IndeX Marks the Spot: Cost-Predictive Server Hosting in Spot Markets

Supreeth Shastri and David Irwin
University of Massachusetts Amherst

Submission Type: Research

ABSTRACT

Cloud spot markets are gaining prominence as platforms that provide at-scale access to inexpensive computing for a wide range of data-intensive and scientific applications. However, applications that run on spot servers suffer from *cost uncertainty* since spot prices are market-based. This inability to predict future spot prices affects both customers and applications: former, because they cannot plan their IT expenses in advance and latter, because the spot price determines the availability and performance characteristics of spot servers. While researchers have proposed techniques for modeling and predicting prices of individual spot markets, their utility have been limited given the proliferation of spot markets, which now exceed 7600 on Amazon EC2.

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

1 INTRODUCTION

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

In order to maintain the cloud's illusion of at-scale computing that is available on-demand, cloud service providers typically provision their capacity for expected peak loads. As a result, significant portions (up to 40% [38]) of servers remain idle in cloud datacenters. To recoup some of the capital and operational expenses of maintaining this surplus capacity, which otherwise cannot be turned off, providers have begun offering them as *Transient servers*. A prominent example is Amazon's Elastic Cloud Compute (EC2) that uses a market-like mechanism to allow users to bid for its transient servers. As users place bids, EC2 continually evaluates the supply-demand dynamics of their idle capacity and determines the current clearing price of the "spot" servers. If a user's bid exceeds the spot price, they are allocated the server and billed at the current spot price. However, if the spot price rises above the user's bid level, EC2 reserves the right to revoke the server with a two-minute warning [7].

Since spot servers are characterized by frequent and deliberate revocations, EC2 offers them at significant discounts (typically 50-90%) compared to the regular servers that come with stronger availability guarantees. As a result, this new contract type suddenly provided access to inexpensive cloud computing for a wide swath of applications that were hither to inhibited by the high costs of on-demand servers. For example, the Fermilab Scientific Computing Division employed spot servers to dynamically scale up their compute capacity by 4× during the discovery Higgs-Boson [8]. Similarly, a group of machine learning and natural language processing researchers recently set the record [9] for the largest ever high-performance cluster on the cloud by using 1.1 million vCPUs on spot servers. While, spot servers offer a significant potential for cost savings, the magnitude of these savings is not guaranteed, is based on future prices, and could ultimately be negative if prices change unexpectedly.

Thus, any application that runs on spot markets suffers from *cost uncertainty*. While variable-priced and market-based allocation schemes are effective in determining the right price and in automatically balancing the supply and demand, they result in the volatility and unpredictability seen in the spot price traces. This is problematic at two different levels. First, customers are typically used to allocating fixed budgets for their IT expenses but spot markets make it difficult to plan such purchases in advance. This concern was significant enough to persuade some cloud providers to resort to fixed-pricing models. For example, both Google's Preemptible VMs and Microsoft Azure's Low-priority VMs are offered at ~30% of their equivalent on-demand prices. Second, it poses system level challenges for applications. This is because, only by accurately predicting future spot prices, an application can select optimal servers as well as minimize their revocations by bidding suitably.

While predicting future spot prices and selecting optimal spot servers for a given application are important, they are challenging for several reasons. First, EC2 spot markets are massive and complex: it comprises of ~7600 independently priced server "listings"

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SoCC '18, Oct 11–13, 2018, Carlsbad, CA, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

across 44 zones in 16 regions. By comparison, there are only around 6000 stocks listed across both the New York Stock Exchange and NASDAQ. While a number of researchers [5, 12, 18, 26, 34, 37, 40, 43] and startups [2, 22, 23] have proposed techniques for modeling and predicting spot market prices, there is no guarantee a one-size-fits-all model even exists as price characteristics are based on local supply-demand conditions. Second, EC2 spot markets exhibit price inversions and arbitrages such that a higher capacity server may be priced lower than a lower capacity one, or identical servers across zones may be priced differently based on the real-time supply-demand conditions. It is impossible for static prediction methods to account for such real-time inversions a priori. Finally, as application's resource usage changes over time, its choice of optimal server is unlikely to remain same throughout its lifetime.

However, a confluence of recent technological advancements in container virtualization [4, 6, 11, 39] and datacenter networking [3] as well as per-second billing model [10] is enabling an alternative approach to managing spot market volatility. For example, HotSpot [30] proposes to actively migrate applications to cope with and benefit from diversity in application usage and volatility in market behavior. While promising, a purely-reactive system that "hops" underlying servers in response to real-time market changes does not provide cost predictability. Additionally, greedy optimizations at a given time may not necessarily lead to optimal deployments overall, especially since the cost of migration is upfront while its benefits are in the future.

In this paper, we make a case for a hybrid approach that combines the predictive and reactive techniques in selecting optimal servers for flexible cloud applications. In doing so, we observe that investors face similar issues in financial markets when making investment decisions. Since predicting individual stock prices is challenging, investors base their decisions, in part, on the characteristics of broader market indices, such as the Dow Jones Industrial Average, S&P 500, and NASDAQ. While market indices provide a high-level benchmark, investors employ active trading to adjust their portfolio so that it continues to meet their investment targets despite the volatility of financial markets. We hypothesize that similar dynamics hold good for cloud spot markets as well i.e., (i) making price predictions at aggregate market-level would be more reliable than predicting at an individual server level, and (ii) knowing the benchmark for cost-estimates a priori enables reactive server management systems to achieve cost-efficiency without sacrificing availability. In evaluating our hypothesis, we make the following contributions:

Spot Market Analysis and Cloud Index. In contrast to the current practice of historical price-based analysis of spot markets, we take a first principles approach to observing the infrastructure-level realities in public cloud datacenters. Based on this, we derive two market properties as well as propose a novel cloud index to benchmark groups of spot servers. Applying the index to EC2 spot markets over a 6-month window, we highlight its salient features.

Cost-predictive Server Hosting. Extending concepts from finance and economics, we design a new mechanism for cost-predictive server hosting in variable-priced spot markets. Our mechanism, *index-tracking-by-server-hopping*, employs server hopping as a technique to maintain the overall cost-efficiency at or below the target index level. We analyze the properties of our mechanism, and design three server selection policies that illustrate the cost-availability tradeoff enabled by this framework.

Implementation and Evaluation. We implement[1] the cloud index, index-tracking mechanism, and server selection policies on the open-source HotSpot framework. We evaluate our system against two prior works: one that performs individual server predictions, and another that does reactive server migrations. Our evaluations on EC2 spot markets show that index-tracking policies are able to reliably predict and maintain the target cost-efficiency for a variety of flexible applications.

2 BACKGROUND AND MOTIVATION

Since the introduction of EC2 spot servers in late 2009, cloud spot markets have been an active area of research and commercialization with a goal to enable a wide range of applications to benefit from these inexpensive compute resources. In this section, we address the challenges of deploying flexible applications on spot servers, and motivate our approach towards solving those.

2.1 Market and Application Characteristics

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

In this work, we address the challenge of cost-uncertainty experienced in variable-priced spot markets. Towards doing so, we narrowly focus on two application categories: (i) *Long-running occasionally-interactive* applications like data sinks for Internet-of-Things (IoT) devices, BitTorrent file trackers, and cryptocurrency miners, and (ii) *Parallel synchronous* applications like Message Passing Interface (MPI) that are characteristic of scientific and high-performance computing. These applications are distinct from classical batch jobs as well as hyper-interactive web servers. Like batch jobs, these applications offer flexibility for migrating and restarting upon server failures but unlike batch jobs, their performance worsens with decreasing availability. Similarly, unlike web servers, these applications are able to tolerate occasional downtimes but just like them, an increase in availability boosts their performance. Thus, any cost prediction framework should include a consideration (and tradeoff) for server availability.

[1] Available at https://umass-sustainablecomputinglab.github.io/cloudIndex

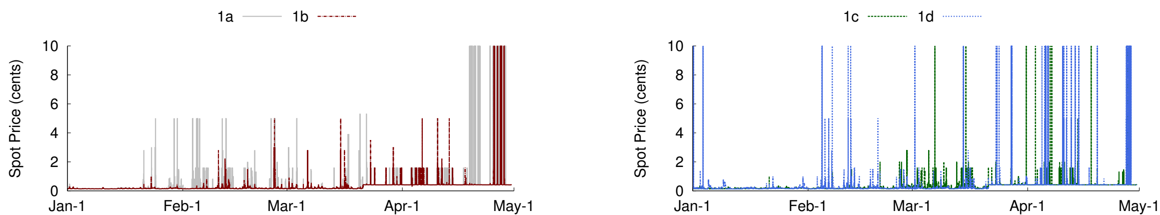


Figure 1: Price of a representative Linux server (r3.4xlarge) across four availability zones of the US-East-1 region.

2.2 Market Properties from First Principles

To model and predict the behavior of EC2 spot markets, most of the prior work analyzes the historical price traces from individual spot markets and then picks a queuing model that best describes the analyzed dataset. While intuitive, this approach faces several challenges that limit its utility (as described in §1). Therefore, we take a first principles approach to observing and understanding the physical infrastructure-level realities in cloud datacenters.

2.2.1 On Diversity

In order to allow users to select the best fit server for their application, cloud providers sell a large number of server types that differ in their resource capabilities. However, these numerous server types are carved out of a limited number of physical machines. For example, EC2 offers 23 general-purpose server types (in $T2$, $M3$, $M4$, and $M5$ series) that are internally hosted on just 4 types of physical machines[2]. While the total number of physical machines in a given datacenter does not change drastically in short timeframes (like hours, days or even weeks), the number of servers of each type is likely to vary more frequently (as governed by the administrative policies, supply-demand dynamics of different contract types etc.). In other words, despite $m4.large$ and $m4.16xlarge$ being sold in separate spot markets, their availability, revocation characteristics and in turn, their prices are not likely to be independent. Prior works have largely ignored this physical reality in assuming that price and revocation characteristics of different spot markets are independent and identically distributed (likely because the spot price traces under consideration did not explicitly reveal this).

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

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

2.2.2 On Stability

Though individual market's spot prices vary drastically (up to 10×), presumably based on the supply-demand dynamics of the given server type, the overall idle capacity of datacenter paints a different picture. In the first public release of its kind, Microsoft researchers published [15, 21] detailed workload- and utilization characteristics of Azure datacenters in 2017. While it was known a priori [38] that significant portions of datacenter resources remain idle, Azure traces shed light on the exact nature of this idleness: the actual CPU utilization varies by the order of half the datacenter capacity but the users are not dynamically scaling their allocated servers to match the actual real-time utilization. Thus, Azure datacenters do not experience large swings in server allocations either at the customer level or at the datacenter level. The reported median volatility for server allocations is 6.3% hourly, 2.6% daily, 3.2% weekly (at the datacenter level). These findings also corroborate with observations from multiple Google datacenters [14], where researchers proposed that large chunks of idle capacity experience higher availability (>98.9%) over the window of 6 months.

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

If compute was a fungible resource like oil and electricity, and all of datacenter's idle capacity was offered in a single marketplace to be consumed by perfectly flexible applications, then property-2 implies (i) that there would be a single unified clearing price like in the commodity spot markets, and (ii) that this clearing price would be largely stable and predictable (via the efficient market hypothesis [17] since the overall supply and demand are stable). Thus, flexible applications operating in this hypothetical setup would always pay the *fair market value* as well as have *predictable expenses.* Obviously, the assumptions on application's flexibility and compute's fungibility are not (yet) practical. However, in the next section, we explore a first order approximation that helps us benefit from this insight.

2.3 Market Indices

The market properties of §2.2 prompt us to analyze and model spot markets at aggregate levels instead of individual ones. Specifically, two granularities of aggregation are natural choices: (i) all the markets belonging to a given datacenter, and (ii) set of markets originating from a given server family (for e.g., all compute-optimized servers or all general-purpose servers housed in the datacenter). Towards collectively modeling a group of spot markets, we employ *market indices.*

A market index, in finance and economics, is a statistical measure of the value of a collection of items, and is useful in representing their collective movement in a time-series. For example, the Consumer Price Index (CPI) measures the changes in the price level of a pre-determined market basket of consumer goods purchased by typical households. Economists use the annual percentage change

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

in CPI as a measure of inflation, which in turn guides the monetary policies on wages and taxes, interest rates, and cost of living adjustments. Similarly, stock market indices like the Dow Jones Industrial Average, the Standard and Poor's 500 and the NASDAQ Composite report the statistical measure of a prominent set of publicly traded stocks, and are considered as broad indicators of the country's economy.

Thus, financial companies that engage in sophisticated market strategies to manage their cost-risk-performance tradeoffs, rely on these indices to evaluate their positions as well as to make investment decisions. We argue that with compute-time turning into a core investment, technology-enabled companies would benefit from an index that succinctly describes the behavior of cloud spot markets. More importantly, when applied to spot markets, market indices overcome several limitations of current approaches. First, index composition is based on directly-observed market properties, and not on indirect inferences from historical price traces. Second, by revealing the fair market value of idle compute capacity in real-time, the index provides a cost-benchmark for flexible applications. Lastly, it yields an open framework that can be easily extended and adapted to the needs of specific applications or market phenomena that the user is trying to model.

3 CLOUD INDEX

> "A bird's eye view is way different from a worm's eye view, when in fact, they are looking at the exact same thing."
>
> Unknown

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

3.1 Methodology

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

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

$$\hat{P}_i(t) = \frac{P_i(t)}{\sqrt{C_i \cdot M_i}} \quad (1)$$

Composition. Composition determines the set of spot server markets that go into computing the index. While this is primarily driven by the market properties of §2.2, it could be further trimmed to accommodate application's resource constraints or expanded to study

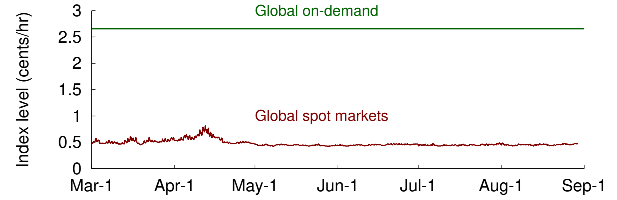


Figure 2: Index level and spread for the global Linux spot markets (2406 across all 14 regions).

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

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

$$\mathbb{I}(t) = \frac{\sum_{i=1}^{N} \hat{P}_i(t)}{N} \quad (2)$$

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

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

3.2 EC2 Spot Markets

The goal of applying the index on EC2 spot markets is twofold: first, to validate the market properties presented in §2.2, and second, to derive insights that can drive spot server selection. In the interest of space, we analyze indices only for Linux markets and only at select geographical locations.

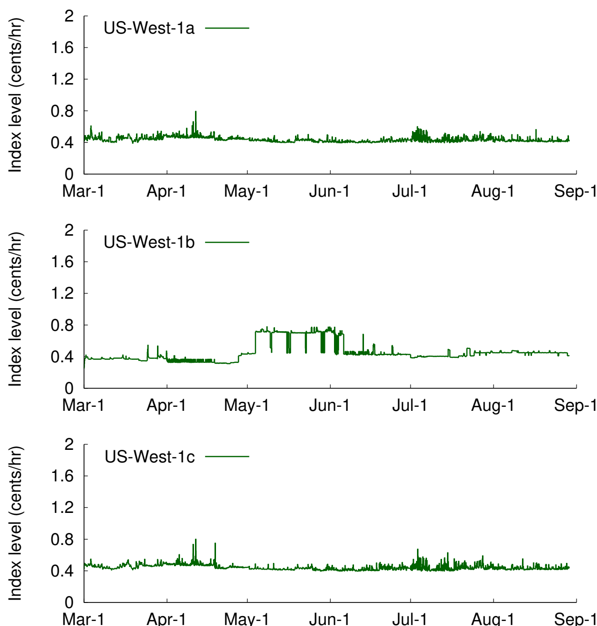


Figure 3: Indices for the three US-West-1 datacenters

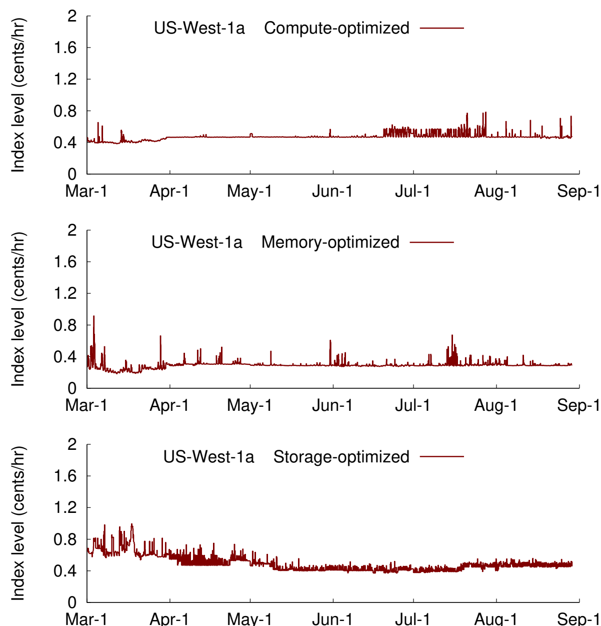


Figure 4: Indices of server families within a datacenter.

First, we observe the spot markets at the highest possible aggregation i.e., global level. Figure 2 shows the index for all 2406 active Linux markets worldwide, with Y-axis plotting the index-level and X-axis indicating the day of the year. The graph shows that EC2's spot market is remarkably stable, in aggregate, with prices around 0.5 cents/hr, which equates to 80% discount over the global on-demand average.

Second, we observe the aggregate markets at the datacenter level with Figure 3 plotting the index-level for the three zones of `US-West-1`. Contrasting these with Figure 1, we see that the characteristic peaky behavior of individual spot markets does not exist at the zone level. We also note that despite being located in the same geographical region, price variations across different zones are largely uncorrelated. This is because each zone is a separate datacenter with its own usage patterns and administrative overheads such that unused server capacity at a given time need not match across the datacenters.

Next, we decrease the granularity of aggregation to observe groups of servers belonging to three distinct set of families: compute-optimized, memory-optimized, and storage-optimized. Figure 4 shows these families for the `US-West-1a` zone. While there is increased volatility compared to the zone-level index, the values are still stable and predictable. Finally, we increase the levels of aggregation with Figure 5 demonstrating the corresponding regional index. As expected, it shows a higher level of stability and predictability compared to the zone-level indices, with price remaining at ~16% of the on-demand price level.

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

Given its generality, the index could be trivially extended to analyze the EC2 on-demand offerings. While on-demand prices are fixed within a region, they vary across regions as shown in Figure 6, which plots the index-level across all 14 of the EC2 regions. First off, we see that the price of compute varies substantially across regions with `SA-East-1` being 57% more expensive than `CA-Central-1` on average. Surprisingly, significant price differentials exist for geographically nearby regions as well. For example, index for `US-East-1` in Virginia is ~20% higher than `US-East-2` in Ohio. While such disparities may be due to the regional economic factors including price of energy, availability of technical staff, and climate conditions, it does provide significant cost saving opportunities for flexible applications (even without using spot servers).

Since on-demand prices vary across regions, the magnitude of cost savings from choosing particular regional spot markets also varies widely. For example, while the index levels of `EU-West-1` and `EU-West-2` (not shown here) hover around 0.45 and 0.3 cents/hour respectively, indicating a 33% price differential, this is reflective of the ~30% price differential that exists in their on-demand price levels. However, many price inversions do exist between on-demand and spot markets. For example, though `AP-Northeast-1` is slightly more expensive than `AP-Southeast-1` for on-demand servers, their spot market averages are flipped, with `AP-Northeast-1` offering 60% discount over `AP-Southeast-1` region as shown in Figure 7.

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

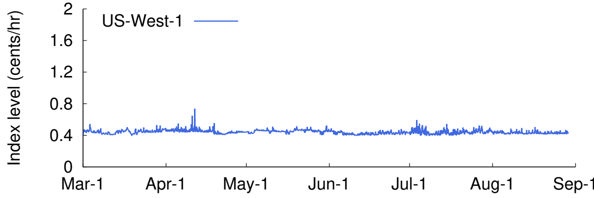


Figure 5: Index at the regional level

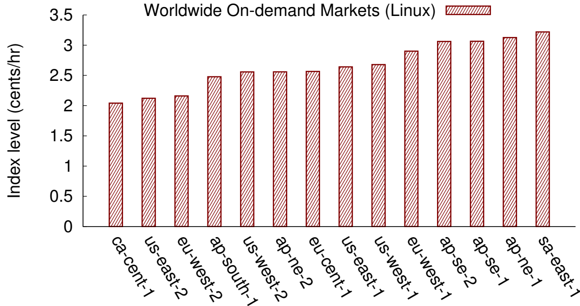
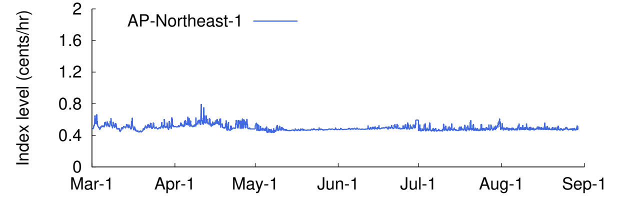


Figure 6: On-demand prices vary across regions.

4 SYSTEM DESIGN

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

We approach this in two steps: first, determine a broad set of candidate spot server markets that satisfy application's resource requirements. For this set, compute the cloud index to get the target cost-efficiency. Second, from amongst the candidate markets, select the best server (§4.3 outlines three policies for this selection) that meets the target level. If changes in market conditions or application characteristics render the selected server no longer meeting the target, then transparently migrate the application to another server that does (§4.2 proves why such a market always exists). Our design draws inspiration from two techniques followed in the financial markets: (i) *index funds*, which are financial instruments constructed to track the performance of a reference market index, and (ii) *active trading*, the strategy of actively trading stocks and other instruments in the short-term in order to benefit from market volatility. However, there are significant differences between financial instruments and cloud servers such that these techniques are not applicable as is. Remainder of this section addresses these challenges.



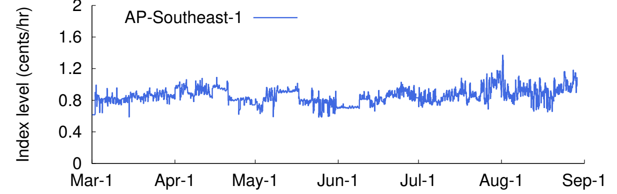


Figure 7: Indices showing price inversion across regions.

4.1 Index Tracking by Server Hopping

Fundamentals of Tracking. Index tracking is a rule-based investment mechanism with a goal to match the financial returns from a portfolio to the performance of the market index it tracks. Originally conceived to quell the notion that *one cannot buy the averages*, index funds have grown to account for ~20% of all the managed funds in the U.S. Their efficacy is rooted in the Efficient market hypothesis [17], which states that the stock prices fully reflect all available information such that the benefits of acting on information do not exceed the transaction costs. Simply, the hypothesis implies that one cannot consistently beat the market by predicting future prices of stocks. We have devised the cloud index on the same principle that while it is hard to predict the future prices of individual spot markets, it is possible to reliably predict the behavior of certain groups of markets in aggregate.

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

$$\mathbb{G}ain(t_1, t_2) = \sum_{t=t_1}^{t_2} (\mathbb{I}(t) - \hat{P}_i(t)) \cdot \sqrt{C_i \cdot M_i} \quad (3)$$

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

Server Hopping. This derives from the techniques such as day trading in financial markets, and loan refinancing in credit markets, where the objective is to benefit from favorable market conditions by actively trading one's assets or obligations. Recent advances in container virtualization, datacenter networking and per-second billing models have made it possible (and even attractive) to frequently migrate applications from one cloud server to another in response to real-time dynamics. For example, Supercloud [33] live

migrates applications in response to geographically shifting workloads, and HotSpot [30] migrates applications to more cost-efficient servers in spot markets. Server hopping algorithms are designed as localized greedy optimizations: they incur upfront migration costs in the hopes of future benefits.

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

$$\mathbb{L}oss(i, j) = (P_i(t) + P_j(t)) \cdot T_m \tag{4}$$

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

4.2 Properties of Tracking-by-Hopping

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

While this property is critical for the functionality of our algorithm, it is also the easiest to establish. In the base case, when the candidate set contains only one spot server market, the cost-efficiency of that market is the same as that of the market index (by definition 2). So, one can always hop back to the default spot market. Next, when the candidate set contains multiple markets, there needs to be at least one spot market whose cost-efficiency is better than or equal to that of the index level (this follows from the definition of market index, which is the average of the constituent market's efficiencies). Thus, there will always be a spot market whose cost-efficiency is better than or equal to the index level.

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

Necessary and sufficient conditions.

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

However, we derive a sufficiency condition that helps make localized greedy decisions instead of having to wait till the end of the execution to verify meeting the target cost-efficiency. Figure 8

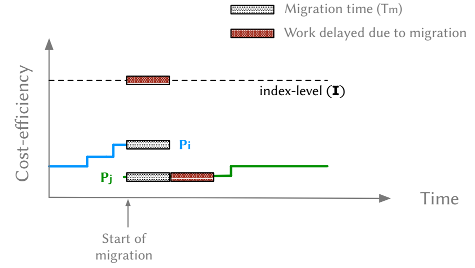


Figure 8: Illustrating the sufficiency condition to accommodate the overhead of migration.

illustrates an application migrating from server i to j with migration taking time T_m (shown by the gray area). In order to completely absorb the overhead of this migration, we need to account for not only the $\mathbb{L}oss(i, j)$ but also the actual work that has gotten delayed by migration (shown by the red area). Thus, if the cost-efficiencies of two spot markets satisfy the following condition with respect to the index-level, then hopping would not affect any previously accrued $\mathbb{G}ain$ on the index. Note that this condition is *sufficient but not necessary.*

$$\hat{P}_i(t) + (2 \cdot \hat{P}_j(t)) \leq \mathbb{I}(t) \tag{5}$$

On the efficacy of the mechanism.

Another advantage of tracking-by-hopping mechanism is that it *scales well with increased adaption.* As more users seek to achieve index-level cost-efficiency, we expect the market to become increasingly stable since everyone's target is the fair market value of the idle cloud capacity. In turn, this should reduce the number of server hopping required to maintain the desired cost-efficiency, thereby leading to an increased application availability. This is in contrast with the behavior induced by HotSpot, where every application is actively trying to hop to the most cost-efficient server, which could exacerbate the market volatility.

4.3 Server Selection Policies

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

Cost-centric Policy. The goal of this policy is to maximize cost savings by aggressively migrating to the *best-fit server* that is also cost-efficient. In order to determine this, we re-normalize the server's cost-efficiency from Eq-1 to take into account the actual resources utilized at time t, namely C_{util} and M_{util}.

$$\check{P}_i(t) = \frac{P_i(t)}{\sqrt{C_{util} \cdot M_{util}}} \tag{6}$$

Thus, cost-aware policy chooses the spot market that provides the best $\check{P}_i$ value at the given time. Since availability is a not concern, selections are triggered every time a better fit cost-efficient server emerges due to any changes in spot market or application behavior.

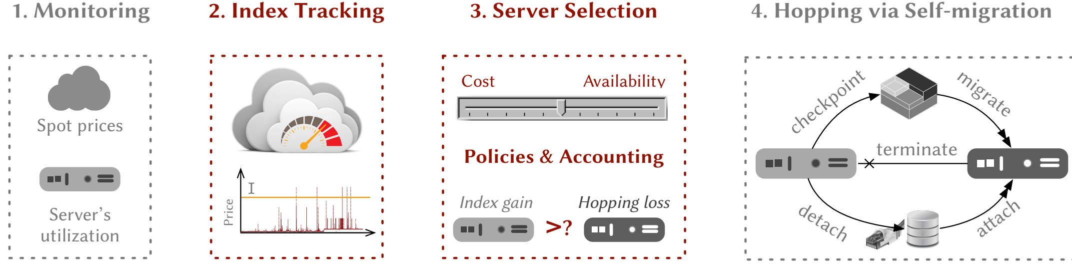


Figure 9: System architecture with HotSpot components boxed in gray and our extensions in red.

However, in order to maintain the target cost-efficiency, only those migrations that satisfy Eq-5 are carried through.

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

Balanced Policy. To mind the gap between the two extremes, we define a policy whose goal is to achieve a balance between higher cost efficiency and higher availability. We infer that higher a spot market's price variability, higher the risk of needing to migrate away. In order to balance this risk-reward tradeoff, we employ the *Sharpe ratio* [29], a statistical measure commonly used in finance to compute the risk-adjusted returns of an asset. We define balance factor as a variant of the Sharpe ratio,

$$\mathbb{S}_i(t) = \frac{\mathbb{I}_g(t) - \check{P}_i(t)}{\sigma_i} \tag{7}$$

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

5 IMPLEMENTATION

Since we propose a middle ground between fully-predictive and fully-reactive approaches to spot server management, we had several options to build on the prior work. However, given the ease of adding a predictive component (i.e., index tracking) to an already functional reactive system (that does server hopping), we chose HotSpot [30] as our base framework.

5.1 HotSpot Overview

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

Since we reuse the HotSpot framework, we also inherit some of its restrictions. First, due to LXC integration, we can only support stop-and-copy migration. Second, hopping semantics force applications to use remote storage and virtual network i.e., Elastic Block Storage (EBS) and Elastic Network Interface (ENI) respectively. Finally, HotSpot is architecturally decentralized i.e., each server manages itself without explicit coordination with others. Thus, efficient coordinated deployments in multi-node configurations may need a centralized orchestrator (we address this in §6.2).

5.2 Extending HotSpot

HotSpot is implemented in `Python` with additional integrations with EC2's `Boto3` library, `LXC` bindings and administrative `Shell` scripts. We retain the monitoring and hopping components but replace the cost-benefit analysis logic with index-tracking and server selection modules as depicted in Figure 9. We implement our extensions in ~850 lines of `Python`.

First, we build a standalone cost estimator utility that takes in application's resource constraints, then computes the market index-level for the selected availability zone, and finally predicts the overall cost to be incurred. This enables users to know their expenses before starting the workload. A library version of this utility is integrated in the index-tracking module, which in turn polls the monitoring engine once every five minutes to update the gain on index-tracking. This also triggers the server selection policy, which iterates through all the applicable spot markets to determine if server hopping is required. For the most populous zone (`US-East-1a` with 106 spot markets), this whole operation sequence of monitoring, tracking and server selection takes an average of ~2 seconds. We configure the migration module for direct memory-to-memory transfer as it maximizes application's availability. We find that the migration latencies observed in HotSpot

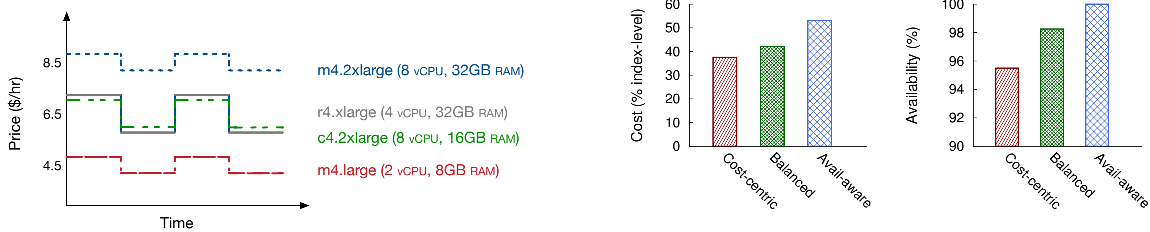


Figure 10: Spot market setup (left), and the performance tradeoffs (right) at the baseline configuration.

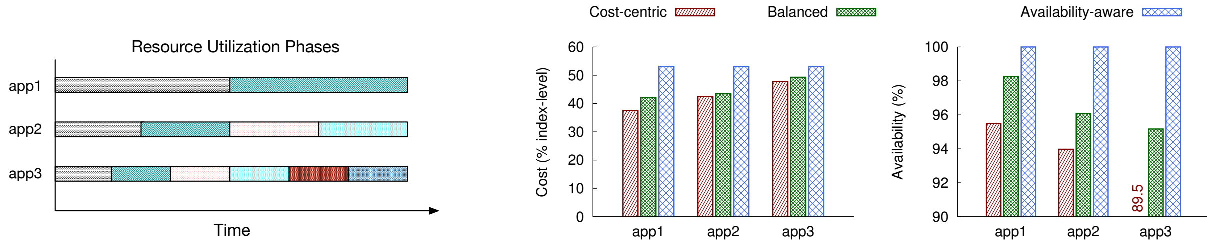


Figure 11: Performance of policies when application's resource utilization varies.

are still current: migration of up to 32GB RAM takes ~30 seconds (as it is bottlenecked by the EBS/ENI transfer happening in parallel), and the latency for larger memory sizes (up to 128 GB) increases linearly at the rate of ~1 second per GB.

6 EVALUATION

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

6.1 Prototype Experiments

In order to evaluate the server selection policies (described in 4.3), we have to be able to control application's and spot market's characteristics. Since the prototype is intended to run real workloads on EC2 instances with real-time prices, it limits our ability to control key parameters. Thus, in the following set of experiments, while our prototype is deployed and run on the EC2 platform, we stub out certain EC2 API calls (for e.g., real-time spot price querying). We also use an emulated job to better control application's CPU and memory usage. Below, we describe these setup, establish a baseline performance and then quantify the effect of varying key parameters.

Application. To predictably control the application behavior, we emulate the job using lookbusy [13]. Our job runs for an hour on the reference server `m4.2xlarge` and has two distinct resource utilization phases. In the first phase (lasting 30 minutes), it consumes 4 vCPUs and 16 GB of memory while in the second phase (the next 30 minutes), it consumes 2 vCPUs and 8 GB of memory.

Spot Markets. To ensure identical market conditions over different runs, we generate synthetic spot price traces for four spot markets: `m4.large`, `m4.2xlarge`, `c4.2xlarge` and `r4.xlarge`. These are chosen as their vCPU varies between 2-8 and memory capacity between 8-32, which cover the entire spectrum of our application's resource utilization. We model their prices as follows: `m4.large` has an average price of 4.5 cents per hour and a standard deviation of 0.5, `m4.2xlarge` has the same standard deviation but an average price of 8.5 center per hour, while `c4.2xlarge` and `r4.xlarge` have identical average price of 6.5, the former has a standard deviation of 1, while the latter has 1.1. Figure 10(left) gives an illustrative representation for these markets. For our experiments, the instantaneous spot price is computed randomly such that their average and standard deviation characteristics hold good. We use the same per-second billing model that EC2 operates on.

Baseline Result. Figure 10(right) shows both the cost incurred and availability achieved by the three different policies. We normalize the cost to that of a reference server running at the index-level cost-efficiency. We observe that all three policies perform better than the index predicted levels. Also, expectedly the cost-centric policy realized the cheapest run, and the availability-aware policy attained the highest availability. The balanced policy managed to be within 12% of the lowest cost, and 1.7% of the highest availability.

Changing Application Behavior. Next, we modify the baseline configuration of the application to exhibit more diversity in its resource consumption as depicted in Figure 11 (left). The resource

variations are such that the application could be executed on at least one of the four target spot markets at all loads. Figure 11 (right) shows the results of hosting the three different application configuration. First thing to notice is that the performance of the availability-aware policy does not change at all, which is not unexpected because this policy optimizes for stability and not savings. Next, we see that both balanced and cost-centric policies incur increasing overheads as the application's utilization gets bursty. As the markets have remained the same, the cost-efficiency gains of moving to a better fit server gets overtaken by the migration overheads. However, since the balanced policy does not react to changes as quickly as the cost-centric policy, its losses are less pronounced.

Changing Market Behavior. Finally, we vary the baseline market conditions to see its impact on the policies. We achieve this by changing the standard deviation. Figure 12 then shows how the policies perform under more volatile conditions. We plot the increase in market volatility (compared to the baseline) along the X-axis. The first graph shows that both cost-centric and balanced policies slightly improve their cost efficiencies when the market volatility increases. Interestingly, the availability policy, when forced to migrate under more volatile conditions, has managed to reduce its cost as a side effect of repeated migrations. However, the availability graph shows that all policies suffer, when markets are more volatile. Our experiments in §6.2 give a better sense of the current state of the EC2 markets as they use real spot price traces.

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

6.2 Simulation Experiments

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

Spot Markets. For these set of experiments, we use EC2's spot price traces from the `US-West-1` region between 1-MAR-2017 and 31-AUG-2017. We run three separate trials, one for each of its three zones (`1a`, `1b`, `1c`), whose index levels are depicted in Figure 3. While each zone consists of 79 Linux markets, every jobs considers only the set of markets that meet its minimum resource requirements.

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

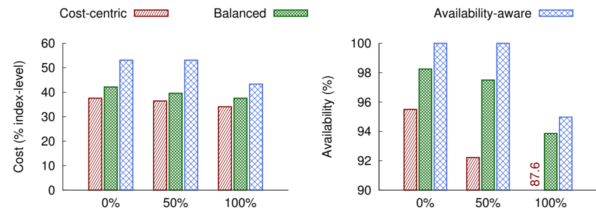


Figure 12: Policies under changing market volatility.

as the reactive component. We configure its server selection to be driven by the balanced policy.

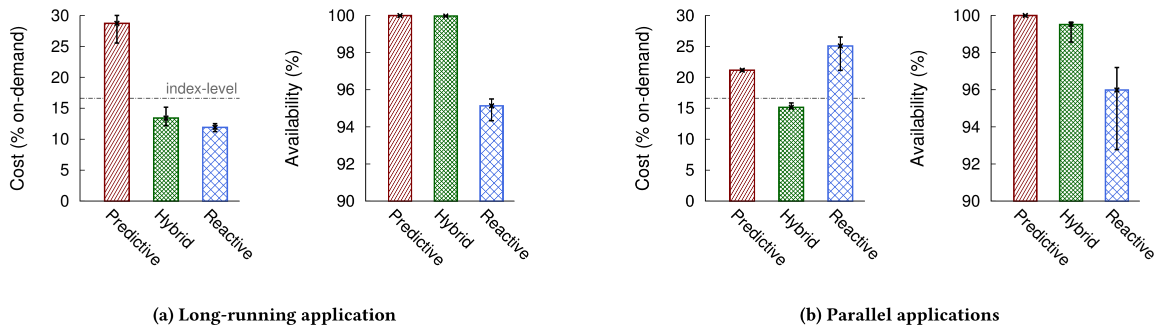
6.2.1 *Long-running Occasionally-interactive Applications*

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

Figure 13a shows both the overall cost and availability of running the application over the 6-month window. To establish a baseline, we simulate running the application on the cheapest on-demand server that meets the resource constraints, which happens to be `r4.large`. Then, we normalize the running cost of all techniques to this level. First, we see that all three approaches are substantially cheaper than on-demand hosting. But the reactive and hybrid schemes not only manage to meet the index-level cost-efficiency but also achieve ~50% cost reduction over the predictive approach. Next, in terms of availability, the predictive and hybrid schemes achieve three nines of availability whereas the reactive scheme manages only 95%. Under the hood, we observe that the predictive scheme experienced an average of 4.33 revocations, the reactive scheme migrated 4208 times with no revocations, and the hybrid scheme suffered 1 revocation and chose to perform 24.66 migrations.

6.2.2 *Parallel Synchronous Applications*

A staple of high-performance scientific computing, these applications are deployed in multi-server configurations with all the servers working in lock step, often involving significant data exchanges and synchronizations. Thus, a downtime for one server negatively impacts all other servers interacting with it. In this experiment, we evaluate how decentralized server management approaches cope with this parallel setup. To do so, we simulate running 1000 randomly selected jobs from the Google cluster traces [25].



(a) Long-running application

(b) Parallel applications

Figure 13: Comparing the fully-predictive, fully-reactive and hybrid server hosting systems on EC2 spot markets.

Internally, each of these jobs comprise of worker tasks (ranging from ∼10-500) that are run on separate servers but coordinate during their execution. Google traces report the CPU and memory consumption of every task at the granularity of 300 seconds. The run length of jobs vary between ∼10-720 minutes. We execute each job, at the time it arrives by choosing the best spot servers for each of its tasks as per the hosting technique. Within the simulator, we make three operational assumptions: (i) if a task gets revoked, the whole job is restarted, (ii) when a task is migrated to a new server, all other tasks of that job pause until the migration is fully completed, and (iii) we consider the job as completed only when all of its tasks are finished.

Figure 13b then shows the cost and availability of running these jobs. For the cost graph, we normalize the Y-axis to that of running all the jobs on the cheapest matching on-demand servers. We see that the hybrid scheme not only meets the index predicted cost levels but also comes out 30-40% cheaper than the other two schemes. The reactive scheme suffers from asynchronous migrations, where a small number of hopping nodes hold a large number of communicating nodes frozen for the duration of migration, thereby increasing the overall cost. This problem was not as pronounced in the hybrid approach since its policy naturally encouraged synchronous (and fewer) migrations. On the other hand, the predictive scheme improved its performance (compared to prior experiment) as it benefited from having a large number short jobs that in turn reduced the probability of revocation and also increased its ability to find better matched servers repeatedly. Availability paints a similar picture as before: predictive achieved four 9s, hybrid managed three 9s, and reactive mustered ∼96%.

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

7 RELATED WORK

To the best of our knowledge, this is the first work to apply market indices to analyze cloud spot markets, as well as propose a mechanism for cost-predictive server hosting on variable-price spot markets. While a preliminary version of this work appeared in [31], the current treatment differs in three significant ways: (i) we validate our intuition for using cloud indices via infrastructure-level observations in public cloud datacenters, (ii) we have a more specific goal (on cost predictability) and do not consider global trading, and (iii) we present a rigorous treatment of the index composition, tracking-by-hopping mechanism, and server selection policies. Below, we describe the related work in detail.

Spot market predictions. The first category of related work comprises of modeling spot markets with a goal to predict its future behavior. The earliest work came from Ben-Yehuda et.al. [12] in 2013, and has been followed since then by a large body of work including [5, 18, 37, 40, 41]. These work mainly focus on predicting the behavior of individual server markets, whereas our work proposes to analyze markets in aggregate. In conjunction with prediction schemes, researchers have also developed bidding strategies [26, 34, 43] for different types of applications. However, bidding policies are orthogonal to our work since our system migrates away from risky (i.e., cost-inefficient) markets naturally.

Financial concepts applied to spot markets. The next category is the application of financial and economic concepts to cloud spot markets. Prior efforts include creating an options market [36], adapting the modern portfolio theory [27], implementing active trading [30], proposing asset pricing [32], and composing derivative cloud markets [28, 35, 42]. However, none of these share our goal of realizing cost-predictive spot server hosting.

Market indices for non-cloud environments. Financial market indices have existed for a long time with the Standard and Poor's index dating back to 1923. Our work extends and adapts the index construction methodology of several indices including the Consumer Price Index [24], the S&P and Dow Jones [20]. Researchers have applied market indices to other spot markets like electricity [16]. However, the unique characteristics of compute-time namely, its state and the use-it-or-lose-it property make its application distinct from the prior indices.

System design aspects. Finally, our work derives several system design elements from HotSpot [30], SmartSpot [19], and Supercloud [33]. These include mechanisms and policies for container- and nested VM migration, automated server hopping, and decentralized server lifecycle management. However, the goals of these projects are distinct from ours. Both SmartSpot and Supercloud employ migration to lower access latency and improve resiliency but do not focus on spot market dynamics. While HotSpot uses

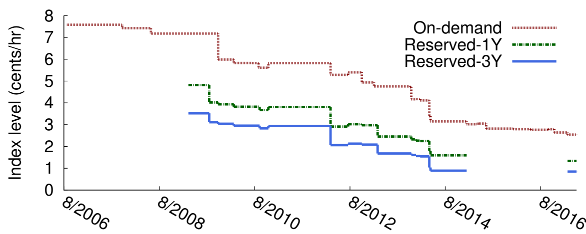


Figure 14: Index-levels of on-demand and reserved servers in the US-East-1 region, since EC2's inception.

automated server hopping to reduce server hosting costs, it makes no predictions on the resulting cost-efficiency.

8 DISCUSSION AND CONCLUSION

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

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

Beyond Spot Markets. In recent years, cloud computing platforms are rapidly evolving the IaaS offerings to cater to the diverse needs of cloud customers. While spot markets exhibit price and risk dynamism in short timescales of seconds and hours, other contract types like on-demand and reserved servers do so in terms of months and years. Cloud indices are a natural way to track this long-term evolution. For example, Figure 14 concisely represents the price trajectory of on-demand and reserved servers in the `US-East-1` region over the last decade. We posit that insights from cloud indices can drive informed investment decisions for cloud users.

REFERENCES

[1] 2017. AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/. (Accessed October 2017).
[2] 2017. Cmpute Inc. http://www.cmpute.io. (Accessed October 2017).
[3] 2017. Improved Networking Performance for Amazon EC2 Instances. https://aws.amazon.com/about-aws/whats-new/2017/09/announcing-improved-networking-performance-for-amazon-ec2-instances/. (Accessed October 2017).
[4] 2017. Linux Containers. http://linuxcontainers.org. (Accessed October 2017).
[5] Sara Arevalos, Fabio Lopez-Pires, and Benjamin Baran. 2016. A Comparative Evaluation of Algorithms for Auction-based Cloud Pricing Prediction. In *IC2E*.
[6] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. 1999. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI*.
[7] Jeff Barr. 2015. EC2 Spot Instance Termination Notices. https://aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/. (January 2015).
[8] Jeff Barr. 2016. Experiment that Discovered the Higgs Boson Uses AWS to Probe Nature. https://aws.amazon.com/blogs/aws/experiment-that-discovered-the-higgs-boson-uses-aws-to-probe-nature/. (March 2016).
[9] Jeff Barr. 2017. Natural Language Processing at Clemson University - 1.1 Million vCPUs and EC2 Spot Instances. https://aws.amazon.com/blogs/aws/natural-language-processing-at-clemson-university-1-1-million-vcpus-ec2-spot-instances/. (September 2017).
[10] Jeff Barr. 2017. Per-Second Billing for EC2 Instances. https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/. (2017).
[11] Muli Ben-Yehuda, Michael Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *OSDI*.
[12] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. 2013. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM TEAC* 1, 3 (2013).
[13] Devin Carraway. 2017. Lookbusy - A Synthetic Load Generator. http://www.devin.com/lookbusy/. (Accessed October 2017).
[14] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. 2014. Long-term SLOs for Reclaimed Cloud Computing Resources. In *SoCC*.
[15] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*.
[16] Paolo Falbo, Marco Fattore, and Silvana Stefani. 2010. A New Index for Electricity Spot Markets. *Energy Policy* 38, 6 (2010).
[17] Eugene Fama. 1970. Efficient Capital Markets: A Review of Theory and Empirical Work. *The Journal of Finance* 25, 2 (1970).
[18] Bahman Javadi, Ruppa Thulasiramy, and Rajkumar Buyya. 2011. Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In *UCC*.
[19] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. 2016. Smart Spot Instances for the Supercloud. In *CrossCloud*.
[20] S&P Dow Jones. 2014. Index Mathematics Methodology. (2014).
[21] Cinar Kilcioglu, Justin Rao, Aadharsh Kannan, and Preston McAfee. 2017. Usage Patterns and the Economics of the Public Cloud. In *WWW*.
[22] Frederic Lardinois. 2016. Spotinst, which helps you buy AWS spot instances, raises $2M Series A. TechCrunch. (March 8th 2016).
[23] Jordan Novet. 2015. Amazon pays $20M-$50M for ClusterK, the startup that can run apps on AWS at 10% of the regular price. (April 29th 2015).
[24] Bureau of Labor Statistics. 2015. The Consumer Price Index. https://www.bls.gov/opub/hom/pdf/homch17.pdf. (2015).
[25] Charles Reiss, John Wilkes, and Joseph Hellerstein. 2011. *Google Cluster-usage Traces: Format + Schema*. Technical Report. Google Inc.
[26] Prateek Sharma, David Irwin, and Prashant Shenoy. 2016. How Not to Bid the Cloud. In *HotCloud*.
[27] Prateek Sharma, David Irwin, and Prashant Shenoy. 2017. Portfolio-driven Resource Management for Transient Cloud Servers. In *SIGMETRICS*.
[28] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. 2015. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *EuroSys*.
[29] William Sharpe. 1994. The Sharpe Ratio. *The Journal of Portfolio Management* 21, 1 (1994).
[30] Supreeth Shastri and David Irwin. 2017. HotSpot: Automated Server Hopping in Cloud Spot Markets. In *SoCC*.
[31] Supreeth Shastri and David Irwin. 2017. Towards Index-based Global Trading in Cloud Spot Markets. In *HotCloud*.
[32] Supreeth Shastri, Amr Rizk, and David Irwin. 2016. Transient Guarantees: Maximizing the Value of Idle Cloud Capacity. In *SC*.
[33] Zhiming Shen, Qin Jia, Gur-Eyal Sela, Ben Rainero, Weijia Song, Robert van Renesse, and Hakim Weatherspoon. 2016. Follow the Sun through the Clouds: Application Migration for Geographically Shifting Workloads. In *SoCC*.
[34] Yang Song, Murtaza Zafer, and Kang-Won Lee. 2012. Optimal Bidding in Spot Instance Market. In *Infocom*.
[35] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. 2015. SpotOn: A Batch Computing Service for the Spot Market. In *SoCC*.
[36] Adel Toosi, Ruppa Thulasiramy, and Rajkumar Buyya. 2012. Financial Option Market Model for Federated Cloud Environments. In *UCC*.
[37] Cheng Wang, Qianlin Liang, and Bhuvan Urgaonkar. 2017. An Empirical Analysis of Amazon EC2 Spot Instance Features Affecting Cost-effective Resource Procurement. In *ICPE*.
[38] Josh Whitney and Delforge Pierre. 2014. *Data Center Efficiency Assessment*. Technical Report. Natural Resource Defense Council.
[39] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. 2012. The Xen-Blanket: Virtualize Once, Run Everywhere. In *EuroSys*.
[40] Rich Wolski and John Brevik. 2016. Providing Statistical Reliability Guarantees in the AWS Spot Tier. In *HPC*.
[41] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. 2017. Probabilistic Guarantees of Execution Duration for Amazon Spot Instances. In *SC*.
[42] Liang Zheng, Carlee Joe-Wong, Christopher Brinton, Chee Tan, Sangtae Ha, and Mung Chiang. 2016. On the Viability of a Cloud Virtual Service Provider. *ACM SIGMETRICS Performance Evaluation Review* 44, 1 (2016).
[43] Liang Zheng, Carlee Joe-Wong, Chee Tan, Mung Chiang, and Xinyu Wang. 2015. How to Bid the Cloud. In *SIGCOMM*.