HotSpot: Automated Server Hopping in Cloud Spot Markets

Supreeth Shastri and David Irwin
UMass Amherst
shastri,deirwin@umass.edu

ABSTRACT

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

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;

KEYWORDS

Transient Server, Spot Market, Revocation, Price Risk, Hopping

1 INTRODUCTION

Infrastructure-as-a-Service cloud platforms (IaaS) sell remote access to computing resources, i.e., co-located bundles of CPU, memory, and storage, in the form of virtual machines (VMs). Recent forecasts predict that spending on public IaaS clouds will increase by more than 4.5× over the next decade [19]. This growth has spurred these platforms, such as Amazon's Elastic Compute Cloud (EC2) and Google Compute Engine (GCE), to expand their offerings to attract new users. As a result, IaaS clouds now offer a wide range of Service Level Objectives (SLOs) for VMs that differ in their cost model, price level, performance guarantees, and risk exposure [1, 5]. In particular, IaaS clouds have begun to offer a new type of *transient VM* [37, 38], which they may take back, or *revoke*. Transient VMs enable platforms to earn revenue from their idle capacity, while retaining the flexibility to reclaim it to satisfy requests for higher-priority *on-demand VMs*, which platforms try not to revoke. Transient VMs are attractive, since platforms offer them for 50-90% less price than on-demand VMs to compensate for their *revocation risk*.

EC2 allocates its variant of transient VMs, called *spot instances*, using a dynamic market-like mechanism [2]. Users place a bid for VMs, such that if a user's *bid price* exceeds the VMs' current *spot price*, EC2 allocates the VMs to the user, who pays the spot price for them. However, if the spot price ever rises above the bid price, EC2 revokes the VMs after a brief warning [11]. EC2's documentation states that a VM's spot (or "clearing") price "...fluctuates periodically depending on the supply and demand of Spot instance capacity," and implies that it is set equal to the lowest winning bid in a continuous multi-unit uniform price auction [8]. EC2's spot market is complex: it sets a different dynamic spot price for each type of VM in each Availability Zone (AZ)[1] of each region, and currently includes over 7,500 separate VM "listings" across 44 AZs in 16 Regions with announced plans for 14 more Regions in the future [3]. By comparison, there are only around 6,000 stocks listed across both the New York Stock Exchange and the NASDAQ.

Many researchers [20, 22, 31, 33, 34, 40, 47–49] and startups [4, 24, 28] are actively working to exploit low price, high risk spot VMs. Prior work primarily focuses on managing revocation risk, since revocations are the defining characteristic of transient VMs and unmodified applications typically do not perform well when servers are frequently revoked. The general approach is to treat revocations as failures and then use fault-tolerance mechanisms, particularly periodic checkpointing, to reduce their impact. This approach is akin to buying insurance, where the fault-tolerance mechanism's overhead represents the "premium" an application pays, while its ability to limit the amount of lost work after a revocation is the "payout." However, spot VMs also expose applications to another form of risk: *price risk*, or the risk that a VM's price will increase relative to others. Since spot prices vary across hundreds of different types of VMs, flexible applications can mitigate price risk by moving to the VM that offers the current lowest cost.

To enable this flexibility, we present HotSpot, a resource container that automatically "hops" spot VMs—by selecting and self-migrating to new VMs—as spot prices change. Our key insight is that applications can proactively and transparently migrate to spot VMs that currently offer the lowest cost. An important design goal of HotSpot is simplicity: to use it, applications need only select and run a HotSpot-enabled VM image that requires little configuration. Applications then execute inside a resource container, while

[1] Each AZ is akin to a separate data center.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3132017

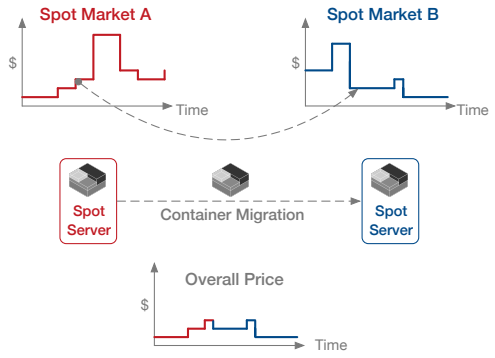

Figure 1: When the price of HotSpot's host VM rises, it self-migrates or "hops" to another VM with a lower cost.

HotSpot's systems-level monitoring and migration functions run transparently in the host VM. Thus, HotSpot is self-contained, requiring no application modifications or external infrastructure, as its functions execute within its current host VM.

HotSpot executes a *migration policy* in its host VM that determines when to migrate the container to another VM based on the transaction costs (from vacating a VM early and briefly double paying for it) and expected cost savings. As a side-effect of VM hopping to minimize cost, we show that HotSpot is also able to reduce the number of revocations without degrading performance. Our hypothesis is that VM hopping is a useful approach for managing spot VMs that can transparently lower cost and reduce risk compared to prior fault-tolerance-based approaches. In evaluating our hypothesis, we make the following contributions.

Market-level Analysis. We analyze market-level characteristics and make a number of observations, including: price risk is more dynamic than revocation risk, with the market's lowest-cost VM type changing frequently; the lowest-cost VM tends to also have a low revocation risk (as a high discount reflects low demand relative to supply); and a VM's normalized cost per unit of resource is independent of its capacity. Our observations show an opportunity to lower an application's cost via automated VM hopping without increasing its revocation risk or decreasing its performance.

HotSpot Design. We design a self-migrating server that runs applications within a resource container, and then executes a migration policy to determine when and where to migrate the container to maximize cost-efficiency (in terms of cost per unit of resource utilized). In particular, the policy only hops VMs if it expects the savings to outweigh the overhead of migration.

Implementation and Evaluation. We implemented HotSpot on EC2 using Linux Containers (LXC) [6], and publicly released it.[2] We evaluated and compared HotSpot's cost, risk, and performance against using i) on-demand VMs, ii) spot VMs without fault-tolerance (as in SpotFleet [12]), and iii) spot VMs with fault-tolerance (as in SpotOn [40]). We performed this evaluation using a small-scale prototype, and at large scales over a long period in simulation using a production Google workload trace [30, 42] and publicly-available EC2 spot price traces. Our simulation results show that HotSpot is able to lower cost and reduce the number of revocations without degrading performance.

[2]https://github.com/sustainablecomputinglab/hotspot

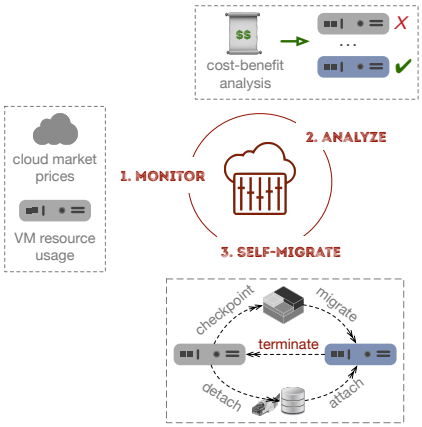

Figure 2: Depiction of HotSpot's basic control loop, which monitors spot prices and application resource usage, determines when and where to self-migrate based on its migration policy, and then executes the migration.

2 BACKGROUND AND ANALYSIS

HotSpot is motivated by both the maturing of systems-level migration for virtualized cloud hosts, e.g., via resource containers [6, 10] or nested virtualization [13, 36, 43], and continuing advances in data center networking, which are reducing the overhead of migrating memory state and accessing remote disks. Figure 1 illustrates HotSpot's basic function: when a VM's price spikes, HotSpot migrates its container to another VM with a lower price to maintain a high *cost-efficiency*. We define cost-efficiency as the cost (in dollars) per unit of resource an application utilizes per unit time. As we discuss in §3.1, we define utilization based on a VM's average CPU utilization. Figure 2 depicts HotSpot's basic control loop, which i) monitors real-time spot prices across the market and application resource usage, ii) uses the information to determine when and where to migrate based on its migration policy, and then iii) executes the migration. These functions are hidden from applications, which run within an isolated virtualized environment—a resource container in our prototype—capable of systems-level migration.

To understand the benefits of automated VM hopping, especially when compared with using prior fault-tolerance-based approaches on spot VMs, we analyze EC2's market-level characteristics and make a number of observations. These observations demonstrate an opportunity to increase cost-efficiency via VM hopping, while reducing revocation risk (independent of applying additional fault-tolerance mechanisms) without degrading performance.

Spot Price Data. Our analysis below and in §5.2's evaluation uses spot price data from 2017-03 to 2017-06 across five AZs in the `us-east-1` region, which has the most VM types of any EC2 region. Across all five AZs, the region has 402 VM types from eight VM families. We have included a link to this data with the public release.

2.1 The Importance of Price Risk

While using fault-tolerance-based approaches on spot VMs offers significant cost savings relative to using on-demand VMs, revocations are not frequent events in EC2's current market, and thus the savings relative to using spot VMs without any fault-tolerance is often not significant. To understand why, recall that users do not pay

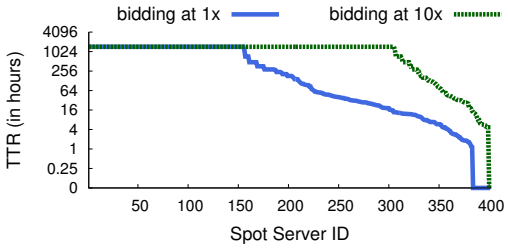

Figure 3: The average Time-to-Revocation (TTR) for 402 Linux spot VMs in EC2's `us-east-1` region when bidding at the on-demand price and 10× the on-demand price over a two month period. The average TTR across all servers is ~25 and ~47 days, respectively, for 1× and 10× bids.

their bid price for VMs, but instead pay the spot price. As a result, there is no penalty for bidding high, as long as applications are flexible enough to switch to lower cost VMs if their current VM's price rises [32]. Thus, even a simple strategy that bids well above the on-demand price is highly effective, since applications can prevent revocations by shifting to a cheaper fixed-price on-demand VM once the spot price rises to near the on-demand price, and before it comes close to the bid price. Since a VM's spot price rarely exceeds its on-demand price, the revocation rate at the on-demand bid level is low for most spot VMs. As we discuss in §3.2.3, HotSpot actually bids 10× the on-demand price (the maximum bid EC2 allows [7]), further reducing the revocation rate.

Figure 3 illustrates this point by showing the average Time-to-Revocation (TTR) over a two month period (from 2017-03 to 2017-04) for 402 Linux spot VMs across five AZs of the `us-east-1` region when bidding the on-demand price and when bidding 10× the on-demand price. As the graph shows, while a few VMs have low TTRs, the vast majority of VMs have high TTRs. The horizontal lines show the number of VMs that did not experience any revocation over the two month period, which includes >35% and >75% of spot VMs when bidding the on-demand price and 10× the on-demand price, respectively. Overall, the average TTR across the 402 Linux spot VMs in `us-east-1` is ~25 and ~47 days when bidding the on-demand price and 10× the on-demand price, respectively.

These results reflect that spot prices often experience long periods of stability interspersed with short periods of volatility, as illustrated in Figure 4. In this case, the `m4.large` spot VM in `us-east-1a`[3] maintains a low and stable spot price for much of the four month period, while experiencing a few highly volatile periods. As the graph shows, during volatile periods, the spot price can rise higher than the on-demand price. These steep rises could occur for many reasons including: "convenience bidding" where users bid high assuming the spot price will not spike and do not vacate VMs when it does [16]; EC2 reclaiming spot VMs by artificially raising their price; or unavailability of on-demand VMs that increases demand for spot VMs [29]. Regardless of the reason, applications should react to price spikes by vacating high-priced spot VMs.

Our analysis above indicates that revocation risk in the current market is low, although it could increase in the future, especially if EC2 alters its bidding rules such that users pay their bid price instead of the spot price, or if users adopt our optimizations and

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

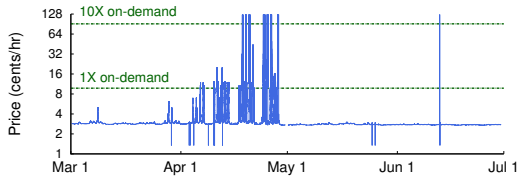

Figure 4: Example spot price trace for an `m4.large` VM with long periods of price stability and short periods of volatility.

those proposed in prior work. In contrast, the market's price risk—or the risk that a VM's price will increase relative to others—is much higher than the revocation risk. We quantify price risk by measuring the average difference between the price of each AZ's cheapest VM (in terms of its normalized cost per unit of resource) at t_0 and its price at each time $t > t_0$ as the market's cheapest VM changes. In this case, we normalize relative to a VM's EC2 Compute Unit (ECU)—Amazon's measure of a VM's integer processing power [9].

The difference between the price of the cheapest server at t_0 and the price of the cheapest server as it changes reflects the cost savings possible from hopping VMs assuming the application i) can fully utilize a VM of any capacity, ii) incurs no overhead when migrating between VMs, and iii) does not experience revocations. Under these assumptions, any approach that does not hop to the cheapest VM incurs a higher cost. Note that we relax these assumptions in HotSpot's design (§3), since in practice applications cannot always fully utilize VMs of any capacity and do incur an overhead when migrating. As a result, our analysis here only sets an upper bound on HotSpot's cost savings, and does not reflect the migrations HotSpot would actually make.

Figure 6 shows the potential cost savings from hopping VMs within each AZ of the `us-east-1` region over two months starting 2017-03. The average savings in each AZ is between 15% and 65% with an average of 33% across all AZs. Since the results are dependent on the cheapest VM's price at the start of each interval, the error bars reflect the maximum and minimum savings from 10 randomly selected start times within the two month period. The low minimum savings reflect times where the price of the cheapest VM at t_0 remained stable and was always near that of the dynamic cheapest VM. Note that the figure represents additional savings relative to using spot VMs without hopping, which is already significantly cheaper (~50-90%) than using on-demand VMs.

Figure 5 then shows the average time until the cheapest VM in the market changes, which we call the Time-to-Change (TTC), on the y-axis for each AZ of the `us-east-1` region. In this case, the cheapest VM across the 402 spot VMs changes every 1.1 hours, which shows that there is an opportunity to reduce cost by migrating to the cheapest spot VM. Since the TTC is two orders of magnitude less than the TTRs in Figure 3, applications are much more likely to experience a change in the cheapest spot VM during their execution than a revocation. Of course, each change in price of the cheapest spot VM might be small relative to the cost of a revocation.

2.2 Revocation Risk and Performance

As mentioned above and discussed in §3, HotSpot containers migrate to new cloud VMs to maximize their cost-efficiency. Thus, the migration policy does not consider either revocation risk or application performance, which are both important metrics. In general, at any time, the lowest cost VM is not necessarily the one with

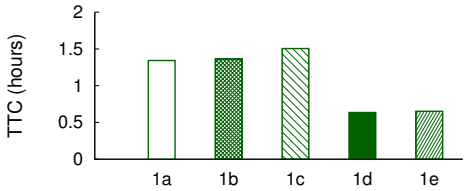

Figure 5: The Time-to-Change (TTC) for the cheapest VM in each of AZs of the `us-east-1` region over a two month period. The average TTC across all AZs is 1.1 hours.

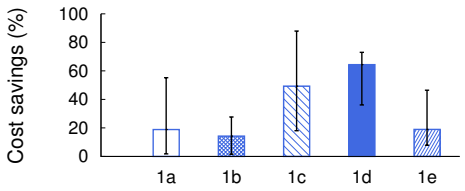

Figure 6: Ideal cost savings from automated VM hopping within each AZ in the `us-east-1` region over one month.

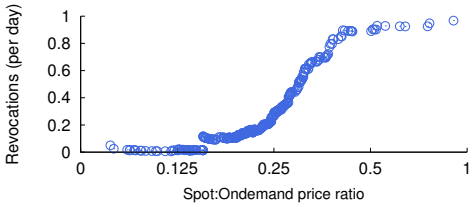

Figure 7: The x-axis is the spot-to-on-demand ratio, while the y-axis is the average revocation rate across spot VMs when the spot price is less than or equal to the x-axis value.

the absolute lowest revocation risk or highest capacity. As a result, migrating to optimize cost-efficiency has the potential to increase revocation risk and decrease performance. Of course, HotSpot could define a migration policy that also considers revocation risk and performance. Configuring such a migration policy would require users to compare and weight the relative importance of each metric. Fortunately, as we show below, hopping VMs to minimize cost tends to also lower revocation risk and does not decrease performance on average. Thus, HotSpot's migration policy focuses on minimizing cost, and does not consider revocation risk or performance.

2.2.1 Revocation Risk. Prior work generally estimates revocation risk in terms of a spot VM's TTR at a given bid level based on its historical spot price over some previous time period, e.g., a few days to weeks. However, since HotSpot frequently switches VMs, its revocation risk is not a function of any single spot VM's TTR, but the TTR of the multiple VMs it uses (weighted by the time it spends on them). Our key insight is that there is a relationship between a VM's instantaneous revocation risk and its current spot price relative to its on-demand price. This relationship derives from the observation that the lower the ratio of the spot price to the on-demand price, the further away the supply/demand balance is from being constrained and thus causing a spike in prices that triggers revocations to occur. That is, assuming the spot price is based on supply and demand (as Amazon claims [8]), the lower a VM's spot price relative to its "risk free" on-demand price, the greater the change in the balance of supply and demand required for the spot price to rise above the on-demand price. As a result, a spot VM's spot-to-on-demand price ratio is an indirect measure of its instantaneous revocation risk relative to other spot VMs.

Figure 7 illustrates the relationship between the revocation risk and the spot-to-on-demand ratio across 402 Linux spot VMs in the `us-east-1` region from 2017-03 to 2017-04. The graph shows that, as the average spot-to-on-demand ratio decreases, the average revocation risk also decreases. While the spot VM with the lowest spot-to-on-demand ratio is not always the same as the most cost-efficient one at any time, the most cost-efficient VM often has a low spot-to-on-demand ratio. This correlation exists, in part, because on-demand prices for VMs in the same family are uniform when normalized per unit of resource, so a low normalized spot price implies a low spot-to-on-demand ratio relative to other spot VMs in the same family [1]. The cost per ECU-hour for VMs in different families is also similar. To illustrate, Figure 8 shows the on-demand price per ECU-hour for all families in the `us-east-1` region. The c4, m4, r4, and i3 families range from 1.25-2.56 ¢/ECU-hour, while the more specialized memory-optimized (x1), storage-optimized (d1), and GPU instances (p2) have a higher normalized cost.

However, as we discuss in §3, HotSpot normalizes spot prices per unit of resource *utilized*, so variations in an application's workload also affect a VM's cost-efficiency. In this case, we define utilization as the VM's average CPU utilization. As a result, if a VM has a low average CPU utilization, e.g., ≪100%, then it is wasting CPU capacity, which increases the relative cost of the CPU capacity it utilizes. Thus, there is the possibility that the VM with the lowest spot-to-on-demand ratio could have a high capacity that is over-provisioned for the application and wastes resources, resulting in a low cost-efficiency. If only high-capacity VMs over-provisioned for an application were to have a low spot-to-on-demand ratio, it is technically possible for the most cost-efficient VM to have a high revocation risk. However, we have not seen this scenario occur, as there are usually many spot VMs at all capacities with low spot-to-on-demand ratios. Thus, given the correlations above, migrating to the most cost-efficient VM results in a low revocation risk. For example, in the experiment from Figure 6, the most cost-efficient VM was *never revoked* over the two month period.

Of course, the high TTRs in Figure 3 from §2.1 show that revocation risk is not a serious concern in EC2. However, our insight above implies that i) hopping VMs to reduce price risk and lower cost will not increase revocation risk, and ii) if spot prices were to become more volatile, VM hopping could reduce revocation risk. Table 1 illustrates the latter point. Here, we emulate a more volatile market by taking the 10 most volatile spot VMs in the `us-east-1` region and assume that we can only migrate among them. The table shows the average revocation rate for each of these volatile VMs from 2017-03 to 2017-04, as well as the average revocation rate that results from migrating to the spot VM with the lowest spot-to-on-demand ratio (and the lowest revocation risk) as spot prices fluctuate. As the table shows, hopping to the spot VM with the lowest instantaneous revocation risk results in a revocation rate nearly 0.5× that of any single VM. Thus, VM hopping is a useful mechanism for managing revocation risk if it ever increases.

2.2.2 Performance. As with revocation risk, HotSpot's migration policy does not consider performance when determining when and where to migrate. Since HotSpot migrates based on cost-efficiency,

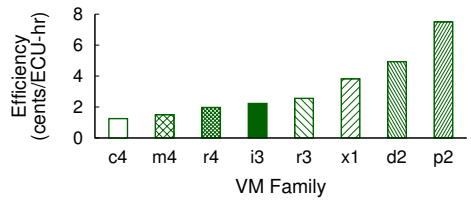

Figure 8: The cost-efficiency of on-demand VMs in the `us-east-1` region for different types of VMs in EC2.

which is a function of resource utilization, it favors VMs that do not waste resources. Given this, HotSpot may select a host VM that is under-provisioned and degrades an application's performance. During our initial analysis of spot price data in `us-east-1` (from 2016-08 to 2016-09), we observed that there was a volume discount: higher-capacity VMs were cheaper on average than lower-capacity ones. As a result, the most cost-efficient VM on average aligned with a high-capacity VM that prevented performance throttling. However, our more recent analysis in Figure 9 across multiple regions shows that this volume discount no longer applies. This change reflects how markets conditions can alter HotSpot's performance.

The figure shows the average normalized price per unit of resource for spot VMs in the `m4` family from 2017-03 to 2017-04, where the error bars represent the maximum and minimum price across each region's AZs. The graph indicates there is no consistent relationship between VM capacity and normalized cost: neither high-capacity nor low-capacity VMs are consistently cheaper per unit of resource. As a result, migrating solely based on cost-efficiency should not favor either low-capacity VMs, which degrade an application's performance, or high-capacity VMs, which may improve its performance. However, as we discuss in §3, HotSpot's migration policy takes into account multiple other factors when making migration decisions that favor higher-capacity VMs, assuming equal cost-efficiency. Thus, while HotSpot optimizes for cost-efficiency, in practice, it can also improve application performance.

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

3 HOTSPOT DESIGN

HotSpot containers automatically self-migrate to new host VMs to optimize cost-efficiency as spot prices and application resource usage change. HotSpot migrations leverage existing systems-level migration mechanisms now offered by resource containers [10]. Since HotSpot's migration functions and policies are entirely embedded into its host VM, it requires no external infrastructure. A key goal of HotSpot is to lower the barrier to entry to leveraging cloud spot markets. Despite their high potential for savings, prior estimates suggest only 3-5% of VMs allocated by EC2 come from the spot market [21], likely due to the complexity of tracking spot prices across hundreds of VMs, selecting a specific VM, determining a bid, etc. Despite this complexity, prior work on optimizing for spot VMs is neither transparent nor general: it requires configuring

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

Table 1: Migrating to the spot VM with the lowest spot-to-on-demand ratio has >0.5× revocations than other servers.

complex external infrastructure [34, 40] or making application-specific modifications [20, 22, 31, 33, 47]. Thus, prior work is not accessible to many users, especially those with moderate workloads that require few VMs. In contrast, HotSpot enables users requiring even a single VM to leverage low spot prices by running within a HotSpot-enabled VM image.

HotSpot's design includes a *controller* daemon that runs on each host VM, which executes HotSpot's monitoring, migration, and policy functions. By default, HotSpot runs only a single container per host VM, which has access to all of the VM's resources. Figure 2 from §1 depicts the role of HotSpot's controller in its control loop. To migrate, the source controller first requests and spawns a new VM via cloud APIs and then migrates the container to it (via a stop-and-copy migration in our prototype) before ceding control of the container to the controller daemon running on the new host VM. Finally, upon gaining control, the new controller terminates the source VM via cloud APIs. We assume containers are long-lived and thus may execute multiple jobs. For example, HotSpot containers may serve in a batch pool that continually assigns them tasks.

While HotSpot is compatible with any systems-level virtualization technology that supports transparent migration, we use Linux Containers (LXC) for multiple reasons. First, containers permit transparent VM-to-VM migration without requiring explicit support from the cloud platform or VM hardware. EC2 does not permit access to native VM migration that enables applications to migrate VMs. While nested VMs [13, 43] also enable VM-to-VM migration, they have a higher performance overhead than containers, especially for I/O-intensive tasks [17]. In addition, containers allow migrations to and from any VM, and not just those running on physical servers with hardware virtualization extensions (as with current nested VMs [43]). Finally, container migration is more efficient than VM migration, as the size of a container's memory state scales dynamically with its applications' memory footprint. In contrast, nested VM migration must transfer the entire nested VM memory state, regardless of its applications' memory footprint.

Containers' primary drawback is that their migration functions are less sophisticated than VMs'. For example, LXC does not yet support standard optimizations that minimize downtime and network traffic [18, 45, 46]. However, given containers' rising popularity, we expect them to adopt these optimizations in the near future.

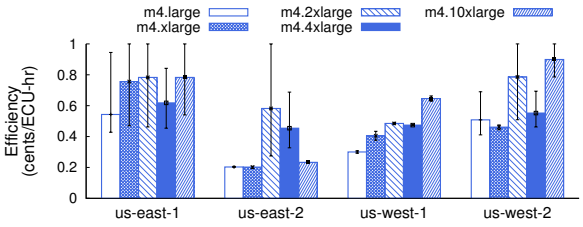

Figure 9: The average normalized cost per ECU for the `m4` family of spot VMs across multiple regions. The error bars represent the maximum and minimum cost across all AZs.

3.1 Migration Policy

HotSpot's migration policy determines when and where to migrate its resource container based on current spot prices across all VMs within its AZ and its current resource usage. Since migrations across AZs and regions require significant additional overhead to migrate disk state, we limit HotSpot to migrating within an AZ, where it can migrate disk state by re-mounting a remote disk volume, e.g., via EC2's Elastic Block Store (EBS), with little overhead. We discuss policies for "global" migration across AZs and regions in [35]. As mentioned in §2, HotSpot's migration policy optimizes for cost-efficiency, or the cost per unit of resources an application utilizes per unit time. However, quantifying a VM's utilization is complex, since it includes many resources, e.g., computation, memory, I/O, etc. Our policy quantifies cost-efficiency based on processor utilization, and only uses an application's memory footprint to eliminate candidate VMs. Specifically, since memory significantly degrades performance when constrained, our migration policy includes a rule to never migrate to a VM with less memory than the container's memory footprint. Since platforms often price network and disk I/O capacity separately from VMs, our policy does not consider them, although we could apply a similar elimination rule as for memory.

At each time t, HotSpot's migration policy computes the expected cost-efficiency of every type of spot and on-demand VM within an AZ in units of $/ECU-utilized/hour. To do so, HotSpot estimates the expected utilization on every potential VM i based on the utilization of its current VM. HotSpot makes this estimate by considering two separate cases based on the current utilization.
Low Utilization. If the ECU utilization on the current VM c is below an upper threshold (near 100%), we approximate ECU utilization u_i on VM i by proportionately scaling the ECU utilization of c across the number of ECUs offered by i. In this case, since c is not fully utilized, new VMs that have more ECUs than c should be even less utilized, while new VMs that have less ECUs than c should have a proportionate increase in utilization up to 100%.
High Utilization. Alternatively, if the current VM c's utilization is above the upper threshold, new VMs that have fewer ECUs than c should also have a utilization near 100%. However, if new VMs have more ECUs than c, we do not know how many additional ECUs the application is capable of consuming. In this case, our policy makes the aggressive assumption that the application can saturate any number of ECUs. This assumption encourages HotSpot to try out higher-capacity VMs, which can improve application performance.

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

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

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

$$e_i(t) = \frac{p_i(t)}{ECU_i \times u_i(t)} \quad (1)$$

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

3.2 Cost-Benefit Analysis

Our analysis in §2.1 assumes an ideal migration policy that is able to migrate with no overhead to the VM with the highest instantaneous cost-efficiency at each time t. Of course, in practice, migrations can incur a substantial overhead. This overhead comes in two different forms: a *performance overhead* that stems from the downtime (or performance degradation) caused by a migration, and a *cost overhead* that stems from paying for two VMs during the migration and vacating a VM early (before the end of its billing period). The combined cost of these overheads represents a migration's *transaction cost*. HotSpot's migration policy only hops VMs if it expects the savings to outweigh the transaction cost.

3.2.1 Transaction Cost. From above, the two things HotSpot requires to estimate the transaction cost are i) the time remaining T_r in the current billing interval and ii) the time required T_m to migrate the container. HotSpot tracks T_r, which is computed from the running time on the current VM and the billing interval, which is well-known. The migration time T_m is a function of the container's memory footprint and network bandwidth. As we discuss in §4, HotSpot uses a memory-to-memory stop-and-copy migration that copies the container's memory state from the memory of the source VM to the memory of the destination VM without saving it to stable storage. Thus, we estimate T_m based on the container's memory footprint M at time t of the migration divided by the available bandwidth B, plus a constant time overhead O to interact with EC2's APIs to configure the new VM, so $T_m(t) = M(t)/B + O$. We evaluate the migration time and the constant time overheads in §4.

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

$$C_i(t) = p_i(t) \times T_m(t) + p_c(t) \times max(T_r, T_m) \quad (2)$$

Here, $p_i(t)$ and $p_c(t)$ are the current spot price of the new host VM i and current host VM c, respectively. This equation represents the cost of the source and destination VM over the migration time, since neither is doing useful work, plus the cost of the remaining unused time HotSpot must pay in the source VM's billing interval. Note that the source VM's spot price $p_c(t)$ is fixed, as EC2 charges for the spot price at the beginning of each billing interval. Since migration times are short, as shown in §4, we also assume the destination's spot price is fixed during the migration.

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

$$S_i(t) = (e_c(t) - e_i(t)) \times (u_i(t) \times ECU_i) \times T_i \quad (3)$$

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

The key unknown variable in Equation 3 is T_i, or the time HotSpot expects to spend on a new host VM i. HotSpot must have an accurate estimate of T_i to determine whether the expected savings exceed the transaction costs. However, T_i is challenging to estimate, since it depends on the future value of numerous other variables, including the remaining lifetime of the container, the application's future resource usage, and the relative spot prices of every VM. A significant phase change in any of these variables can decrease the time T_i that HotSpot spends on a new VM, reducing the expected savings and altering the cost-benefit analysis.

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

Finally, HotSpot may migrate to a more cost-efficient VM, incurring transaction costs, only to find an even more cost-efficient VM becomes available before it has recouped the previous transaction costs. Thus, our policy also reduces the expected savings $S_i(t)$ in Equation 3 from hopping to a new host VM i to account for any "unpaid" transaction costs not recouped from previous migrations. As a result, HotSpot only hops to a new VM if its price is low enough to yield a positive net benefit even after paying off accumulated transaction costs from previous migrations. This choice is conservative in rate-limiting migrations and preventing accumulating large transaction costs that increase cost. HotSpot also enables users to specify a minimum time between migrations to rate-limit them.

3.2.3 Bidding. When requesting a new spot VM, HotSpot must place a bid. Since EC2's current spot market requires applications to pay the spot price and *not* their bid price, HotSpot does not require a sophisticated bidding strategy, as it automatically migrates to a new VM if the spot price rises. In contrast, the bidding policy is more important in prior work [22, 40, 53], which commits to spot VMs until they are revoked, as the greater the bid the lower the probability of revocation and the higher the potential cost. Thus, HotSpot adopts a simple bidding strategy: it always bids the maximum price, which is 10× the on-demand price in EC2. Since HotSpot also includes on-demand VMs in its cost-benefit analysis, it will never pay near its bid price, since it will always migrate to an on-demand VM if the spot price of all spot VMs ever rises above their corresponding on-demand price. Note that the specific bidding policy is orthogonal to HotSpot's design. For example, if EC2 were to change the spot market rules such that applications paid their bid price, instead of the spot price, HotSpot could support a different bidding policy without altering any of its other functions.

3.3 Qualitative Discussion

In reducing price risk, HotSpot addresses problems with current fault-tolerance-based approaches that manage revocation risk. In particular, the primary problem with fault-tolerance-based approaches is that applications configure their fault-tolerance mechanism based on the historical TTR from traces of past spot prices. However, as discussed in §2.1, spot prices often experience phases of stability and volatility. Thus, if applications experience a phase change in the prices, they may incorrectly configure their fault-tolerance mechanism, e.g., by checkpointing too much or too little, causing them to "pay" non-optimal premiums. The benefits of employing fault-tolerance are also probabilistic, and must be amortized across long time periods or a large number of applications. As a result, any individual application may end up paying high premiums without ever receiving a payout, e.g., if a revocation never occurs. In comparison, HotSpot has the following advantages.

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

HotSpot's design has some limitations. In particular, to remain application-agnostic, our migration policy makes a number of simplifying assumptions in §3.1 that do not apply to all applications.

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

Table 2: Migration latencies for EC2 API operations.

Designing migration policies for specific applications that limit these assumptions is future work. In addition, to simplify our design, HotSpot VMs are self-contained and do not coordinate their migration decisions with other HotSpot VMs. As a result, HotSpot's local migration decisions may not be globally optimal for distributed applications with complex dependencies. Applying HotSpot to distributed applications by coordinating their migration is future work.

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

4 IMPLEMENTATION

We implement HotSpot's controller daemon in Python, including the migration policy from §3 and the monitoring and migration functions described below. Specifically, HotSpot uses EC2's Python binding Boto3, and Linux Container (LXC) 2.0.7's Python API.

4.1 Spot Price and Resource Monitoring

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

4.2 Host Migration and Handoff

When triggered by the migration policy, HotSpot's controller must request and migrate to a new EC2 host. The controller performs a sequenced handoff to complete a migration to a new host VM by first requesting the VM via EC2's REST API, waiting until it is running, and then transmitting the container's memory state to it. The source controller must also perform a hand-off to a new

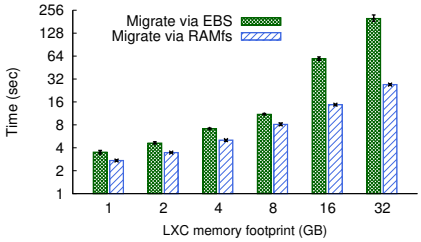

Figure 10: The time to transfer a container's memory state and restore it as a function of its memory footprint. The graph shows the average from four trials with error bars representing the minimum and maximum transfer time.

controller running on the destination VM. This hand-off requires transferring the metadata necessary to request and configure new host VMs via EC2's REST API, including the credentials necessary to access the API, such as the Secret Key. The source controller must also transfer container configuration meta-data, including the IP address and name of the root EBS volume, which the destination VM must configure via EC2's REST API before re-activating the container. Once the source controller transfers this information, the destination controller terminates the source VM.

We ran a series of microbenchmarks to quantify the overhead of the REST API operations associated with migration, as listed in Table 2. Since there is some variance in the latency, we report the mean, maximum, and minimum latency over 25 experiments. The table shows that price and resource monitoring overhead (from monitoring 402 spot prices) is negligible, even at per-second resolution. While the latency to acquire an on-demand or spot VM is between 15s and 167s, these operations do not result in application downtime, as the source VM continues to run during this period. Likewise, the 30-46s required to terminate the source VM also does not incur downtime. Application downtime is a function of the time to i) disconnect and reconnect the container's disk and network interfaces, and ii) physically transfer the container's memory state.

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

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

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

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

5 EXPERIMENTAL EVALUATION

We evaluate HotSpot at small scales using a prototype on EC2, and at large scales over a long period in simulation using a production Google workload trace [30, 42] and publicly-available EC2 spot price traces. Our simulator, also implemented in Python, executes the same migration policy as our prototype, but replaces its real-time monitoring with functions that read spot price and resource usage data from traces. The traces include each application's processor utilization and memory footprint over time. Instead of migrations, the simulator inserts a downtime derived from our microbenchmarks based on an application's current memory footprint.

We compare HotSpot with three other approaches, which select the single optimal i) on-demand VM, ii) spot VM, and iii) spot VM plus optimal fault-tolerance mechanism to run the application. The first case represents current practice; the second case is akin to EC2's SpotFleet tool, which automates bidding for spot instances (at the on-demand price by default), and the third case uses SpotOn [40], which is a representative fault-tolerance-based approach. Note that SpotOn only switches VMs on a revocation. Our evaluation then compares three metrics—cost, performance, and revocation risk—for each approach.

5.1 Prototype Results

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

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

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

We construct synthetic price traces to exhibit the correlation between price level and revocation risk in the real market, where a low-cost server is less likely to be revoked. While our experiments isolate some key factors in HotSpot's design, they do not isolate all of them. In particular, we set the billing interval to one minute, rather than one hour, so the experiments do not include the increase in cost from vacating a VM early. We use a short billing interval to enable us to isolate other factors when running half-hour jobs. Thus, in these experiments, HotSpot's transaction cost derives solely from its migration overhead. Note that this overhead is higher, as a fraction of a job's running time, the shorter the job. The migration overhead ranges from $25 - 56$s, or 0.7-1.6% of the running time for our baseline half-hour jobs. We evaluate real-world performance over longer periods with an hour-long billing interval in §5.2.

5.1.1 Baseline Experiment. Figure 11 compares the cost, running time, and revocation risk of using the on-demand, SpotFleet, SpotOn, and HotSpot approaches. Note that the y-axis of the first two graphs is normalized relative to the metric's value using on-demand VMs. For each approach, we execute three trials with a different set of randomly chosen offsets for each spot VM in the synthetic price function, where the error bars reflect the maximum and minimum values. As expected, the cost (left) decreases between 30-75% when we switch from using on-demand VMs to any approach that uses spot VMs, since spot VMs are cheaper on average than on-demand VMs. Thus, even when the SpotFleet approach, which does not use fault-tolerance or migration, experiences a revocation and has to restart the job from the beginning, it remains cheaper than using an on-demand VM. The cost further decreases for SpotOn (by 34%) and HotSpot (by 65%) compared to SpotFleet, since SpotOn benefits from periodic checkpoints that limit the work lost after a revocation, while HotSpot does not experience a revocation. Finally, we see that HotSpot's cost is 45% less than SpotOn, since HotSpot always migrates to the lowest cost VM, while SpotOn remains on each VM until it is revoked and thus experiences high price periods.

Figure 11(middle) shows that the job's running time increases relative to using an on-demand VM. This occurs because our experiment, in contrast to EC2's actual spot market, does not include VMs with multiple capacities at different normalized prices. As a

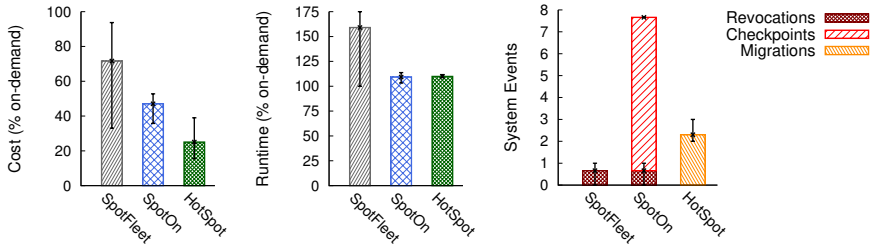

Figure 11: Comparison of cost (left), run time (middle), and revocation-related events (right) when using on-demand VMs, spot VMs without fault-tolerance (SpotFleet), spot VMs with checkpointing (SpotOn), and HotSpot when running our baseline job on our HotSpot prototype. The error bars represent the maximum and minimum of each metric across three trials.

result, there is no opportunity to improve on the performance of an on-demand VM by selecting a higher-capacity spot VM. In the figure, the average running time of SpotFleet is worse than both SpotOn and HotSpot. For SpotFleet, the decrease in running time derives from the large overhead of having to restart the job from the beginning after each revocation, while, for SpotOn and HotSpot, the decrease derives primarily from the smaller overhead of checkpointing and migration, respectively. As shown in Figure 11(right), while HotSpot migrates an average of ~2 times, its performance overhead does not exceed either SpotFleet or SpotOn. Unlike SpotFleet, HotSpot experiences no revocations and never has to recompute lost work. The figure also shows that HotSpot executes fewer migrations than SpotOn executes checkpoints. However, each checkpoint only has an overhead of 8s, while the migration overhead of the 8GB memory footprint ranges from 25-56s. These overheads balance out such that SpotOn and HotSpot maintain a similar performance.

5.1.2 Changing Memory Footprint. We next evaluate how changes in an application's memory footprint, which dictate the transaction cost of migration, affect cost and running time relative to our baseline 8GB memory footprint. Figure 12 shows the results where the error bars indicate the maximum and minimum value across three trials. Using the same configuration as our baseline experiment, we vary the memory footprint from 8GB to 64GB. As before, the cost (left) of using an on-demand VM is high relative to the other approaches in nearly all cases. In this experiment, note that the cost and performance of using an on-demand VM is the same for any size memory footprint. Only SpotOn with a 64GB memory footprint costs slightly more than using an on-demand VM due to its high periodic checkpointing overhead. SpotOn's cost decreases relative to using on-demand VMs as the memory footprint and the resulting checkpointing overhead decrease. SpotFleet maintains the same cost across all memory footprints, since it does not perform any checkpoints or migrations. HotSpot's cost is consistently lower than both SpotFleet and SpotOn, since it always migrates to the lowest-cost server. As the memory footprint increases, the overhead of these migrations also increase, which reduces the cost savings relative to using on-demand VMs. However, even for a 64GB memory footprint, HotSpot's cost is 40% less than the on-demand VM.

Figure 12(middle) shows that HotSpot's running time is also consistently equal to or less than the running time of SpotFleet and SpotOn as the memory footprint varies. SpotFleet's running time remains constant since it does not depend on the memory footprint, and is greater than HotSpot's running time even for a 64GB memory footprint. SpotOn's running time is nearly equal to HotSpot's running time for the 8GB memory footprint, as discussed in §5.1.1. However, SpotOn's running time increases more rapidly as the memory footprint increase compared to HotSpot's running time. For a 64GB memory footprint, HotSpot's running time is nearly 40% lower than SpotOn's running time.

The differences in running time stem from the overheads of checkpointing, migrating, and recomputing lost work after each revocation. Figure 12(right) plots the number of revocations, checkpoints, and migrations, where the order of the bars for SpotFleet, SpotOn, and HotSpot are the same as the adjacent figures. We do not include the on-demand approach, since it does not experience any revocation-related events. The graph shows that HotSpot never experiences a revocation, since it always migrates to the lowest-cost server with a low revocation risk. In contrast, SpotFleet and SpotOn experience at least one revocation on average. For low memory footprints, SpotOn checkpoints frequently, since the overhead of checkpointing is low. However, for large memory footprints, the checkpointing overhead is high so SpotOn only checkpoints once. Thus, the revocations for SpotOn at large memory footprints incur a large recomputation overhead that increases its running time.

5.1.3 Changing Spot Price Volatility. We also vary the frequency of revocations relative to our baseline to illustrate the impact of market volatility on cost and running time. In this case, we vary the revocation rate by changing the periodicity of our sinusoidal price function. Figure 13 plots the resulting revocation rate (in revocations per hour) on the x-axis. Again, all spot-based approaches in Figure 13(left) cost less on average than using an on-demand VM. The error bars represent the maximum and minimum cost across three trials. We also see that HotSpot has a lower cost than the other approaches across all revocation rates. As the spot price becomes more volatile, HotSpot's cost advantage improves relative to both SpotFleet and SpotOn due to the overhead they experience from both checkpointing and recomputing lost work after a revocation.

Figure 13(middle) plots the job running time as the revocation rate changes. We see that SpotFleet's performance is highly sensitive to the revocation rate, since each revocation incurs a large recomputation overhead. Note that once the revocation rate increases to two per hour SpotFleet never finishes, so we label ∞ for its cost, running time, and system events. By comparison, both SpotOn and HotSpot have similar running times across all revocation rates. While SpotOn experiences some revocations at higher revocation rates, as shown in Figure 13(right), the performance impact of these revocations is limited by its periodic checkpoints.

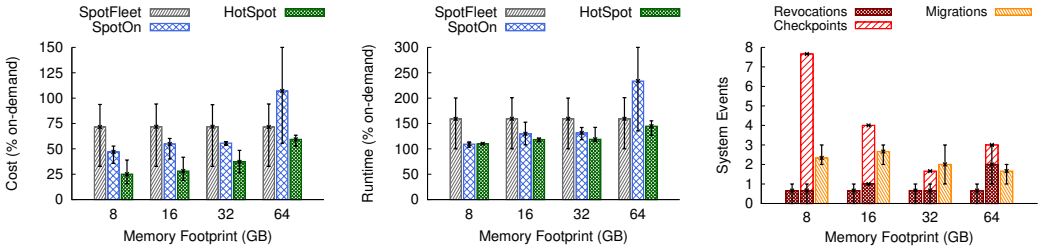

Figure 12: Comparison of cost (left), run time (middle), and system events (right) when using on-demand VMs, spot VMs without fault-tolerance (SpotFleet), spot VMs with checkpointing (SpotOn), and HotSpot as the memory footprint varies. The error bars represent the maximum and minimum of each metric across three trials.

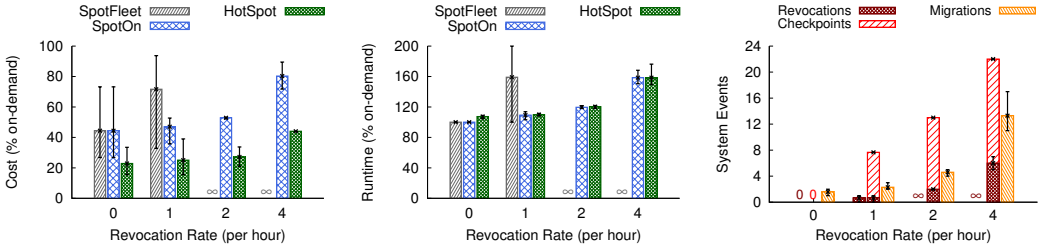

Figure 13: Comparison of cost (left), run time (middle), and system events (right) when using on-demand VMs, spot VMs without fault-tolerance (SpotFleet), spot VMs with checkpointing (SpotOn), and HotSpot as the spot price volatility changes. The error bars represent the maximum and minimum of each metric across three trials. Once the revocation rate increases to two per hour SpotFleet never finishes, so we label ∞ for its cost, running time, and system events.

As before, the order of the bars for SpotFleet, SpotOn, and HotSpot in Figure 13(right) are the same as the adjacent figures.

Since the checkpointing overhead is low for our 8GB baseline memory footprint, SpotOn is able to checkpoint many times over the length of the job, e.g., one checkpoint every 6 minutes in the baseline case of one revocation per hour. This frequent checkpointing limits the performance impact of revocations. In comparison, the lowest-cost VM changes more frequently as price volatility increases, which requires HotSpot to migrate more frequently. While each migration incurs more overhead than each checkpoint, HotSpot experiences no revocations and thus incurs no recomputation overhead. As in Figure 11's baseline, the checkpointing and migration overheads of SpotOn and HotSpot, respectively, balance out such that their performance is similar across all volatilities.

5.2 Simulation Results

Our simulator uses job traces from a production Google cluster [30, 42], and the same EC2 spot price traces described in §2. The Google cluster traces report each job's memory footprint and normalized CPU utilization every five minutes. Since the Google cluster trace normalizes the server capacity between zero and one, it does not contain the actual CPU capacity of the servers. As a result, we re-normalize the server capacity to between 6.5 and 195 ECUs, which represent the minimum and maximum number of ECUs offered by EC2 across its VM types. For the on-demand approach, we select the VM type that has the closest number of ECUs to the ECUs in the trace. For the spot-based approaches, we initially select the lowest cost VM that matches each job's average ECU utilization. For SpotFleet, we also use this policy to select a new VM on each revocation. SpotOn and HotSpot use their own respective policies for selecting VMs after a revocation. We also assume each job's performance is a linear function of its resource utilization, i.e., ECUs utilized. That is, if a job is at 100% utilization of its normalized server capacity, and then migrates to a server with half the number of ECUs, we assume a 2× slowdown. If a job's utilization in the trace is 100% we do not know what its utilization would be on a higher capacity server. In this case, we make the pessimistic assumption that the job cannot scale up to use more ECUs than it did in the original trace. Our experiments assume EC2's standard one-hour billing interval, and sets the expected time to spend on a new server T_i equal to the MTTC of 1.1 hours based on our analysis in §2.1.

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

Figure 14(left) shows the average cost of each approach, where the error bars reflect the maximum and minimum values across five trials. As the graph shows, all of the spot-based approaches have a significantly lower cost than using on-demand VMs. In addition, HotSpot is able to further lower the average cost compared to both SpotFleet and SpotOn. In addition, Figure 14(middle) shows that all the spot-based approaches slightly increase the average running time relative to using the on-demand VM specified in the job trace. In particular, HotSpot increases the running time compared to using on-demand VMs by <0.5% due its migration overhead, which is

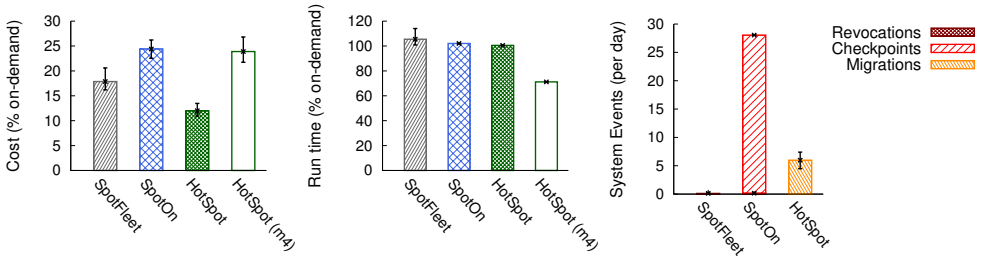

Figure 14: Comparison of cost (left), run time (middle), and system events (right) when using on-demand VMs, spot VMs without fault-tolerance (SpotFleet), spot VMs with checkpointing (SpotOn), and HotSpot from simulating jobs from a production trace on spot VMs based on EC2 spot price traces. The error bars represent the maximum and minimum over five trials.

less than the increase caused by both SpotOn and SpotFleet. This increase in running time occurs because jobs in the original trace tend to run on over-provisioned servers, such that their average utilization is low. Thus, there is never an opportunity to improve performance by migrating to even higher capacity servers.

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

Finally, Figure 14(right) shows the revocation, checkpointing, and migration rate per day of each approach. HotSpot migrates on average ~6 times per day, while SpotOn executes ~28 checkpoints per day. While difficult to see in the graph, all approaches also experience revocations. Specifically, SpotFleet, SpotOn, and HotSpot have average revocation rates of 0.118, 0.152, and 0.02 revocations per day, respectively. Thus, HotSpot migration policy reduces the revocation rate compared to the other spot-based approaches.

6 RELATED WORK

Many researchers have recognized the opportunity to reduce cost by leveraging spot VMs, however, there is little prior work similar to HotSpot that supports proactive migration as market conditions change. Instead, the focus of prior work has been on selecting the "optimal" spot VM based on an application's expected resource usage and future spot prices [20, 22, 31, 33, 34, 40, 47, 53]. Much of the prior work focuses on reducing revocation risk by configuring fault-tolerance mechanisms, such as checkpointing and replication [20, 22, 31, 33, 34, 40]. In §5, we compare with SpotOn [40], which automatically selects and configures the optimal spot VM and fault-tolerance mechanism to execute a job. Prior work applies similar fault-tolerance-based approaches to specific distributed applications including Hadoop [53], Spark [31, 33], parameter servers [20], and matrix multiplication [22]. In contrast, HotSpot is transparent to the application and operates at the level of a single server (not a distributed system). Unlike HotSpot, prior work does not dynamically migrate to new VMs as spot prices change, but instead selects new VMs after a revocation [31, 33, 40] or at periodic intervals [20].

Since prior work often implicitly commits to running on a particular spot VM until a revocation occurs, the bidding strategy is important in balancing high costs due to an increase in the spot price (when bidding too high) and the performance penalty from increased revocations (when bidding too low). Thus, there is a significant body of work on spot VM bidding strategies [25–27, 39, 41, 44, 50–53]. In contrast, the bidding strategy is not as important to HotSpot, as it proactively migrates as spot prices change. HotSpot never commits to a spot VM, and often migrates to a new VM before prices spike and cause revocations. Prior work also notes that EC2's spot market is artificial, since Amazon both operates the market and is the sole provider. For example, Ben-Yehuda et al. showed that before 2011, EC2 spot prices were not consistent with a constant minimal price auction [14, 15]. However, HotSpot's cost benefits do not rely on spot prices being driven by supply and demand, but only that there is a price difference between VMs.

Finally, similar to HotSpot's container migrations, Smart Spot instances migrate nested VMs between spot VMs in EC2 [23]. However, Smart Spot instances use a centralized scheduler that monitors a group of nested VMs and determines an optimal packing of them on spot VMs to reduce cost. Smart Spot Instances also do not consider revocation risk in their placement decisions, and suggest applications use fault-tolerance mechanisms, such as replication or checkpointing to mitigate this risk, which violates transparency.

7 CONCLUSION

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

Acknowledgements. This work is supported by NSF grant #1422245 and a Google Faculty Research award. We would like to thank anonymous reviewers for their feedback. We would especially like to thank our shepherd John Wilkes for providing multiple rounds of detailed comments that improved the quality of the paper.

REFERENCES

[1] 2017. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/. (Accessed August 2017).

[2] 2017. Amazon Spot Instances. https://aws.amazon.com/ec2/spot/. (Accessed August 2017).

[3] 2017. AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/. (Accessed August 2017).

[4] 2017. cmpute.io, Inc. http://www.cmpute.io. (Accessed August 2017).

[5] 2017. Google Compute Engine. https://cloud.google.com/compute/pricing. (Accessed August 2017).

[6] 2017. Linux Containers. http://linuxcontainers.org. (Accessed August 2017).

[7] 2017. Spot Instance Limits. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-limits.html. (Accessed August 2017).

[8] 2017. Spot Instance Product Details. https://aws.amazon.com/ec2/spot/details/. (Accessed August 2017).

[9] C. Babcock. 2015. Amazon's 'Virtual CPU'? You Figure it Out, In *InformationWeek*. (December 23rd 2015).

[10] G. Banga, P. Druschel, and J. Mogul. 1999. Resource Containers: A New Facility for Resource Management in Server Systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[11] J. Barr. 2015. New - EC2 Spot Instance Termination Notices. AWS Blog, https://aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/. (January 6th 2015).

[12] J. Barr. 2015. New Spot Fleet Option - Distribute Your Fleet Across Multiple Capacity Pools. AWS Blog, https://aws.amazon.com/blogs/aws/new-spot-fleet-option-distribute-your-fleet-across-multiple-capacity-pools/. (September 15th 2015).

[13] M. Ben-Yehuda, M. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[14] O. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. 2011. Deconstructing Amazon EC2 Spot Instance Pricing. In *International Conference on Cloud Computing Technology and Science (CloudCom)*.

[15] O. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. 2013. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Transactions on Economics and Computation (TEAC)* 1, 3 (2013).

[16] J. Boutelle. 2011. What to do when Amazon's spot prices spike, In *Gigaom*. (December 27th 2011).

[17] L. Chaufournier, P. Sharma, P. Shenoy, and Y. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *International Middleware Conference (Middleware)*.

[18] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Waeldr. 2005. Live Migration of Virtual Machines. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[19] L. Columbus. 2016. Roundup of Cloud Computing Forecasts and Market Estimates, In *Forbes*. (March 13th 2016).

[20] A. Harlap, A. Tumanov, A. Chung, G. Ganger, and P. Gibbons. 2017. Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets. In *European Conference on Computer Systems (EuroSys)*.

[21] S. Higginbotham. 2013. Bidding Strategies? Arbitrage? AWS Spot Market is where Computing and Finance Meet, In *Gigaom*. (October 8th 2013).

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

[23] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. 2016. Smart Spot Instances for the Supercloud. In *International Workshop on CrossCloud Infrastructures and Platforms (CrossCloud)*.

[24] F. Lardinois. 2016. Spotinst, which helps you buy AWS Spot Instances, raises $2M Series A, In *TechCrunch*. (March 8th 2016).

[25] Q. Liang, C. Wang, and B. Urgaonkar. 2016. Spot Characterization: What are the Right Features to Model?. In *International Workshop on System Analytics and Characterization (SAC)*.

[26] M. Mazzucco and M. Dumas. 2011. Achieving Performance and Availability Guarantees with Spot Instances. In *International Conference on High Performance Computing and Communications (HPCC)*.

[27] I. Menache, O. Shamir, and N. Jain. 2014. On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud. In *International Conference on Autonomic Computing (ICAC)*.

[28] J. Novet. 2015. Amazon pays $20M-$50M for ClusterK, the startup that can run apps on AWS at 10% of the regular price, In *VentureBeat*. (April 29th 2015).

[29] X. Ouyang, D. Irwin, and P. Shenoy. 2016. SpotLight: An Information Service for the Cloud. In *International Conference on Distributed Computing Systems (ICDCS)*.

[30] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2011. *Google cluster-usage traces: format + schema*. Technical Report. Google Inc., Mountain View, CA, USA. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[31] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. 2016. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *European Conference on Computer Systems (EuroSys)*.

[32] P. Sharma, D. Irwin, and P. Shenoy. 2016. How Not to Bid the Cloud. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[33] P. Sharma, D. Irwin, and P. Shenoy. 2017. Portfolio-driven Resource Management for Transient Cloud Servers. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[34] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. 2015. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *European Conference on Computer Systems (EuroSys)*.

[35] S. Shastri and D. Irwin. 2017. Towards Index-based Global Trading in Cloud Markets. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[36] Z. Shen, Q. Jia, E. Sela, B. Rainero, W. Song, R. van Renesse, and H. Weatherspoon. 2016. Follow the Sun through the Clouds: Application Migration for Geographically Shifting Workloads. In *Symposium on Cloud Computing (SoCC)*.

[37] R. Singh, D. Irwin, P. Shenoy, and K.K. Ramakrishnan. 2013. Yank: Enabling Green Data Centers to Pull the Plug. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[38] R. Singh, P. Sharma, D. Irwin, P. Shenoy, and K.K. Ramakrishnan. 2014. Here Today, Gone Tomorrow: Exploiting Transient Servers in Datacenters. *IEEE Internet Computing* 18, 4 (April 2014).

[39] Y. Song, M. Zafer, and K. Lee. 2012. Optimal Bidding in Spot Instance Market. In *Infocom*.

[40] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. 2015. SpotOn: A Batch Computing Service for the Spot Market. In *Symposium on Cloud Computing (SoCC)*.

[41] S. Tang, J. Yuan, and X. Li. 2012. Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In *International Conference on Cloud Computing (CLOUD)*.

[42] John Wilkes. 2011. More Google cluster data. Google research blog. (Nov. 2011). Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[43] D. Williams, H. Jamjoom, and H. Weatherspoon. 2012. The Xen-Blanket: Virtualize Once, Run Everywhere. In *European Conference on Computer Systems (EuroSys)*.

[44] R. Wolski and J. Brevik. 2016. Providing Statistical Reliability Guarantees in the AWS Spot Tier. In *High Performance Computing Symposium (HPC)*.

[45] T. Wood, A. Lagar-Cavilla, K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. 2011. PipeCloud: Using Causality to Overcome Speed-of-Light Delays in Cloud-Based Disaster Recovery. In *Symposium on Cloud Computing (SoCC)*.

[46] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. 2011. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *International Conference on Virtual Execution Environments (VEE)*.

[47] Z. Xu, C. Stewart, N. Deng, and X. Wang. 2016. Blending On-Demand and Spot Instances to Lower Costs for In-Memory Storage. In *International Conference on Computer Communications (Infocom)*.

[48] Y. Yan, Y. Gao, Z. Guo, B. Chen, and T. Moscibroda. 2016. TR-Spark: Transient Computing for Big Data Analytics. In *Symposium on Cloud Computing (SoCC)*.

[49] Y. Yang, G. Kim, W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B. Chun. 2017. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *European Conference on Computer Systems (EuroSys)*.

[50] M. Zafer, Y. Song, and K. Lee. 2012. Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs. In *International Conference on Cloud Computing (CLOUD)*.

[51] S. Zaman and D. Grosu. 2011. Efficient Bidding for Virtual Machine Instances in Clouds. In *International Conference on Cloud Computing (CLOUD)*.

[52] Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao. 2011. Dynamic Resource Allocation for Spot Markets in Clouds. In *Workshop on Hot Topics in Management of Internet, CLoud, and Enterprise Networks and Services (HotICE)*.

[53] L. Zheng, C. Joe-Wong, C. Tan, M. Chiang, and X. Wang. 2015. How to Bid the Cloud. In *ACM SIGCOMM Conference (SIGCOMM)*.