servers, permanent discontinuation or reconfiguration of existing servers. 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. To keep our indices consistent, we temporarily exclude all the 10× markets from index computation for as long as their prices don't fall below the cap level.

Extensions.

Several interesting extensions of the cloud index are possible. First,the indices could be trivially extended beyond its current geographical grouping to any other logical grouping of servers. For example, we can define an index to cover all the big cloud vendors. Second, we can define indices on the lines of the S&P 500 that sample the broad market yet give a reliable indication of the overall market behavior. Finally, the defintion could be expanded to include the statistical spread of the compsite spot servers. For example, the price volatility of the underlying spot servers may affect cloud applications as much as the index movement itself, and thus likely a useful feature to model.

6.3 Insights from Indices

In this section, we use cloud indices to examine the EC2 spot markets at global, regional, and zone levels, and characterize their salient attributes. To establish a benchmark, we first apply the index formulation to on-demand servers. While on-demand prices are fixed

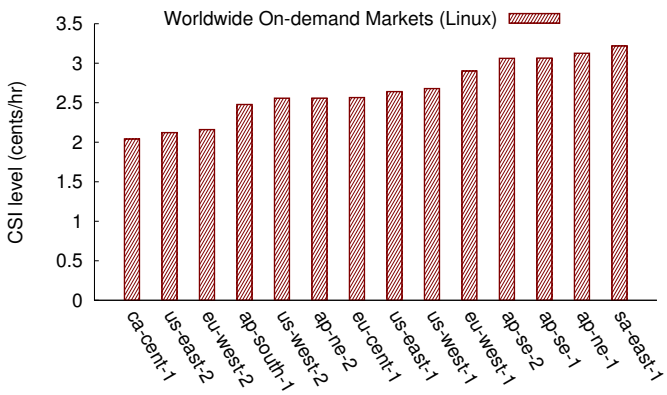


Figure 6.1: On-demand price levels across regions (prices as of May-1-2017)

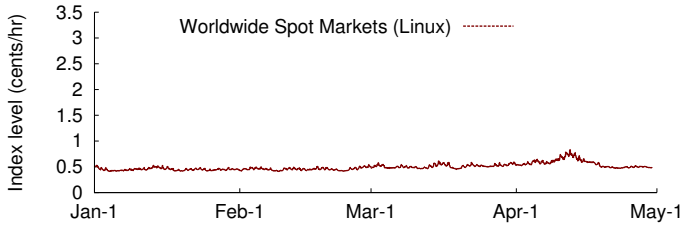


Figure 6.2: Global index for all 2406 Linux spot server markets across all 14 EC2 regions.

within a region, they vary across geographical regions as shown in Figure 6.1, which plots the normalized price per unit of compute (i.e., index-level) across all 14 of the EC2 regions. First off, we see that the price difference across regions is substantial. For example, SA-East-1 is 57% more expensive than CA-Central-1 on average. We also see that on-demand prices do not directly correlate with the regional electricity prices.

Figure 6.2 shows the global spot market index for all of the 2406 active Linux markets worldwide, with Y-axis plotting the normalized price of compute. 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. Also, we see that the characteristic peaky behavior of individual spot markets do not exist at the global level.

Next, we examine the spot markets on a regional basis to get an idea of each region's overall behavior with respect to the others. While the global spot market index reveals the general state of the market, per-region indices can inform users' choice of regions in the global market. This choice is important because migrating workloads across regions is

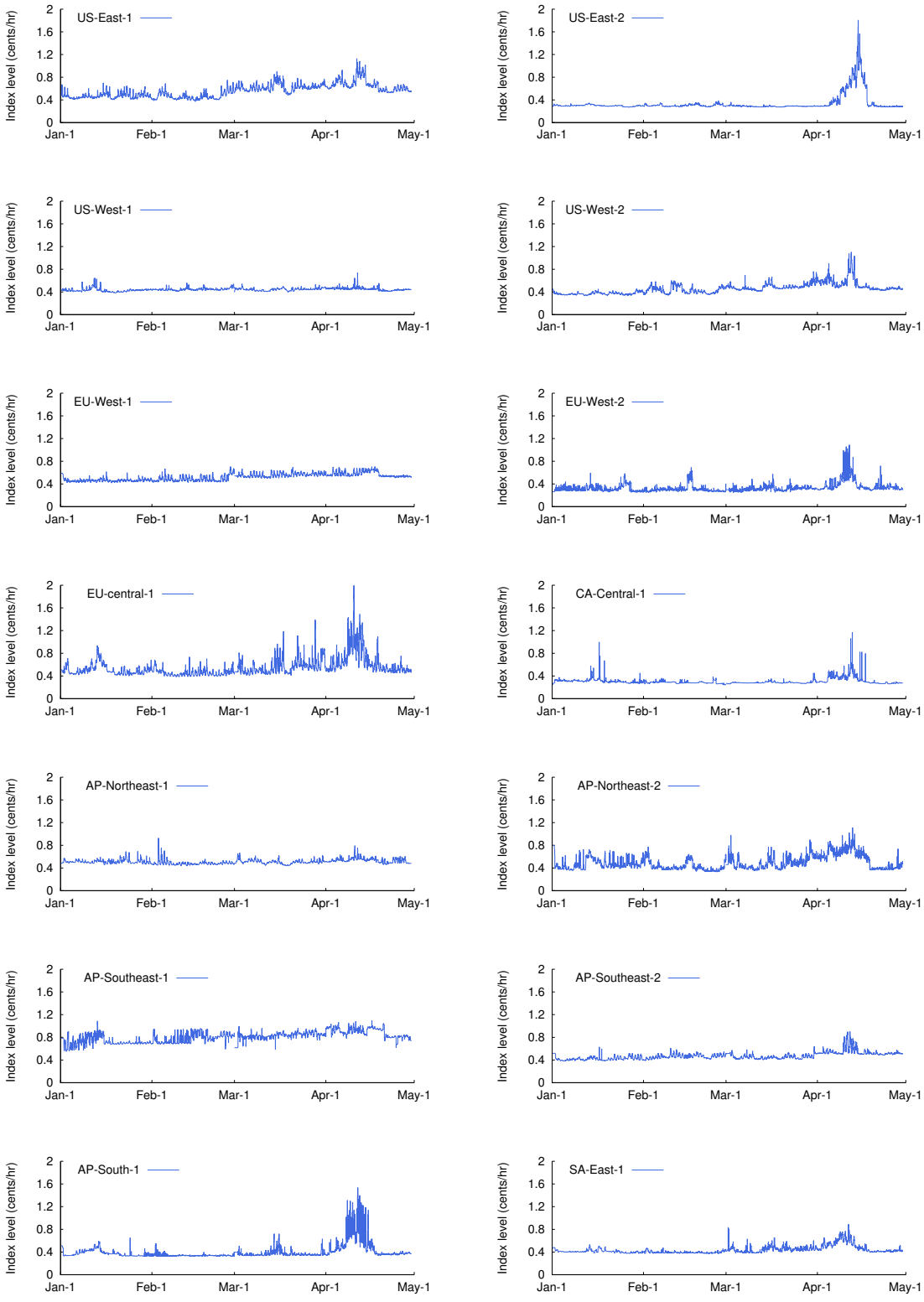


Figure 6.3: EC2 Regional Indices

prohibitively expensive. Figure 6.3 demonstrates that the regional index levels differ widely in both their magnitude and variance. For example, US-West-1 and EU-West-1 are highly stable with their index prices remaining at ∼16% of the on-demand price level for nearly the entire four months. In contrast, the EU-Central-1 exhibits much higher overall variance. Interestingly, regional indices highlight system-wide events that are not clearly visible at either the global level or individual server level. For example, ten of the 14 markets show an abrupt price peak and increased volatility around mid April.

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 level of EU-West-1 and EU-West-2 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, 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.

Finally, we analyze indices at the zone-level. Just as the regional indices revealed more features than the global index, zone level indices show even more entropy. Figure 6.4 plots the index level for all three zones of US-West-1 and US-West-2. While the regional indices of US-West-1 and US-West-2 in Figure 6.3 are largely stable with comparable averages, their zone level indices are much less stable and have larger price variances (2–3×). Additionally we see the price variations to be largely uncorrelated with each other, even within the zones of a given region. This is because each zone is a separate datacenter with its own usage patterns and administrative overheads such that the unused server capacity at a given time need not match across the physical datacenter boundaries. However, we do observe significant events like the mid-April price peak affect all three zones of US-West-2, albeit unequally.

Summary. *Our analysis confirms the intuition that, in general, the broader the index, the more stable and predictable its future prices. The global market index is generally more stable than the regional indices, which are more stable than the zone level indices, which are in turn more stable than the individual server markets.*

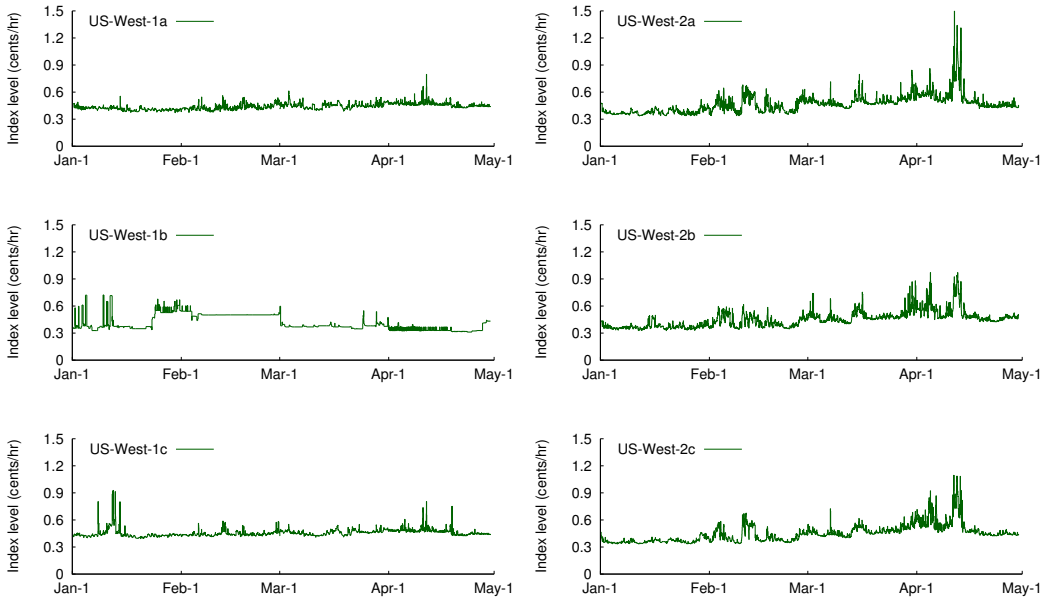


Figure 6.4: EC2 zone indices for US-West-1 and US-West-2

6.4 Proposed Work: Index-aware Decision-Making

Building on the insights from Section 6.3, we propose to explore index-aware decision-making as a new approach to financialized cloud computing. Specifically, we will investigate three research directions:

- **Index-aware Server Trading**

  How can "flexible" applications leverage broad market-level information in making server choices? Do the overhead of migration outweigh the benefits of trading? How to design system software that enable such trading in an application agnostic way? How do we adapt and extend trading methodologies from finance and economics to design cost-efficient server selection and trading policies?

- **Index-based Benchmarking**

  Do cloud-based systems need a benchmark? How does one compare the performances of systems developed at different times or deployed in different regions? How could cloud indices be used to develop standard benchmarks like SPEC and TPC?

- **Index-aware Investing**

  While spot server prices vary in short time scales, the prices of other server types change over longer time horizons. How do these trends affect consumption of cloud compute? Specifically, how to enable cloud consumers make decisions on investing in one type of contract over the other while building their cloud portfolio?

6.5 Status

We have composed a cloud index and applied it to over 6 months of EC2 spot market data, and 10 years of on-demand data. A preliminary version of this work has been published at [56]. We have also built and open-sourced our index computation at [3].

In the next few months, we plan to explore applications of cloud index including the ones listed in Section 6.4. Our goal is to establish the effectiveness of cloud indices and index-aware decision-making, so that a wider audience (even outside of the research community) can manage their cloud portfolio better.

CHAPTER 7

CONCLUSION AND REMAINING WORK

7.1 Conclusion

Cloud providers are evolving their infrastructure server offerings into a dynamic marketplace, where contract types vary in their pricing model, time commitment, resource guarantees, and risk exposure. While this presents a cost saving opportunity for customers, the current generation of system software frameworks are ill-prepared to leverage these dynamics. As a result, many applications simply resort to using the default on-demand servers despite their high costs.

Our work address the challenges of managing and benefiting from the newfound contract diversity in infrastructure clouds. We demonstrate that by "*financializing cloud computing*"—i.e., treating compute resources as investments, rationalizing about their risk-reward tradeoffs, and managing them in a way that adapts to changes in market prices—cloud applications can significantly reduce their compute costs while limiting their risks. As part of our work, we identify four significant problems and design system-level solutions that help cloud applications, transparently and automatically, overcome these challenges.

We present an asset pricing model, in Chapter 3, to help customers asses the savings they get from using transient cloud servers. In Chapter 4, we demonstrate how to employ fault-tolerance mechanisms as a cost-effective insurance against revocation risks of transient servers. Chapter 5 presents an alternative risk management technique that completely avoids paying the upfront cost of insruance by actively "trading" servers as market conditions change. Finally, we propose broad market-based indices that enable cloud users to make better predictions, benchmarking, trading and investment.

Project	Proposed Work	Target timeline
Automaton	SoCC camera-ready, and open source software	2 months (Sep 2017)
Cloud Index	Design, evaluation, and SIGMETRICS submission	3 months (Dec 2017)
Cloud Index	Build a public repository	1 month (Jan 2018)
Transient Guarantees	Extended version to IEEE TCC	2 months (May 2018)
Dissertation	Writeup and defense	2 months (July 2018)

Table 7.1: Proposed work and their timelines

7.2 Proposed Work

Table 7.1 summerizes the work proposed to be carried out over the course of this dissertation.

Our work on active server trading (Chapter 5) was recently accepted for publication at the ACM Symposium on Cloud Computing (SoCC). In the next month, we will work on the camera ready version, preparing Automaton software for public release, and participating in the symposium in Sep 2017.

Chapter 6 laid the groundwork for a broad market-based index for the cloud. We are planning to evolve this effort in two directions. First, we will explore the new approach of index-aware decision-making with a goal to develop (i) index-aware server trading (ii) index-based benchmarking and (iii) index-aware investing. In parallel, we would like to setup a public infrastructure for example, a website, where cloud users can explore markets via indices. We are tentatively targetting a submission to SIGMETRICS conference in Jan, 2018.

In chapter 3, 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. We will take up this work in the spring 2018, with a goal to submit it to IEEE Transactions on Cloud Computing.

Finally, we dedicate a portion of the time for dissertation write up, defense preparation as well as academic job interviews.

BIBLIOGRAPHY

[1] 10 Years of AWS. https://aws.amazon.com/10year/.

[2] Automaton. http://github.com/transientCloud/automaton.

[3] Cloud index. http://github.com/transientCloud/spot-index.

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

[5] Linux Containers. http://linuxcontainers.org.

[6] DrAFTS – Durability Agreements from Time Series for AWS Spot Instances. `http://predictspotprice.cs.ucsb.edu/`, Accessed March 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] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan OKeeffe, Mark Stillwell, David Goltzsche, Dave Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *OSDI*, November 2016.

[10] A AuYoung, L Grit, J Wiener, and J Wilkes. Service contracts and aggregate utility functions. In *HPDC*, June 2006.

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

[14] 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.

[15] Batchly, Inc. http://www.batchly.net/, September 2016.

[16] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*, October 2014.

[17] 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.

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

[19] Devin Carraway. Lookbusy - A Synthetic Load Generator. http://www.devin.com/lookbusy/.

[20] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term SLOs for Reclaimed Cloud Computing Resources. In *SoCC*, November 2014.

[21] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See Spot Run: Using Spot Instances for MapReduce Workflows. In *HotCloud*, June 2010.

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

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

[24] 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.

[25] Christina Delimitrou and Christos Kozyrakis. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *ASPLOS*, April 2016.

[26] Peter Desnoyers, Jason Hennessey, Brent Holden, Orran Krieger, Larry Rudolph, and Adam Young. Using OpenStack for an Open Cloud eXchange(OCX). In *IC2E*, March 2015.

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

[28] W. Guo, K. Chen, Y. Wu, and W. Zheng. Bidding for Highly Available Services with Low Price in Spot Instance Market. In *HPDC*, June 2015.

[29] A. Harlap, A. Tumanov, A. Chung, G. Ganger, and P. Gibbons. Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets. In *EuroSys*, April 2017.

[30] 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.

[31] X. He, R. Sitaraman, P. Shenoy, and D. Irwin. Cutting the Cost of Hosting Online Internet Services using Cloud Spot Markets. In *HPDC*, June 2015.

[32] B. Huang, N. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cumulon: Matrix-Based Data Analytics in the Cloud with Spot Instances. *PVLDB*, 9(3), November 2015.

[33] D Irwin, J Chase, L Grit, and A Yumerefendi. Self-recharging virtual currency. In *EconP2P*, August 2005.

[34] D Irwin, L Grit, and J Chase. Balancing risk and reward in a market-based task service. In *HPDC*, June 2004.

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

[36] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. Smart Spot Instances for the Supercloud. In *CrossCloud*, April 2016.

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

[38] S. Khatua and N. Mukherjee. Application-centric Resource Provisioning for Amazon EC2 Spot Instances. In *EuroPar*, August 2013.

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

[40] H. Liu. Cutting MapReduce Cost with Spot Market. In *HotCloud*, June 2011.

[41] A. Marathe, R. Harris, D. Lowenthal, B. de Supinski, B. Rountree, and M. Schulz. Exploiting Redundancy for Cost-effective, Time-constrained Execution of HPC Applications. In *HPDC*, June 2014.

[42] M. Mazzucco and M. Dumas. Achieving Performance and Availability Guarantees with Spot Instances. In *HPCC*, September 2011.

[43] M. Mazzucco and M. Dumas. Achieving Performance and Availability Guarantees with Spot Instances. In *HPCC*, September 2011.

[44] I. Menache, O. Shamir, and N. Jain. On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud. In *ICAC*, 2014.

[45] 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.

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

[47] 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.

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

[49] F Popovici and J Wilkes. Profitable services in an uncertain world. In *SC*, November 2005.

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

[51] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *EuroSys*, April 2016.

[52] P. Sharma, D. Irwin, and P. Shenoy. Portfolio-driven Resource Management for Transient Cloud Servers. In *SIGMETRICS*, June 2017.

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

[54] 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.

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

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

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

[58] 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.

[59] J. Shneidman, C. Ng, D. Parkes, A. AuYoung, A. Snoeren, A. Vahdat, and B. Chun. Why Markets Could (but don't currently) Solve Resource Allocation Problems in Systems. In *HotOS*, June 2005.

[60] Rahul Singh, David Irwin, Prashant Shenoy, and K. K. Ramakrishnan. Yank: Enabling Green Data Centers to Pull the Plug. In *NSDI*, April 2013.

[61] Rahul Singh, Prateek Sharma, David Irwin, Prashant Shenoy, and K. K. Ramakrishnan. Here Today, Gone Tomorrow: Exploiting Transient Servers in Data Centers. *IEEE Internet Computing*, 18(4), July 2014.

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

[63] Spotinst. The State of the Amazon EC2 Spot Market: Research, Conclusions, and Opportunities. `https://spotinst.com/white-papers/the-state-of-the-amazon-ec2-spot-market/`, 2016.

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

[65] S. Subramanya, A. Rizk, and D. Irwin. Cloud Spot Markets are Not Sustainable: The Case for Transient Guarantees. In *HotCloud*, June 2016.

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

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

[68] S. Tang, J. Yuan, and X. Li. Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In *CLOUD*, June 2012.

[69] W. Voorsluys and R. Buyya. Reliable Provisioning of Spot Instances for Compute-Intensive Applications. In *AINA*, 2012.

[70] C Waldspurger, T Hogg, and B Huberman. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[71] 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.

[72] 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.

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

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

[75] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. Blending On-Demand and Spot Instances to Lower Costs for In-Memory Storage. In *Infocom*, July 2016.

[76] Y. Yan, Y. Gao, Z. Guo, B. Chen, and T. Moscibroda. TR-Spark: Transient Computing for Big Data Analytics. In *SoCC*, October 2016.

[77] Y. Yang, G. Kim, W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B. Chun. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *EuroSys*, April 2017.

[78] S. Yi, D. Kondo, and A. Andrzejak. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *CLOUD*, July 2010.

[79] M. Zafer, Y. Song, and K. Lee. Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs. In *CLOUD*, 2012.

[80] S. Zaman and D. Grosu. Efficient Bidding for Virtual Machine Instances in Clouds. In *CLOUD*, July 2011.

[81] Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao. Dynamic Resource Allocation for Spot Markets in Clouds. In *HotICE*, March 2011.

[82] Liang Zheng, Carlee Joe-Wong, C Brinton, Chee Tan, S Ha, and Mung Chiang. On the Viability of a Cloud Virtual Service Provider. In *SIGMETRICS*, June 2016.

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