SpotOn: A Batch Computing Service for the Spot Market

Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy
University of Massachusetts Amherst

Abstract

Cloud spot markets enable users to bid for compute resources, such that the cloud platform may revoke them if the market price rises too high. Due to their increased risk, revocable resources in the spot market are often significantly cheaper (by as much as $10\times$) than the equivalent non-revocable on-demand resources. One way to mitigate spot market risk is to use various fault-tolerance mechanisms, such as checkpointing or replication, to limit the work lost on revocation. However, the additional performance overhead and cost for a particular fault-tolerance mechanism is a complex function of both an application's resource usage and the magnitude and volatility of spot market prices.

We present the design of a batch computing service for the spot market, called SpotOn, that automatically selects a spot market and fault-tolerance mechanism to mitigate the impact of spot revocations without requiring application modification. SpotOn's goal is to execute jobs with the performance of on-demand resources, but at a cost near that of the spot market. We implement and evaluate SpotOn in simulation and using a prototype on Amazon's EC2 that packages jobs in Linux Containers. Our simulation results using a job trace from a Google cluster indicate that SpotOn lowers costs by 91.9% compared to using on-demand resources with little impact on performance.

Categories and Subject Descriptors D.4.7 [*Organization and Design*]: Batch Processing Systems

General Terms Performance, Reliability, Measurement

Keywords Spot Market, Fault-tolerance, Batch job

1. Introduction

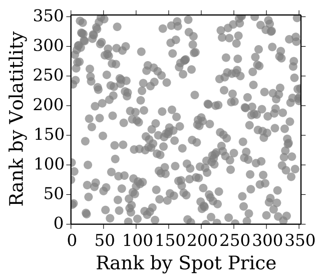
Infrastructure-as-a-Service (IaaS) cloud platforms are becoming increasingly sophisticated, and now offer a wide variety of resources under contract terms that expose a tradeoff between cost and availability. For example, Amazon's Elastic Compute Cloud (EC2) offers contracts for *spot* "instances," i.e., virtual machines (VMs) bound to a specified amount of computation, storage capacity, and network bandwidth. Users place a bid for spot instances (in dollars per unit time of use) and receive them only when the *spot price*, which changes in real time, falls below their bid price. As long as the spot price is below their bid price, users have access to the resources and pay the spot price for them. However, once the spot price exceeds the bid price, EC2 may unilaterally reclaim the resources, while giving users only a brief (two minutes in EC2 [4]) warning to vacate them.

Spot instances differ from *on-demand* instances, which users relinquish voluntarily and EC2 cannot revoke. While the price of spot instances is significantly less (often by a factor of ten or more) than the equivalent on-demand instances, the risk associated with using them is significantly higher, since EC2 may revoke them at any time if the spot price rises. While their low price makes spot instances attractive, the spot market introduces additional market and application dynamics that increase the complexity of using it.

- **Market Complexity.** EC2's global market for instance types is massive and diverse, as it operates a different spot market with a different dynamic price for each instance type in each availability zone of each region. Currently, EC2 operates more than 2000 distinct spot markets across 9 regions and 26 availability zones,[1] each with a different dynamic price per unit of computational resources. Given the spot market's size, selecting the instance type, region, and zone that yields the lowest overall cost per unit of work is highly complex.
- **Application Complexity.** The spot market also introduces new dynamics for applications, which must be able to gracefully handle the sudden revocation (and allocation) of resources as the spot price changes. Amazon recommends using spot instances for simple delay-tolerant (or optional) tasks that store their persistent state on remote disks, e.g., in Amazon's Elastic Block Store (EBS), enabling them to simply pause and resume their

SoCC '15, August 27 - 29, 2015, Kohala Coast, HI, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3651-2/15/08…$15.00.
http://dx.doi.org/10.1145/2806777.2806851

[1] Each availability zone can be thought of as a different data center within a distinct geographical region, e.g., US-East, US-West (Oregon), etc.



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

Figure 1. Scatterplot of the rank in spot prices' volatility and magnitude for 353 markets (left). Table of the top 10 most volatile markets (in revocations/day when bidding the on-demand price) and their per-hour spot and on-demand price (right).

execution whenever EC2 revokes or allocates spot instances, respectively [1]. Applications may also leverage various fault-tolerance mechanisms, such as checkpointing, to limit the work lost on each revocation. In this case, applications may continue execution by replacing revoked instances with instances from a different market.

Of course, different fault-tolerance mechanisms impose different performance penalties (and thus different costs) based on each application's resource usage characteristics. Thus, optimizing an application's performance and cost is challenging: it requires managing volatile market and application dynamics by determining i) where to execute an application based on global spot market prices and volatility and ii) what fault-tolerance mechanism to employ based on an application's resource usage. In this paper, we present the design of a batch computing service, called *SpotOn*, to specifically optimize the cost of running non-interactive batch jobs on spot instances. By focusing narrowly on batch jobs, SpotOn has the freedom to i) select from a wide set of available fault tolerance mechanisms and ii) exploit favorable spot markets across availability zones and regions.

SpotOn enables a user to select an instance type to run their job with the goal of achieving similar performance as running on an on-demand instance, but at a price near that of spot instances. To do so, SpotOn dynamically determines i) the best instance type and spot market (in a particular zone and region) to run the job (which may differ from the user's choice) and ii) the fault-tolerance mechanism that best balances the risk of revocation with the overhead of the mechanism. Our hypothesis is that by judiciously selecting the fault-tolerance mechanism and spot market, SpotOn can decrease the cost of running jobs, without significantly increasing the job's running time (and in some cases decreasing it), compared to using on-demand instances. In evaluating our hypothesis, we make the following contributions.

Fault-tolerance Modeling. We review existing systems-level fault-tolerance mechanisms, and derive simple models that capture their overhead as a function of a job's resource usage. The mechanisms fall broadly into three categories: i) reactively migrating a job prior to a revocation, ii) proactively checkpointing memory and local disk state, and iii) replicating computation across multiple instances.

Selection and Bidding Policies. Based on our models, we derive a greedy cost-aware selection policy that minimizes each job's expected cost by selecting i) the spot market and bid to run a job, ii) the fault-tolerance mechanism to use and how to use it, and iii) whether to use local versus remote I/O.

Implementation and Evaluation. We implement SpotOn in simulation and using a prototype on EC2, and evaluate the effect of our greedy cost-aware policy on performance and cost. Our prototype packages jobs in Linux Containers (LXC) to facilitate efficient checkpointing and migration. Our results on a Google cluster trace indicate that SpotOn lowers costs by 91.9% compared to using on-demand resources with little impact on performance.

2. Background and Overview

Our work assumes an IaaS platform that sells resources in a market, which sets a resource price that changes dynamically based on supply and demand. In this paper, we focus on EC2 due to the large size and diversity of its global spot market, although Google recently introduced preemptible instances, which have similar properties as spot instances [3].

2.1 Background

EC2 offers a wide range of VM (or instance) types with different resource allotments. Each instance type is available in multiple geographic locations (or *regions*), where each location includes a cluster of data centers (or *availability zones*). EC2 operates a separate spot market with a distinct dynamic spot price for each instance type in each zone, such that, if the spot price for an instance type exceeds a user's bid price for it, the platform revokes, i.e., shuts down, the instance after a brief warning, e.g., two minutes in EC2 [4]. Finally, EC2 also offers the same instance types in the on-demand market for a fixed price, such that the platform cannot revoke them. We consider these fixed-price on-demand instances as another spot market where the price is stable and there is a 0% revocation probability.

Our work focuses narrowly on designing a service to run non-interactive batch jobs. While these jobs may either be sequential or parallel, we consider individual jobs and not multi-job workflows, e.g., where multiple jobs execute as part of a sequential (or graphical) pipeline with each job passing its output as the input to one or more other jobs. We also assume jobs are idempotent, since, to mitigate the impact of resource revocations, SpotOn must be capable of rolling back to a previous checkpoint or replicating a job's computation on multiple instances.

Finally, SpotOn uses estimates of job runtime and resource usage to guide the cost-aware selection of a spot market and fault-tolerance mechanism for each job. Importantly, SpotOn's estimates need not be highly accurate, as our experiments in Section 6 demonstrate that all cost-aware policies are more cost-effective than running on on-demand machines. While there is substantial prior work on job runtime estimation and resource usage, e.g. [7, 14, 20], in practice, users often submit thousands of the same type of job, e.g., to conduct parameter space searches, with the same resource characteristics, which enables batch schedulers to profile them. SpotOn may use any available technique to estimate job runtime and resource usage. In this paper, we assume SpotOn can profile a job's resource usage *a priori* and characterize it as a simple vector specifying its running time, memory footprint, and the fraction of time waiting on I/O versus using the CPU on a reference instance type.

An important premise behind our work is that spot markets and jobs exhibit a wide range of characteristics. Figure 1 gives some indication of the diversity in price characteristics across different EC2 spot markets. The figure shows a scatterplot of the rank of 353 markets in terms of their average spot price and volatility over the past three months (left), which demonstrates that markets differ widely in their combination of volatility and price, i.e., the lowest price is not always the least volatile. The figure also lists the top 10 most volatile markets and shows that their average spot price[2] is as much as 10$\times$ less than the corresponding on-demand price. We capture volatility in revocations/day when bidding the on-demand price, since users have no incentive to pay more than the on-demand price for spot instances. Similarly, application resource usage is diverse: Figure 2 shows a scatter plot of CPU, memory, and I/O resource usage for jobs in a Google cluster trace (normalized to the 99th percentile value) [15]. As we discuss, the choice of fault-tolerance mechanism is a function of both resource usage and spot price dynamics, and is likely different for each job.

2.2 SpotOn Overview

Given the assumptions above, SpotOn offers a service that enables users to select an instance type to execute their batch job. SpotOn's goal is to complete the job in near the time it would take on an on-demand instance for a cost near that of running on a spot instance. Figure 3 depicts SpotOn's architecture, which accepts job submissions as Linux Containers (LXC). We choose to package batch jobs within containers for a number of reasons. First, containers are convenient because they encapsulate all of a job's dependencies similar to a VM. Second, containers include efficient checkpointing and migration mechanisms, which SpotOn requires; unlike with VMs, the size of a container checkpoint scales dynamically with a job's memory footprint. Third, containers enable SpotOn to partition a single large instance type into smaller instances, which makes a broader set of spot markets available to run a job. Finally, containers require only OS support, and do not depend on access to underlying hypervisor mechanisms, which are typically not exposed by cloud platforms.

[2] We exclude periods where the spot price exceeds the on-demand price.

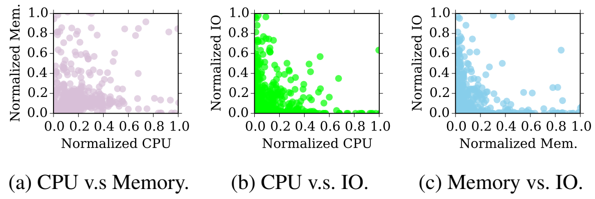


Figure 2. Scatter-plot of normalized CPU, memory, and I/O resource usage per task in a Google cluster trace [15].

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

Upon revocation, SpotOn always continues executing a job on another instance in another market. As we discuss, there is a penalty associated with revocation based on a job's resource usage and chosen fault-tolerance mechanism.

3. Fault-tolerance Mechanisms and Models

SpotOn executes jobs on spot instances when they are cheaper than the equivalent on-demand instances, and then employs fault-tolerance mechanisms to mitigate the impact of revocations. Note that SpotOn employs systems-level variants of these mechanisms, and requires no application modifications. Our fault-tolerance mechanisms fall into three broad categories: i) reactive job migration prior to a revocation, ii) checkpointing of job state to remote storage, and iii) replicating a job's computation across multiple instances. Each mechanism incurs different overheads (and costs) during normal execution and upon revocation based on a job's resource usage. Figure 4 depicts these overheads, which we capture using the simple models described below.

3.1 Reactive Migration

The simplest fault-tolerance mechanism is to migrate a job immediately upon receiving a warning of impending revocation. Since EC2 provides a brief two-minute warning, Spo-

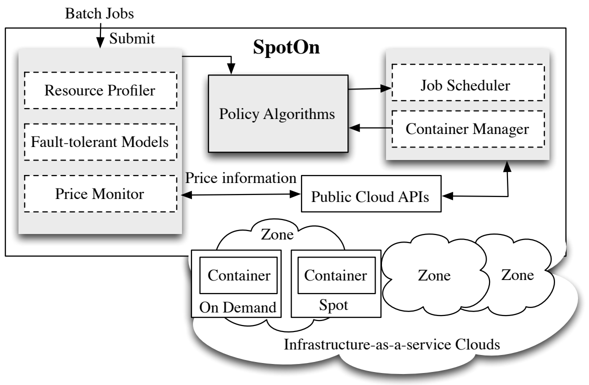


Figure 3. Depiction of SpotOn's Architecture

tOn can use this approach for jobs that are capable of checkpointing their local memory and disk state to a remote disk within two minutes. The time to checkpoint a job's state is a function of both the size of its local memory and disk state based on the network bandwidth and disk throughput between the job's VM instance and the remote disk. Of course, if a job's checkpoint does not complete within two minutes, this approach risks a failure that requires restarting a job.

While there are many migration variants, such as live precopy [5] and post-copy [8] migration, a simple stop-and-copy migration is the optimal approach for batch jobs that permit downtime during migration. To reduce downtime, live migration extends (often significantly) the total migration time. In addition, live migration requires a destination host to be available during the migration process. However, it may take over 90 seconds to acquire and boot a new on-demand instance in EC2 [10], and it takes even longer to acquire and boot a spot instance (even when the bid price exceeds the spot price). Thus, even if SpotOn immediately requests a new on-demand instance to host a job, it leaves at most 30 seconds of overlap to complete a live migration, which drastically reduces the jobs that are amenable to reactive migration (based on the size of the local state).

Given that reactive migration incurs only a modest downtime at each revocation, it is generally the best option if it is possible. However, migration has some important limitations and drawbacks. First, only jobs with small memory footprints and local disk state can leverage migration. As our experiments show, even within an availability zone, the combined memory and local disk state must be less than 4GB to ensure SpotOn is able to reliably complete a migration within the two minute revocation warning time. Thus, migration generally precludes using any local disk state (even within an availability zone), which results in increased running time (and additional cost) for I/O-intensive applications. Reactive migration is also not possible across zones/regions, since there is much less network bandwidth available between zones/regions and it requires migrating much more state, as remote disks are not available across zones/regions in EC2. As a result, reactive migration cannot exploit attractive spot markets across multiple zones/regions.

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

$$T_m = \frac{M+D}{R_b^s} + \frac{M+D}{R_b^r} \tag{1}$$

The first term captures the time to save the memory and local disk state to a remote disk, while the last term captures the time to restore it. Since the job is paused over T_m, the migration time also represents the downtime (or overhead) associated with each migration. Note that each migration incurs a cost based on T_m, since SpotOn must pay for resources during this time but the job does no useful work. Thus, the overhead (and cost) for reactive migration is a function of the magnitude of T_m and the market's volatility, i.e., the number of revocations over the job's run time.

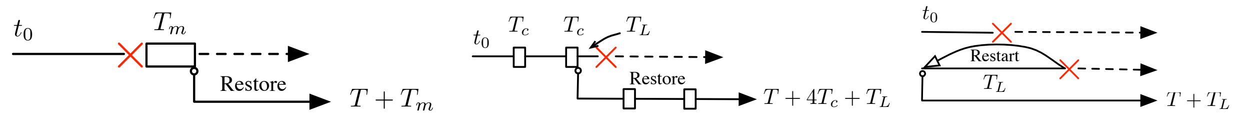
3.2 Proactive Checkpointing

Proactive checkpointing is an extension of migration that stores checkpoints at periodic intervals. The per-checkpoint latency T_c to checkpoint a job's state to remote disk is equivalent to the first term of the time to migrate as shown below. Note that, while continuous checkpointing mechanisms exist [6, 17], they incur a higher overhead than necessary for batch jobs, which permit much coarser periodic checkpoints.

$$T_c = \frac{M+D}{R_b^s} \tag{2}$$

Unlike reactive migration, proactive checkpointing is applicable to any job, not just those with small memory footprints and local disk state. With this approach, the number of checkpoints is not related to market volatility and the number of revocations, but on a specified checkpointing interval τ. Thus, the total time spent checkpointing a job with running time T is $\frac{T}{\tau} * T_c$. As before, we assume the job pauses during each checkpoint, which increases the job's running time and cost. As with reactive migration, there is also an additional cost associated with restoring a job after each revocation, such that restoring across zones/regions is often prohibitively expensive, since it requires migrating both memory state and any persistent state. Finally, there remains a tradeoff between using local versus remote storage: using local storage incurs a higher checkpointing overhead but decreases the running time of I/O-intensive jobs. In general, since checkpointing the local disk is time-consuming, jobs only use remote disks when proactively checkpointing.

Importantly, proactive checkpointing not only incurs an overhead for each checkpoint, but also requires rolling a job back to the last checkpoint on each revocation. For example, if a platform revokes a job right before a periodic checkpoint, then it loses nearly an entire interval τ of useful work. Thus, proactive checkpointing presents a tradeoff between the overhead of checkpointing and the probability of losing



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

Figure 4. Each fault-tolerance mechanism incurs a different overhead during normal execution and on revocation. Here, reactive migration incurs an overhead of T_m on each revocation, proactive checkpointing incurs an overhead of T_c for each checkpoint, and replicating computation incurs an overhead of T_L based on the work lost when both replicas are revoked.

work on revocation: the smaller the interval τ the higher the checkpointing overhead during normal execution but the lower the probability of losing work on revocation and vice versa. This overhead is a function of a job's resource usage, i.e., its memory footprint, and the spot market's volatility.

3.3 Replicating Computation

Finally, we consider replicating computation on multiple spot instances across multiple markets. When replicating computation on multiple instances, the overhead is related to the magnitude and volatility of spot prices in the market, and not the size of a job's memory and local disk state. As a result, replicating computation provides SpotOn useful flexibility along multiple dimensions relative to reactive migration and proactive checkpointing, as listed below.

- **Enables Local Storage.** Unlike with checkpointing and migration, replicating a job enables it to use local storage, since there is no need to save local disk state prior to revocation. I/O-intensive jobs may execute significantly faster when using local storage, as opposed to using remote storage with checkpointing/migration.
- **Exploits Multiple Zones/Regions.** Replicating computation enables SpotOn to exploit multiple zones/regions. With checkpointing and migration, the overhead of transferring state between zones/regions is prohibitively expensive. In contrast, SpotOn can replicate a job in two or more different zones/regions, enabling it to exploit price drops in either zone/region.
- **Supports Parallel Jobs.** Replicating jobs also more easily accommodates parallel jobs, since taking distributed checkpoints is significantly more complex than checkpointing a single node. Since LXC does not support distributed checkpoints, SpotOn is currently only able to support parallel jobs by replicating them.

SpotOn considers replicating computation in two different, but complementary, ways, as we describe below.

3.3.1 Replication across Spot Instances

SpotOn may execute multiple replicas of a job across two or more spot instances in different markets. These markets may be in different zones/regions or within the same availability zone but on different instance types. Since the price of spot instances is often much more than a factor of two less than an equivalent on-demand instance, deploying multiple spot instances is often cheaper than executing a job on an on-demand instance. Assuming the price of spot instances across markets is independent, then the probability of at least one instance completing before a revocation is much higher than the job completing on any single instance.

Specifically, based on each spot market's historical prices and for a given bid price, we can compute a revocation probability P_r that a job with running time T is revoked before it completes. Given P_r in each market, the completion probability P_c that at least one of n job replicas across different spot markets completes is one minus the probability that all of the jobs are revoked, or $P_c = 1 - \prod_{k=1}^{n} P_r^k$. Of course, the longer a job's running time, the higher the revocation probability P_r at each instance, and the lower the probability P_c the job will not complete and need to be re-started. Thus, replication across spot instances is better for shorter jobs, since they have a lower probability of all replicas being revoked.

3.3.2 Replication on On-demand Instances

While replicating computation across many spot markets is useful, there always exists a non-zero probability of all replicas being revoked, which increases with the running time of the application and does not work well for parallel jobs (where a revocation of any instance requires restarting the job). Thus, another approach to replication is to execute a replica on an on-demand instance, which has a 0% revocation probability. Unlike replication across spot instances, this approach never requires re-starting a job from the beginning even for long-running jobs where there is a high probability of all replicas being revoked.

Of course, if SpotOn were to replicate a job on the same on-demand instance type selected by the user, there would be no benefit in using spot instances. As a result, SpotOn multiplexes multiple jobs on one on-demand instance, which effectively serves as a *replication backup server*. In this case, each job is given an isolated partition of the on-demand server's resources, such that the application executes slower than on a dedicated spot instance. SpotOn then accounts for the cost of the on-demand instance by partitioning its cost in proportion to the fraction of resources each job replica uses.

On revocation, SpotOn loses any work associated with the primary spot instance, which causes the job's progress to revert to that of the backup replica. SpotOn may then simply run the job at the slower rate, or acquire a spot instance in another market and migrate the backup server's job replica to it. This approach is similar to checkpointing in that, if the primary spot instance is revoked, there is some loss of work, since the backup replica's progress is behind the primary, but the job does not have to restart from the beginning. As before, there is a tradeoff between cost and the amount of work lost on a revocation: the higher the performance (and cost) of the backup replica, the less work is lost on a revocation and vice versa. Of course, the approach differs from checkpointing in that the work lost on each revocation is a function of the difference in resources on the primary and the backup, rather than a fixed checkpointing interval τ. Another difference is that the cost overhead of replicating computation is based on the spot prices in various markets, and is independent of the job's memory and disk footprint.

4. SpotOn Selection and Bidding Policies

For each job, SpotOn must determine in which spot market to execute it (and how to bid) and which fault-tolerance mechanism to use, with the goal of completing the job near the performance of an on-demand instance at a cost near that of spot instances. SpotOn must also determine where to resume a job if its current server is revoked.

Before detailing SpotOn's policies, we first define a job's slack, as a percentage of its estimated running time, which captures the additional time available for SpotOn to checkpoint and migrate jobs over their lifetime. SpotOn considers slack to be a user preference: the greater the slack the user permits, the more frequently SpotOn will checkpoint the job, and, thus, the longer the completion time when using checkpointing. In particular, for a job with slack S and time to checkpoint (from the previous section) T_c, the number of times SpotOn may checkpoint the job over its running time T without exceeding the slack is $\frac{T \cdot S}{T_c}$. Thus, the slack dictates a regular checkpointing interval of $\tau = \frac{T_c}{S}$.

4.1 Basic Server Selection Policy

We first define a basic policy that always chooses the spot market with the lowest normalized price for resources without considering the market's volatility. After choosing the lowest-cost spot market, this policy reactively migrates a job if it is possible, i.e., if its memory is less than 4GB, and always uses proactive checkpointing otherwise. On revocation, our basic policy migrates the job to an on-demand instance where it runs for the remainder of its lifetime.

4.2 Cost-aware Server Selection Policy

SpotOn employs a greedy cost-aware policy that selects the spot market and fault-tolerance mechanism in tandem to minimize a job's expected cost per unit of running time (modulo overhead) until it is either revoked or completes, given historical spot market prices and the job's resource usage and remaining running time. SpotOn generally re-evaluates its decision whenever a job's state changes, e.g., due to a revocation, in order to select a new instance type and market to migrate the job. That is, on revocation, SpotOn re-executes the greedy cost-aware policy to determine where to restore the job and the fault-tolerance mechanism to use based on the job's remaining running time.

We profile each spot market as a function of jobs' remaining running time. In particular, we define a random variable Z_k for each spot market k to represent the amount of time a job can run on a spot instance without being revoked. We then define the probability that Z_k is less than a job's remaining running time $P_z = P(Z_k \leq T)$, which represents the probability that a job's spot instance from market k is revoked before it completes. We use $E(Z_k)$ to denote the expected time a job executes before being revoked. For a given running time T, we can compute both $P(Z_k \leq T)$ and $E(Z_k)$ over a recent window of prices, e.g., the past day, week, or month. For each spot market k, we also maintain the average spot price $\bar{C}_{sp}^k$, excluding periods where the spot price exceeds the on-demand price for the equivalent instance. We use these values in computing the expected cost $E(C_k)$ for running each job in a particular spot market k.

Given a job's resource vector, our cost-aware policy uses a brute-force approach that simply computes the expected cost of using each fault-tolerance mechanism until the job either completes or is revoked across each spot market, and then chooses the least cost mechanism and market. Below, we show how to compute the expected cost for each of our fault-tolerance mechanisms. Note that SpotOn always configures jobs employing reactive migration and checkpointing to use remote storage, e.g., EBS in EC2, which may increase the running time of I/O-intensive jobs, and always configures jobs replicated across spot instances to use local storage. SpotOn adjusts the expected running time of the job based on its rate of I/O and the performance difference between local and remote I/O, such that using remote I/O has a longer running time for an I/O-intensive job than when using local I/O. Thus, when computing the costs, the input running time for each mechanism for the same job may differ based on whether the mechanism uses local or remote storage. Below, we detail the steps for computing the expected cost $E(C_k)$ and job execution time $E(T_k)$ for each mechanisms; the cost per unit of running time is then calculated as $\frac{E(C_k)}{E(T_k)}$.

4.2.1 Expected Cost of Migration

In this case, as shown below, the expected cost until the job either completes or is revoked is the probability the job is revoked (which is a function of its remaining running time) multiplied by the cost of running the job to the first revocation plus the probability the job finishes without being revoked multiplied by the cost of running the job to completion. On each revocation, the job incurs migration overhead T_m. Recall that P_z represents the revocation probability when running on a spot instance from market k.

$$E(C_k) = \left[P_z * (E(Z_k) + T_m) + (1 - P_z) * T\right] * \bar{C}_{sp}^k \quad (3)$$

After computing the expected cost $E(C_k)$, we compute the cost per unit of running time by dividing the expected job execution time until the job either finishes or is evicted, or $E(T_k) = (1-P_z)*T + P_z*E(Z_k)$. Since reactive migration is the best option if migration is feasible, SpotOn generally uses it whenever a job's memory footprint permits migration within the two minute warning.

4.2.2 Expected Cost of Checkpointing

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

$$E'(T_k) = E(T_k) - \frac{E(T_k)}{\tau} * T_c - \frac{\tau}{2} \tag{4}$$

Here, $E(T_k)$ is the same as with reactive migration, the second term represents the downtime due to checkpointing over $E(T_k)$, and $\frac{\tau}{2}$ is the expected work lost on each revocation, assuming that revocations are uniformly distributed over each checkpoint interval.

4.2.3 Expected Cost of Replication

Finally, we derive the cost for each replication variant.
Replication across Spot Instances. When replicating across spot instances, we do not re-run our selection policy on each revocation. Instead, if all spot instances are revoked, we restart the job on an on-demand instance to ensure the job completes. The expected cost when replicating a job with remaining running time T across n spot markets is the expected cost if all spot instances are revoked plus the expected cost if the job completes, weighted by the probability of each event occurring. Here, P_c and P_r are the probability of a job completing and being revoked from Section 3.3.1; C^k_{od} is the price of the on-demand instance for market k; and $E(T_k)$ is the expected running time until the instance from market k is revoked.

$$E(C_k) \ = \ P_r\Big(\sum_{k=0}^{n} \bar{C}^k_{sp}E(T_k) + C^k_{od} * T\Big) + P_c\sum_{k=0}^{n} \bar{C}^k_{sp}T \tag{5}$$

As before, we use the expected cost to compute a cost per unit of the expected amount of useful work completed.
Replication on On-demand Instances. Computing the cost of replicating on a "slower" and cheaper on-demand instance is similar to checkpointing, except that we incur an additional cost for the discounted on-demand instance. Here, we assume SpotOn pays the same price for the backup on-demand instance as it does for the primary spot instance, which mirrors the price for replicating across two spot instances above, and makes the different replication approaches comparable. As we discuss in Section 4.3, our bidding policy only replicates across two spot instances. In this case, if the ratio of the on-demand to spot price is r, then we assume the remaining running time of our job on the backup instance is $r*T_i$, since we partition the resources of the backup on-demand instance based on its price. The expected cost below is then similar to checkpointing, but multiplies the price of the spot instance by a factor of two to account for the cost of the primary spot instance and the backup on-demand instance.

$$E(C_k) \ = \ P_z * (2E(T_k)\bar{C}^k_{sp}) + (1 - P_z) * (2T\bar{C}^k_{sp}) \tag{6}$$

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

$$E(T_k) = P_z * \frac{E(T_k)}{r} + (1-P_z)*T \tag{7}$$

4.3 Bidding Policies

The probability of revocation and the expected time to a revocation in any spot market is based on the bid's value, which SpotOn can adjust. Note that, since EC2 caps the maximum bid price at 10× the on-demand price, SpotOn cannot reduce the probability of revocation to 0%. Since we assume that on-demand instances are available (of some type in some availability zone/region) at a fixed price with a 0% probability of revocation, we define a bidding budget such that SpotOn does not exceed the on-demand price for spot instances. Since migration and checkpointing impose some performance overhead that increase a job's running time and cost, SpotOn bids a price equal to the cost of running the job on an on-demand instance divided by the expected running time on the spot instance (including any overhead).

SpotOn could also adjust its bid price to alter a spot instance's revocation probability. However, in prior work [16], we show that bidding slightly more (or less) than the on-demand price does not significantly decrease (or increase) the revocation probability, as current market prices tend to spike from very low to very high. Thus, we do not consider adjusting the bid price in this paper, although our approach could be extended to support such variable bid strategies.

With replication, there is no performance overhead during normal execution. In this case, when replicating across spot instances, we divide the on-demand price across the degree of replication and bid that value for each spot instance. Figure 5 plots the probability of completing a job without all replicas being revoked as a function of job duration for

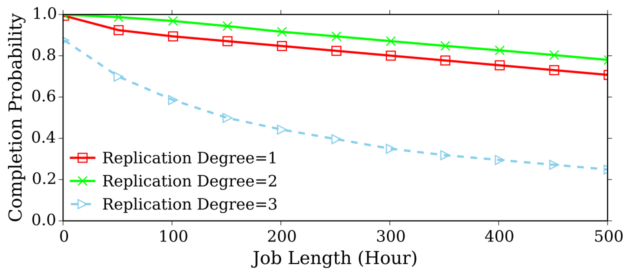


Figure 5. When replicating across spot instances, a replication degree greater than two decreases the probability of completing the job before all instances are revoked.

different replication degrees across all spot markets in the `us-east-1a` availability zone. The figure demonstrates that, while replication degree of two improves the probability of completion, especially for short jobs, higher replication degrees decrease the probability due to spreading the bidding budget across more instances. As a result, in this paper, when replicating across spot instances, we only use a replication degree of two. Finally, when replicating on a backup on-demand server, we discount our bid to ensure the maximum price we pay for both the spot instance and the on-demand backup server is not greater than the price of a dedicated on-demand instance based on the ratio r above.

5. Prototype Implementation

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

SpotOn's job manager selects the EC2 spot market and fault-tolerance mechanism for each job based on the cost-aware policy in Section 4.2. To execute the policy, the job manager monitors and records spot prices across EC2 markets. For each market, the job manager computes the expected cost of each fault-tolerance mechanism using the historical price data, as well as the the job's running time and resource usage vector. Our current prototype assumes a job's running time and resource usage vector are accurate and does not monitor a job's resource usage while it is running. In addition, our current prototype does not support "phased" jobs, where resource usage changes significantly during different phases of execution. After computing the expected cost for each market and fault-tolerance mechanism, the job manager selects the least cost fault-tolerance mechanism and spot market combination to run the job and bids based on the policies in Section 4.3. The job manager interacts with EC2 to monitor prices, place bids, and fetch instance information using the EC2 web services APIs. If the current spot price in the market is above the on-demand price, then the job manager selects the market with the next lowest expected cost.

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

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

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

6. Experimental Evaluation

The goal of our evaluation is to quantify the benefit of SpotOn's cost-aware selection policy that chooses the fault-tolerance mechanism and spot market to minimize costs, while mitigating the impact of revocations on job completion time. We compare the cost and performance of our pol-

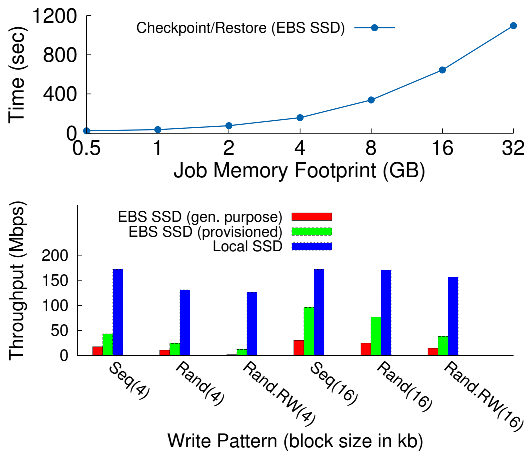


Figure 6. The time to checkpoint/restore a container is a function of a job's memory footprint (top). The I/O throughput for local disks is an order of magnitude greater than for remote disks over a range of workloads (bottom).

icy with three other policies: a control policy that always executes jobs on an on-demand instance, our basic policy from Section 4.1 that always selects the lowest price spot market using checkpointing and reverts to an on-demand instance on the first revocation, and a variant of our cost-aware policy that only uses checkpointing. We conduct experiments using our prototype and in simulation. The prototype experiments demonstrate the impact of resource usage and price characteristics on real jobs, while the simulations assess the impact on performance and cost when using our cost-aware policies to execute multiple jobs over time with realistic price traces.

We first conduct microbenchmarks to verify the assumptions of our models in Section 3 and to seed our simulator. In particular, we plot LXC checkpoint/restore time in Figure 6(top) as a function of a job's memory footprint to verify relationship between checkpoint/restore overhead and memory. The graph demonstrates that it is possible to migrate jobs that use less than roughly 4GB of memory within EC2's two-minute warning time. In addition, for Figure 6(bottom) we use the FIO tool to measure the local versus remote EBS storage throughput for multiple I/O workloads (in this case using the SSD variant of EBS); we see that, as expected, the local I/O throughput is an order of magnitude larger than the remote EBS throughput, which favors using local storage for I/O-intensive jobs. Our simulator uses Figure 6 to compute a job's checkpoint/restore and I/O overhead based on its resource usage. The simulator also imposes delays of 62 seconds and 224 seconds for booting an on-demand and spot instance, respectively, based on our experiments.

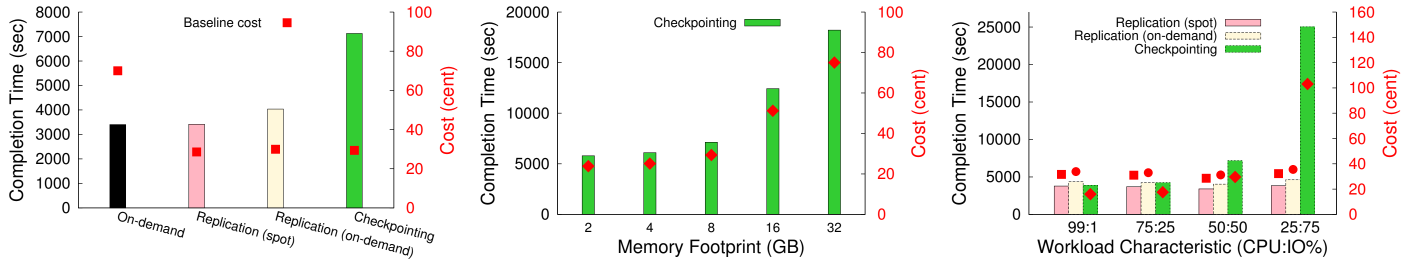
6.1 Prototype Results

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

For our baseline experiment, we assume the cost of the spot instance is 20% of the cost of the on-demand instance and the revocation rate is 2.4 revocations per day (or 0.1 revocations per hour). We chose 2.4 revocations per day as a median between the extreme values in Figure 1 and the many markets that currently experience nearly zero revocations per day. We execute the job on a `r3.2xlarge` instance type, which costs 70¢ per hour, and measure its average completion time across multiple runs to be 3399s. Figure 7a shows the job's completion time (each bar corresponding to the left y-axis) and its cost (each dot corresponding to the right y-axis) when running on an on-demand instance versus running on a spot instance and i) replicating on a backup on-demand instance, ii) replicating across two spot instances, and iii) checkpointing every 15 minutes. To fairly compare the two replication approaches, when replicating on a backup on-demand instance we assume the job runs at 20% the performance of the dedicated instance and is charged 20% of the cost of the backup.

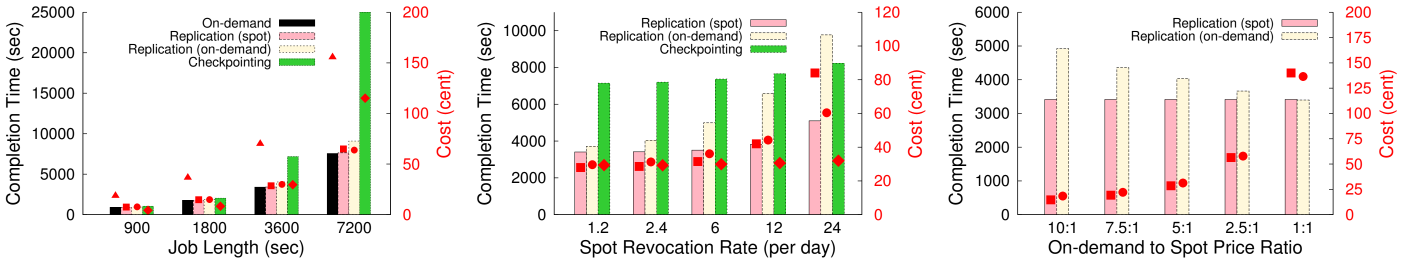
Our baseline experiment shows that both forms of replication and checkpointing reduce the job's cost by over a factor of two compared to running on an on-demand instance. However, both replication mechanisms complete the job sooner than when using checkpointing. The reason is that the probability of revocation over the job's running time is only 10%, so 90% of the time the job will finish without incurring any performance overhead due to a revocation. In contrast, checkpointing repeatedly incurs the overheads from Figure 6. In addition, checkpointing requires using a remote disk to facilitate migration, while replication is capable of using the local disk. Thus, replication benefits from the I/O intensity of our baseline job. Note here that the cost of replicating on a backup server and checkpointing is similar, since the backup server doubles the cost (as we fix the amount we pay for the backup server to be equal to that of the spot instance), while checkpointing nearly doubles the running time, which also doubles the cost.

Figure 7 also plots the job's performance and cost as its memory footprint and CPU:I/O ratio change. Figure 7b shows that, as expected, an increase in the memory footprint causes an increase in the overhead of checkpointing, while it has no effect on the replication approaches. Figure 7c then shows that, as the job becomes more I/O-intensive, the job completion time and cost of checkpointing rise due to the need to use remote I/O. In contrast, the cost and performance of the replication approaches remain constant. Note that for CPU-intensive jobs the cost of replication is slightly more than the cost of checkpointing, as there is less benefit to



(a) Baseline (b) Memory Footprint (c) CPU:I/O Ratio

Figure 7. Performance and cost for our baseline job when running on an on-demand instance versus running on spot instances using different fault-tolerance mechanisms (a). We also plot the job's performance and cost as its memory footprint (b) and CPU:I/O ratio (c) vary, while keeping other job attributes constant.



(a) Job Duration (b) Revocation Rate (c) On-demand:Spot Price Ratio

Figure 8. The impact of varying job duration (a), spot instance revocation rate (b), and on-demand:spot price ratio (c) on our baseline job's performance and cost, while keeping other attributes constant.

using the local disk, but both variants of replication incur the cost of additional compute resources.

Figure 8 plots the job's performance and cost for other relevant parameters, including job length, the revocation rate, and the ratio of on-demand-to-spot prices. Figure 8a shows that checkpointing has the lowest cost for short jobs ($<$ 1 hour), since short jobs require fewer checkpoints and less overhead. However, the longer the job, the higher checkpointing's overhead and cost. While the overhead of both replication variants also increase with job duration, due to the increased probability of losing work due to revocation, the increase is less than with checkpointing.

Figure 8b shows that as the revocation rate increases the cost and performance of replication becomes worse relative to checkpointing. Replication is highly sensitive to the revocation rate, since revocation's result in rolling back to either the progress of the slower backup server or to the start. In contrast, checkpointing's cost and performance is more robust to an increasing revocation rate, since it only loses at most the smaller time window between each checkpoint. The figure also demonstrates the key difference between replication across spot and replication on on-demand: under a high revocation rate (24 per day) replication across spot has low running time, but a high cost (since it reverts to using an on-demand instance), while replication on on-demand has a higher running time but a much lower cost, since it always makes progress. Finally, Figure 8c shows that as spot prices rise relative to the on-demand price, the replication variant that uses an on-demand backup server takes longer to complete. This is due to increased multiplexing of jobs on the backup server at a higher spot price. Since checkpointing and replication across spot do not use an on-demand backup server, they are robust to this effect.

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

6.2 Policies and Cost Analysis

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

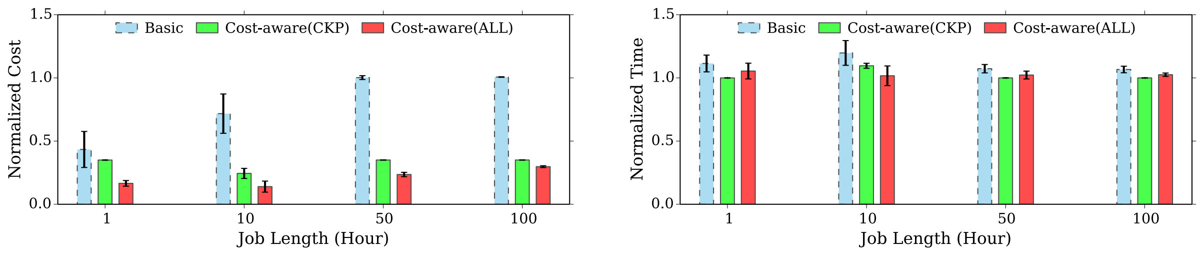


Figure 9. SpotOn's cost-aware policy has the lowest cost and similar performance to an on-demand instance.

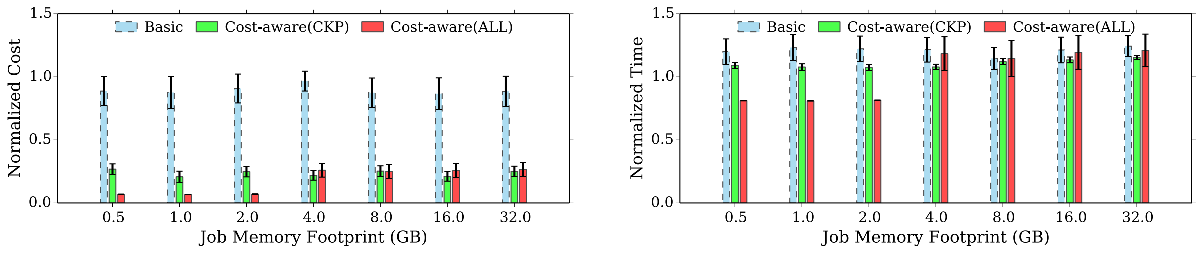


Figure 10. SpotOn's cost-aware policy has the lowest cost and similar performance to an on-demand instance. The cost is substantially lower when jobs' memory footprint is less than 4GB, since reactive migration is feasible for such jobs.

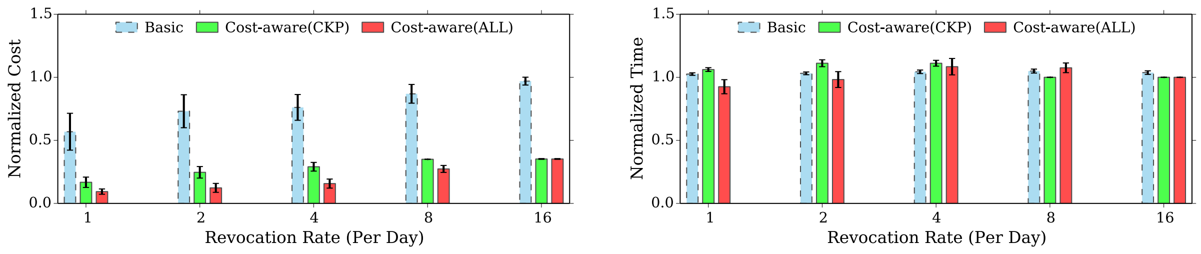


Figure 11. Job cost (a) and performance (b) for each policy as a function of the revocation rate.

We first evaluate SpotOn's selection policies as we vary the duration of the jobs. We simulate the execution of 30 jobs arriving randomly over the three-month time window and record the cost and completion time for each of our policies. Figure 9 shows the results normalized to the cost and completion time when using an on-demand instance. We include error bars for the 95% confidence intervals over the 30 jobs. The results demonstrate that all cost-aware policies incur a lower cost than using an on-demand instance for a similar performance level. In addition, our basic policy, which always selects the spot instance with the lowest price without regard to volatility and migrates to an on-demand instance after the first revocation, has a significantly higher cost than either of our cost-aware policy variants, which demonstrates the benefits of considering volatility in addition to price when choosing a market. Our cost-aware policy also has a lower cost than a variant that only uses checkpointing, which demonstrates the benefit of using replication in addition to checkpointing. However, the cost benefit of replication decreases as job duration increases, since the probability of revocation increases (which in turn increases the overhead of replication). Finally, the completion time for each approach is similar and near the completion time of the job on an on-demand instance.

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

Next, Figure 10 examines the impact of a job's memory footprint on each policy. As before, all policies reduce costs relative to on-demand with the same level of performance. In fact, for small jobs less than 4GB, the completion time is lower than using an on-demand instance. This is due to inversions in spot market prices, where the spot price of a particular instance type drops below the spot price of a smaller instance type. Since SpotOn always seeks the lowest-cost resources, it takes advantage of these price inversions. Figure 10 also shows that jobs with memory footprints that use less than 4GB incur a much lower cost and have higher performance than jobs that use more memory. This occurs because reactive migration is feasible in this case, and reactive migration does not incur the performance overhead of checkpointing or the cost overhead of replication. When reactive migration is not possible after 4GB, the cost-aware and cost-aware checkpointing policies have a similar cost with performance similar to an on-demand instance.

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

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

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

Lastly, to get a sense of SpotOn's potential for savings with a real workload, we randomly select 1000 tasks from a Google cluster trace [15] and compare the cost of SpotOn's greedy cost-aware policy and running the jobs on an `m1.large` on-demand instance. Our results show a cost savings of 91.9% when using SpotOn's cost-aware policy versus the `m1.large` on-demand instance. In addition, the total running time across all jobs when using SpotOn actually *decreases* by 13.7%. In this case, the decrease occurs because SpotOn often chooses to execute jobs on spot instance types that are faster than the `m1.large` because their spot price is actually cheaper than an `m1.large` on-demand instance.

7. Related Work

SpotOn is similar to recent startup companies, such as ClusterK [2], that offer low prices by executing batch jobs submitted by users on spot instances. However, their policies for handling revocations are not public, so it is unclear if they restart jobs if spot instances fail, or if they use fault-tolerance mechanisms to mitigate the impact of revocations.

In recent work, we propose a derivative cloud platform to transparently mask spot instance revocations from interactive applications [16]. The platform runs applications in nested VMs, which continuously checkpoint their memory state to a backup server. When notified of an impending revocation, the platform requests an on-demand instance and uses the backup server to migrate the nested VM within the two minute period between the revocation warning and spot instance termination. To ensure transparency for interactive applications, these migrations must minimize their downtime, which precludes migrating between regions or using local storage. By focusing narrowly on batch jobs that permit some downtime, SpotOn has much more flexibility, enabling it to chose from multiple fault tolerance mechanisms, exploit spot markets in multiple regions, and use local storage.

Prior work examines bidding [12, 13, 18, 19, 23–25] and checkpointing [9, 21, 22] policies for batch jobs to minimize the cost of spot instances and mitigate the impact of revocations. This work generally evaluates bidding and checkpointing policies in simulation without considering how job resource usage affects their overhead (and cost) relative to other fault-tolerance mechanisms. Instead, the simulations often assume the overhead is small and do not take into account the difference between using local versus remote I/O.

SpotOn also differs from the prior work above in its focus on a service that selects the spot market and fault-tolerance mechanism with the lowest expected cost. Prior work [11, 12] focuses on using only one fault-tolerance mechanism within a single spot market. One exception is work by Voorsluys and Buyya [21], which considers replicating computation across two spot instances. However, since they only consider simulated compute-intensive jobs where the cost of checkpointing is low, they find replication performs poorly by comparison; our results indicate replication is effective at current spot prices, especially for I/O-intensive jobs, since it enables use of local storage.

8. Conclusion

SpotOn optimizes the cost and performance of running non-interactive batch jobs on the spot market. Our results demonstrate that the spot market has significant arbitrage opportunities available, which SpotOn exploits to transparently lower costs by packaging jobs in containers and using existing fault-tolerance mechanisms to mitigate the impact of revocations. Our current work only considers static switching between individual fault-tolerance mechanisms at revocation boundaries. As part of future work, we plan to explore multiple extensions, including i) mixed policies that combine multiple fault-tolerance mechanisms, such as periodically checkpointing the on-demand backup server or any spot replicas, ii) adaptive policies, which switch mechanisms as the remaining runtime decreases or spot price characteristics change, and iii) variable bid policies, which consider setting the bid price to adjust the revocation probability.

Acknowledgments

We would like to thank our shepherd Ashraf Aboulnaga for his valuable feedback, which improved the quality of this paper. This work is funded by the National Science Foundation under grants #1422245, #1405826, and #1229059.

References

[1] Scientific Computing Using Spot Instances. `http://aws.amazon.com/ec2/spot-and-science/`, June 2013.

[2] ClusterK. `https://clusterk.com/`, July 10th 2015.

[3] Google preemptible instances. `https://cloud.google.com/compute/docs/instances/preemptible`, July 10th 2015.

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

[5] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Waeldr. Live Migration of Virtual Machines. In *NSDI*, May 2005.

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, April 2008.

[7] S. Di, D. Kondo, and F. Cappello. Characterizing and Modeling Cloud Applications/Jobs on a Google Data Center. *Supercomputing*, 69(1), July 2014.

[8] M. R. Hines, U. Deshpande, and K. Gopalan. Post-copy Live Migration of Virtual Machines. *SIGOPS Operating Systems Review*, 43(3), July 2009.

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

[10] M. Mao and M. Humphrey. A Performance Study on VM Startup Time in the Cloud. In *CLOUD*, June 2012.

[11] M. Mattess, C. Vecchiola, and R. Buyya. Managing Peak Loads by Leasing Cloud Infrastructure Services from a Spot Market. In *HPCC*, September 2010.

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

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

[14] A. Mu'alem and D. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *TPDS*, 12(6), June 2001.

[15] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google Cluster-usage Traces: Format + Schema. Technical report, Google Inc., November 2011.

[16] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *EuroSys*, April 2015.

[17] R. Singh, D. Irwin, P. Shenoy, and K. Ramakrishnan. Yank: Enabling Green Data Centers to Pull the Plug. In *NSDI*, April 2013.

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

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

[20] D. Tsafrir, Y. Etsion, and D. Feitelson. Modeling User Runtime Estimates. In *JSSP*, 2005.

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

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

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

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

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