Transient Guarantees: Maximizing the Value of Idle Cloud Capacity

Supreeth Shastri, Amr Rizk, and David Irwin
University of Massachusetts Amherst

Abstract*—To prevent rejecting requests, cloud platforms typically provision for their peak demand. Thus, a platform's idle capacity can be significant, as demand varies widely over multiple time scales, e.g., daily and seasonally. To reduce waste, platforms have begun to offer this idle capacity in the form of *transient servers*, which they may unilaterally revoke, for much lower prices—$\sim$50-90% less—than on-demand servers, which they cannot revoke. However, transient servers' revocation characteristics—their volatility and predictability—influence their performance, since they affect the overhead of fault-tolerance mechanisms applications use to handle revocations. Unfortunately, current cloud platforms offer no guarantees on revocation characteristics, which makes it difficult for users to optimally configure (and correctly value) transient servers. To address the problem, we propose the abstraction of a *transient guarantee*, which offers probabilistic assurances on revocation characteristics. Transient guarantees have numerous benefits: they increase the performance of transient servers, enable users to optimally use and correctly value them, and permit platforms to control their freedom to revoke them. We present policies for partitioning a variable amount of idle capacity into classes with different transient guarantees to maximize performance and value. We then implement and evaluate these policies on job traces from a production Google cluster. We show that our approach can increase the aggregate revenue from idle server capacity by up to $\sim$6.5$\times$ compared to existing approaches.

I. Introduction

Cloud computing is growing rapidly with global spending on Infrastructure-as-a-Service (IaaS) platforms estimated to increase by nearly 33% in 2016 [1]. This growth has led to intense competition between providers, such as Amazon's Elastic Compute Cloud (EC2), Google Compute Engine (GCE), and Microsoft Azure, to aggressively expand their infrastructure. The goal is to offer low prices and the illusion of access to infinite computing resources on demand. Of course, platforms do not have infinite capacity: if the demand for servers exceeds the supply, they must reject user requests. Since rejecting requests may turn away new users, platforms provision conservatively for their expected peak demands.

Thus, a platform's *idle server capacity* can be significant, as the demand for cloud servers varies widely over multiple time scales, including daily and seasonally. Importantly, any idle capacity partially wastes the capital and operational expenses providers incur to house, provision, maintain, power, and cool the idle servers. To recoup some of these costs, providers have begun to offer idle capacity in the form of *transient servers*, which they may unilaterally revoke from users. Transient servers are typically much cheaper—$\sim$50%-90% less on average—than *on-demand servers*, which a platform cannot revoke. Transient servers enable platforms to gain revenue from their idle capacity, while retaining the flexibility to reclaim servers to satisfy higher-priority requests, e.g., for on-demand servers. Transient servers also benefit users by reducing the cost to execute workloads that can handle server revocations, and are tolerant to unexpected delays or performance degradation. Since high-performance computing (HPC) and scientific workloads are often delay-tolerant, and thus particularly well-suited to exploit transient servers, providers explicitly market them to these workloads [2], [3], [4], [5]. Similar to other financial investments, transient servers enable HPC workloads to pay a lower price for resources in exchange for a higher risk tolerance and less strict performance requirements. Note that transient servers also arise in private clusters that support separate background and foreground tasks such that background tasks are always revoked to make room for new foreground tasks [6].

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

Our key insight is that *transient servers' revocation characteristics influence their performance relative to on-demand servers*, since these characteristics affect the overhead of the fault-tolerance mechanisms applications employ to handle revocations. For example, consider a batch job that runs on a transient server and periodically checkpoints its memory state to a remote disk, such that after a revocation the job can restart from its last checkpoint. Such periodic checkpointing incurs an overhead that increases job running time and decreases the perceived performance of a transient server relative to an on-demand server that requires no checkpointing. In addition, the optimal checkpointing interval that minimizes this fault-tolerance overhead is a function of transient servers' mean-

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 ©2016 IEEE

time-to-revocation (MTTR) [7]. As a result, estimating an MTTR that is too low or too high results in checkpointing too much or too little, respectively, causing the job's running time to further increase due to either excessive checkpointing or recomputation. Thus, inaccurate knowledge of the MTTR can also decrease transient server performance by causing users to misconfigure their checkpointing interval.

The example above highlights that knowing even probabilistic information about a transient server's revocation characteristics can increase its performance by enabling users to optimally configure fault-tolerance mechanisms. Unfortunately, the revocation characteristics for EC2 and GCE are unknown and unbounded. In EC2, revocation characteristics derive from real-time changes in the spot price, which are based on instantaneous supply and demand.[1] While EC2 provides three months of price history, historical prices are not necessarily an accurate predictor of future prices. In addition, as the spot market matures it should increasingly follow the efficient market hypothesis, which states that you cannot "beat the market" by predicting future prices, as current prices reflect all available information. In contrast, GCE provides users no revocation information. Thus, users are unable to accurately configure fault-tolerance mechanisms, such as the checkpointing interval, to maximize transient server performance.

Due to the lack of information, EC2 and GCE users are also unable to accurately quantify transient server value. For example, while a transient server may be 50% the price of an on-demand server, its unknown revocation characteristics may result in a 50% performance overhead due to fault-tolerance. Thus, "cheaper" transient servers may actually offer no normalized discount relative to on-demand servers.

To address the problem, we propose the abstraction of a *transient guarantee*, which provides probabilistic assurances on revocation characteristics. Our hypothesis is that transient guarantees enable users to maximize and quantify the performance and value of transient servers, while enabling platforms to control the freedom to revoke them for higher-priority tasks. Since transient guarantees increase transient server value (by increasing their performance), platforms can employ them to either increase their revenue (by increasing prices to reflect value) or increase their discounts to users (by maintaining current prices and offering higher performance). In evaluating our hypothesis, we make the following contributions.

Transient Server Characterization. We characterize the performance and value of applications using transient servers after accounting for the overhead of fault-tolerance. In doing so, we highlight three key metrics—availability, volatility, and predictability—that affect application performance on transient servers, and evaluate how these general characteristics impact perceived performance (modulo revocation-related overheads).

Transient Guarantees. We define the transient guarantee abstraction, which provides probabilistic assurances of revocation characteristics. We show how transient guarantees benefit users by enabling them to quantify transient server performance and value relative to on-demand servers. We then show how they benefit platforms by designing policies that partition idle capacity into multiple classes with different transient guarantees to maximize aggregate performance.

Implementation and Evaluation. Finally, we implement our policies, and evaluate them using job traces from a production Google cluster. We show that partitioning idle capacity into as few as four classes with different strength transient guarantees increases its aggregate revenue by up to $\sim$6.5$\times$ compared to using EC2 spot instances or GCE preemptible instances.

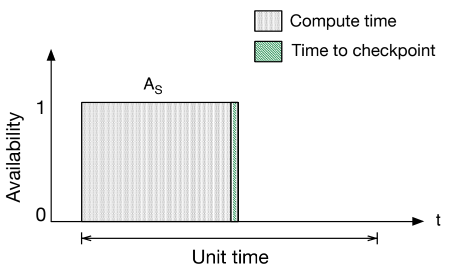
[1]Note that, while data analysis initially indicated that EC2 spot prices may not be supply-demand driven, follow-on work has shown that prices have been characteristic of a true market since October 2011 [8].

II. Background

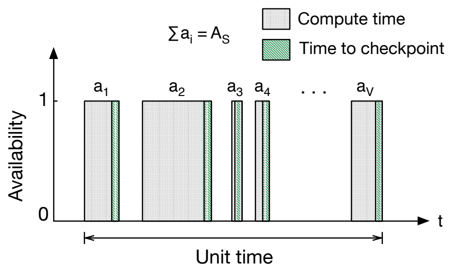
Our work assumes a cloud platform that sells a fixed capacity of virtual machines (VMs) to users under multiple contracts that offer different levels of risk and cost. The most common contract is for *on-demand servers*, which users may request at any time and, once allocated, a platform cannot unilaterally revoke. Note that on-demand servers are not guaranteed to be available: platforms may occasionally reject requests for on-demand servers due to a lack of resources [9]. Importantly, on-demand servers restrict a platform's control over its resources, as only users can decide how long they hold on-demand servers and when they release them. As a result, allocating too many on-demand servers may prevent a platform from satisfying requests for *reserved servers*, which, unlike on-demand servers, it guarantees are always available upon request. Cloud platforms allocate reserved servers because many users do not want to risk that on-demand servers will not be available at a critical business time, e.g., during an unexpected surge in demand. To support reservations, platforms have only two options: *either keep physical resources idle or maintain a pool of resources they can reclaim to satisfy reserved requests*. Of course, keeping physical servers idle is highly inefficient, as it wastes their computational resources, as well as the capital and operational expenses incurred to provide them. Thus, transient servers exist both to reduce this waste by enabling platforms to earn revenue from their idle capacity, and also to provide a pool of revocable resources to support reservations.

A. Transient Server Characteristics

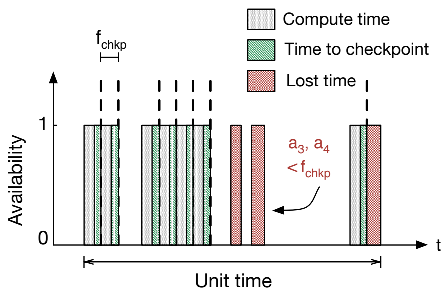
The price of on-demand, reserved, and transient servers reflect their different levels of risk. Transient servers in EC2 and GCE are significantly cheaper because they entail an unbounded risk of revocation, as EC2 and GCE may revoke them at any time. Handling revocations not only introduces additional application complexity, but also additional performance overheads, which decrease transient server performance. While applications can reduce this overhead, given sufficient knowledge of revocation characteristics, *they cannot eliminate it*. Thus, transient server performance is strictly less than on-demand (or reserved) performance. We distill the revocation characteristics that influence performance into three independent metrics: availability, volatility, and predictability.



(a) Available, Not Volatile, Predictable



(b) Available, Volatile, Predictable



(c) Available, Volatile, Not Predictable

Fig. 1: Availability, volatility, and predictability are three distinct metrics that affect transient server performance.

Below, we discuss these metrics in the context of EC2, as GCE releases no information on, and provides no control over, them.

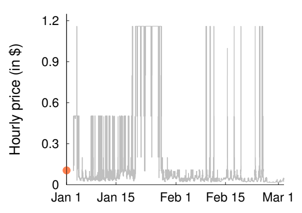
- **Availability** is the percentage of time a transient server is available—in EC2, this translates to the percentage of time the spot price is below a user's bid price.
- **Volatility** is the frequency of transient server revocations—in EC2, this translates to the frequency at which the spot price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

Figure 1 illustrates, in the context of our simple batch job from Section I, that these three metrics are distinct from, and independent of, each other. Figure 1(a) shows a time-series of transient server availability that is not volatile and highly predictable. In this case, there is only a single revocation at a well-known time. As a result, the application need only checkpoint immediately before the revocation occurs, thereby minimizing its overhead (in green) and maximizing the useful work it performs (in grey). In contrast, Figure 1(b) shows a similar time-series with the same availability over time, but with a higher volatility that includes many revocations. In this scenario, the application incurs more overhead (in green) than before because it needs to checkpoint much more frequently. However, since the time of each revocation remains well-known and predictable, it still need only checkpoint immediately prior to each revocation. Finally, Figure 1(c) shows a time-series again with the same availability, but with a high volatility and low predictability. Here, the application incurs a higher checkpointing overhead (in green), since it does not know precisely when revocations will occur, and must instead periodically checkpoint at a fixed interval. In this case, the application also incurs some recomputation overhead (in red) when it loses work after an unexpected revocation.

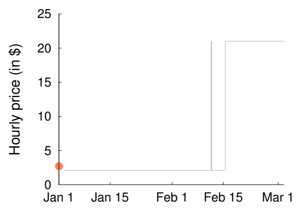
Our simple example illustrates that volatility and predictability affect transient server overhead and performance much more than availability. Despite this, prior work focuses largely on optimizing availability in EC2—by determining the bid that minimizes cost, while allowing an application to satisfy a performance target, e.g., a deadline [10], [11], [12], [13] or specified availability [14]. However, we contend that *there is no reason to ever wait for a particular transient server to become available*, since cloud platforms are large enough that resources are effectively always available somewhere (at some price). For example, EC2 operates ~2500 spot markets (one for each server configuration and type in each availability zone of each region), each with its own dynamic spot price, with nearly 1000 spot markets in the U.S. East region alone. GCE offers preemptible instances on a similar scale. Thus, applications can always immediately resume execution from saved state on the lowest-cost available server. The only reason for an application to wait is if it decides the price of a platform's lowest-cost server is too high.

Recent work has recognized that resources are nearly always available in the cloud, and focuses on continuously migrating to the lowest-cost resources [15], [16], [17], [18], [19]. This work exploits the increasing maturity and popularity of resource containers [20] and nested virtual machines [21], which provide systems-level checkpointing and migration for applications. As a result, volatility and predictability—and not availability—are the critical metrics that affect transient server overhead and performance. Prior work likely does not focus on these metrics because EC2's current spot market is predictable and not volatile—prices generally remain low and stable for long periods. However, as more users exploit the spot market's arbitrage opportunities (by using spot instances when the spot price is low, and migrating to on-demand instances when it rises), spot prices will not only rise, but also become more volatile and less predictable. This will ultimately decrease the performance and value of using spot instances [22].

There are indications this is already happening, as multiple startup companies are now working aggressively to exploit the spot market's arbitrage opportunities [23], [24]. These companies automate the process of optimally selecting spot instances that balance low cost with low risk, and reconfiguring applications on revocation, e.g., by selecting and spawning new spot instances elsewhere. As an example of the potential effects of large-scale exploitation of these optimizations, Figure 2 shows spot prices for the `c4-large` and `cg1-4xlarge` over two months in one availability zone of the U.S. East region. We expect that the c4-large is a much higher volume



(a) c4-large



(b) cg1-4xlarge

Fig. 2: Spot prices in mature markets (a) are more volatile and less predictable than spot prices in immature markets (b).

and mature market than the cg1-4xlarge market, since the c4-large is a current generation compute-oriented server, while the cg1-4xlarge is a more exotic GPU-oriented server that is deprecated. The graphs show that prices for the more mature c4-large market are significantly more volatile and unpredictable, and often rise above the on-demand price (the red dot represents the on-demand price for each server).

While prior work uses exponential and Pareto distributions to accurately model the probability the spot price equals a particular value in select markets [10], these models are only useful in predicting availability at a given bid price: they do not model or predict when and how often revocations occur. Our analysis indicates that the distribution of inter-arrival times for revocations (at each bid level) varies widely across EC2's $\sim$2500 markets, and is much harder (or impossible) to capture using a single one-size-fits-all model.

B. Quantifying the Performance Impact

We quantify the performance impact of revocation characteristics for HPC-oriented batch applications that use checkpointing to handle server revocations. We focus on batch applications, since these are commonly run on transient servers [2], [3], [4]. We discuss other types of applications in Section VI. In addition, we assume these applications have non-trivial memory footprints that prevent dynamically checkpointing memory state after a platform notifies a server of impending revocation, but before server termination. Current revocation warnings for transient servers range from thirty seconds (on GCE) to two minutes (on EC2), which prevents such dynamic checkpointing once memory footprints exceed 1GB to 4GB, respectively. Note that the current trend is towards shorter revocation warning times, as platform's are placing a higher priority on short re-provisioning and booting times [25]. Platform's revoke transient servers to reallocate them to satisfy other higher-priority requests, e.g., for on-demand and reserved instances. As a result the re-provisioning and boot times for on-demand and reserved instances must be greater than the warning time. Thus, platform's cannot arbitrarily increase their warning time to accommodate the dynamic migration of applications after a warning without increasing the boot times for high-priority on-demand and reserved instances.

Thus, applications with non-trivial memory footprints must employ checkpointing to ensure forward progress and prevent restarting from the beginning after each revocation. Based on prior work [7], the optimal checkpointing interval that minimizes application running time when accounting for the overhead of recomputation and checkpointing is below.

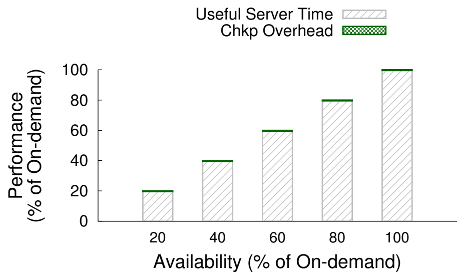
$$t_{opt} \sim \sqrt{2 * \delta * MTTR} \qquad (1)$$

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

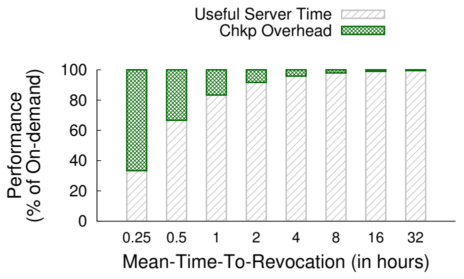
Figure 3 then shows how the performance of transient servers (as a fraction of on-demand server performance) varies based on availability, volatility, and predictability. Here we model the transient server revocations as following a Poisson process with a specified MTTR over a two week period. We then simulate running a batch job on a transient server with a 16GB memory footprint, which incurs a checkpointing overhead of $\sim$10 minutes on EC2 using an EBS magnetic disk. We consider 16GB a medium-sized memory footprint, as we assume the use of systems-level mechanisms that checkpoint the memory of an entire server. Currently, 28 of the 40 instance types offered by EC2 have >=16GB memory. Of course, a larger memory footprint would increase the checkpointing overhead resulting in a larger δ. In this example, since we use system-level checkpointing of the entire memory footprint, we also assume that applications are well-matched to the transient server's memory size. If an application uses significantly less memory than a transient server offers, it should acquire a smaller (and cheaper) server. In addition, each revocation in EC2 would also incur an additional two minute overhead to acquire and boot a replacement server. As before, we show the useful server-time in grey, the checkpointing overhead in green, and the recomputation overhead in red.

Figure 3(a) shows that, as expected, the percentage of time a transient server is available is linearly related to its rate of computation: if the server is only available 50% of the time, its rate of computation is at most 50% that of an on-demand server. However, as we discuss, the availability of any single transient server is not an important metric, as cloud platforms are large enough that servers of some type are nearly always available. In contrast, Figure 3(b) demonstrates the impact of volatility on performance over a range of MTTRs. As in Figure 1(b), this figure assumes revocation times are entirely predictable, and thus represents the minimum overhead of transience at each MTTR. As the figure shows, transient server performance is 35%-70% less than an on-demand server with MTTRs of 0.25-1 hour even for a modest-sized 16GB server.

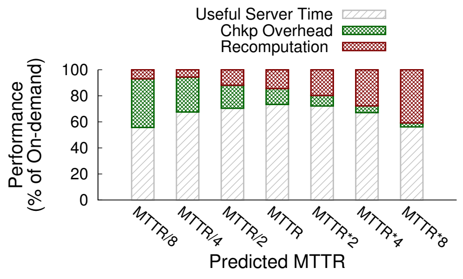
Finally, Figure 3(c) shows the impact of unpredictability on performance. Here, we fix the MTTR at four hours, but assume the precise revocation times are not known. We then plot the overhead due to checkpointing and recomputation for different estimates of the unknown MTTR. As the figure shows, even with a correct MTTR, the overhead increases by more than 6$\times$ compared to Figure 3(b) where the revocation times are known (from $\sim$4% overhead in (b) to $\sim$27% overhead in (c)). Thus,



(a) Available, Not Volatile, Predictable



(b) Available, Volatile, Predictable



(c) Available, Volatile, Not Predictable

Fig. 3: Impact on transient server performance when (a) varying availability, (b) varying volatility at a given level of availability, and (c) varying predictability at a given level of availability and volatility.

unexpected server revocations can incur substantial overhead. The figure also shows that inaccurate MTTR estimates further increase this overhead. Note that GCE releases no information on MTTR to guide users, while EC2's price history (which indirectly reveals historical MTTRs at different bid levels) does not necessarily guarantee future performance.

III. Transient Guarantees

A transient guarantee is the simple idea of providing users a probabilistic assurance on a transient server's availability, volatility, and predictability. While many variants of transient guarantee are possible, we propose a variant that provides a probabilistic guarantee by specifying a transient server's MTTR. Providing a probabilistic guarantee on the MTTR has the advantage of not imposing strict limits on a platform's freedom to revoke servers, as the MTTR need only converge to a particular value across many user requests. We define transient guarantees with respect to the MTTR because the overhead of fault-tolerance mechanisms, including the optimal checkpointing interval from Section II-B, is typically defined with respect to an MTTF (or MTTR in this case).

Of course, we could define much stronger transient guarantees to enable even higher performance. For example, EC2 introduced spot block instances in October 2015, which guarantee access to a transient server for a fixed block of time between 1 and 6 hours. Thus, EC2 promises a spot block instance will be revoked with 100% probability at the end of each block but not before. While spot (and preemptible) instances are 50-90% cheaper (in an absolute sense) than on-demand servers, spot blocks are typically only 30-45% cheaper [26]. Based on our analysis in Figure 1(a), since spot block revocations are predictable, they require less fault-tolerance overhead (and have higher performance) than spot instances, as applications need to checkpoint only once, immediately prior to the revocation. However, spot blocks also impose greater restrictions on a platform's freedom to revoke.

In effect, the more accurately a platform can predict the future supply of its idle capacity, the stronger the transient guarantees it can offer users. For example, if EC2 could perfectly predict the precise times of all future requests for on-demand and reserved servers (and when users would release them), it could simply offer spot block instances to exactly fill any idle time. As mentioned, these spot block instances are much more valuable (and cost more) than current spot instances. Of course, platforms are likely not able to predict their future demand with such precision. As a result, EC2 typically offers only a small number of spot block instances (for short time windows), likely when they are near 100% certain they will not need to revoke them within the window. However, we expect platforms are better able to forecast the statistical attributes of their future demand, e.g., its distribution, mean, and variance. For example, recent work develops prediction techniques to accurately forecast the percentage of idle server capacity, i.e., not allocated to on-demand or reserved servers, over multi-month periods for multiple production Google clusters [27]. Transient guarantees assume that platforms can accurately estimate the MTTR of transient servers based on the distribution of demand for on-demand (and reserved) servers.

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

TABLE I: Approaches to selling idle cloud capacity.

We envision platforms offering transient servers with transient guarantees for a fixed price, similar to GCE's model for preemptible instances. Note that platform's could also offer servers with transient guarantees for a variable spot price. However, fixed pricing is simpler for users to budget than EC2's variable priced bidding model because users know the actual price in advance (and not just the maximum possible price). Since EC2 charges users based on the variable spot price, and not their bid price, users do not know the cost of transient servers *a priori*. Fixed pricing also makes decision-making for users much simpler, as they do not have to monitor, analyze, and predict prices across thousands of markets to select an optimal market and determine an optimal bid.

Table I summarizes the differences between our MTTR-based transient guarantees, GCE preemptible, and EC2 spot instances in terms of their volatility, predictability, and pricing.

A. *Quantifying Transient Server Value*

An advantage of transient guarantees is that they enable users to quantify transient server value. We define a transient server's maximum value in relation to its amortized performance compared to on-demand servers after accounting for the overhead of revocations, e.g., checkpointing, migration, and recomputation. That is, if a transient server with high volatility and low predictability incurs a 25% overhead for checkpointing, migration, and recomputation, then we say its value is 25% less than an equivalent on-demand server. Since platforms generally list a fixed price per unit time for on-demand servers, we can also assign a dollar amount to a transient server's maximum value. In this case, if an on-demand server costs \$0.10 per hour, then the transient server's maximum value would be 25% less or \$0.075 per hour.

We call this the transient server's *equilibrium price*: where the price per unit of useful time (modulo overhead) between a transient server and an on-demand server is equal. Even though the equilibrium price is necessarily less than the on-demand price, rational users should never pay more than it for a transient server, as it provides no discount. Similarly, rational users should compute the discount offered by transient servers in relation to their equilibrium price and not the absolute price of on-demand servers. For example, if the transient sever above costs \$0.05 per hour, its discount per unit of useful time compared to an on-demand server is only 33%, and not 50%.

Based on our analysis, we can derive the equilibrium price for a batch application in terms of its volatility, checkpointing overhead, and the price of an equivalent on-demand server. The expected completion time $E[T_j]$ for an application j with running time T_j on a transient server is below.

$$E[T_j] = T_j + \frac{T_j}{t_{opt}}\delta + \frac{T_j}{MTTR} * \frac{t_{opt}}{2} \qquad (2)$$

Here, the first term is the application's actual running time, the second term is the additional overhead from checkpointing (at the optimal frequency), which incurs δ overhead at every checkpoint interval, and the last term is the expected recomputation overhead across all revocations (assuming the probability of revocation at any time during each interval is equal). Recall that we assume δ is a function of the transient server (and not the application), since we assume application's are well-matched to the transient server's size and use systems-level checkpointing of the entire memory footprint. Based on this analysis, a transient server has a performance that is $\frac{T_j}{E[T_j]}$ that of an on-demand server. Thus, if an equivalent on-demand server costs p_o, then the transient server's equilibrium price is:

$$p_e = p_o * \frac{T_j}{E[T_j]} \qquad (3)$$

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

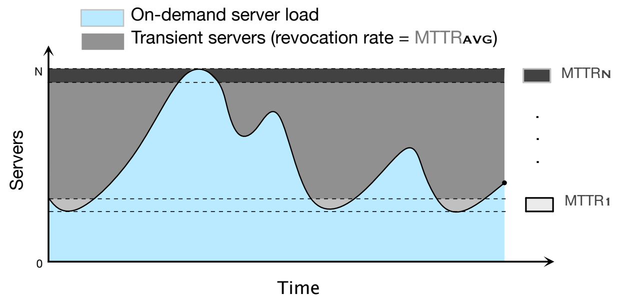


Fig. 4: Platforms may offer their idle capacity as multiple classes of transient servers with different transient guarantees.

B. Maximizing Transient Server Value

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

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

To illustrate, consider Figure 4, which depicts the idle capacity over time after servicing on-demand requests for a platform with a total capacity of N servers. This idle capacity can be offered as a single class of transient servers with $MTTR_{avg}$ based on the average revocation characteristics across all idle servers. However, notice that if we allocate on-demand requests with servers 0 to N in order, higher ranked servers experience fewer revocations than lower ranked servers. Thus, a platform could carve out the top N^{th} server to be allocated and offer it as a separate class with $MTTR_N \gg MTTR_{avg}$. Since the N^{th} server's MTTR is much longer than the average, it is more valuable to applications: it experiences fewer revocations, incurs less overhead, exhibits higher performance, and has a higher equilibrium price. In contrast, offering the N^{th} server as part of a single class with $MTTR_{avg}$ significantly undersells its true value, since its actual revocation characteristics are much better than average.

In the extreme, to maximize aggregate transient server performance and value, platforms would offer each individual server as a separate class with a unique MTTR that precisely captures its revocation characteristics. As per Equation 1, this minimizes the fault-tolerance overhead incurred by each transient server, thereby maximizing the performance and value of the entire transient server pool. Of course, this extreme is not feasible, as it would result in thousands of classes, each composed of a single server. However, offering only a single class reduces aggregate performance by treating the most available servers similarly to the least available ones. Thus, to mind this gap, we define a small number of classes such that they approach the optimal performance and value. In addition, defining a small number of transient classes, each consisting of many servers, enables providers to more accurately estimate the aggregate MTTR of each class [27].

We assume the number of idle servers ranges from $[0, N]$ at any time t. The platform then partitions N servers into k classes with each class having M_j servers, such that $\sum_{j=0}^{k} M_j = N$. As in Figure 4, we assume a strict ordering of servers with server N having the fewest revocations and server 0 having the most. Thus, higher numbered classes have higher performance than lower numbered classes. We then offer each class with a transient guarantee specifying the average MTTR for transient servers within that class. Since composing the optimal set of k classes that minimize overhead requires examining all $\binom{N}{k}$ combinations classes (with complexity $O(n^k)$), we define two simple heuristic policies.

Equal-Split Policy. Our equal-split policy naïvely divides the idle server capacity into k equal-sized classes, such that each class has the same number of N/k servers.

Greedy-Split Policy. In contrast, our greedy-split policy iteratively composes classes as follows. The policy starts with only the most available server N in the first class. The policy then proceeds iteratively by adding the next most available server $N-1$ to the first class, and then determines whether the addition of the server increases the aggregate value across all servers in the class. The aggregate value is computed as the number of servers M_j in the class multiplied by the equilibrium price p_e of servers in the class, where the equilibrium price is computed based on the average MTTR across all servers in the class.

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

C. Transient Server Pricing

We assume that transient servers are priced such that they are never idle, i.e., their price is always low enough to attract saturating demand. Note that our analysis in Section III-A only derives an equilibrium price, which represents the *maximum* amount a user should be willing to pay for a transient server.

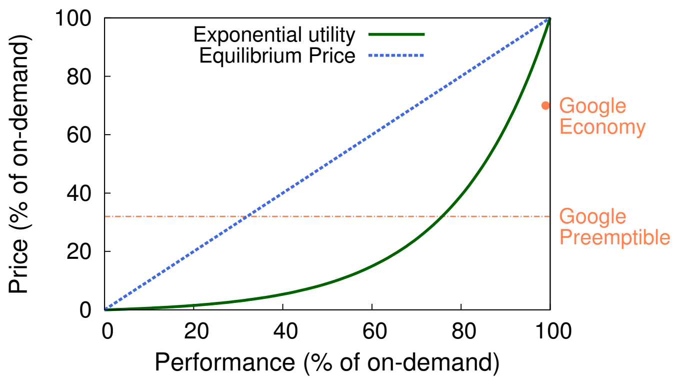


Fig. 5: Utility functions that specify an offered price for a transient server with a transient guarantee.

In practice, transient servers should be discounted from this maximum price. For example, recent work defines an economy class of on-demand servers for Google that are >98.9% available (instead of near 100%), but proposes selling them for only 70% of the on-demand price [27]. Thus, they discount these servers 30% due to only a slight reduction in availability.

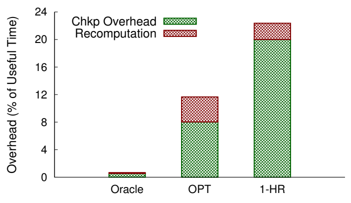
Prior work on defining utility functions for real users indicates a similar steep dropoff in utility for the initial degradation in performance [28], [29]. Thus, we adopt a similar exponential utility function for estimating the offered price of transient servers with different MTTRs. Figure 5 plots our exponential utility function, where the offered price (as a percentage of the on-demand price) on the y-axis is a function of transient server performance on the x-axis (based on the MTTR). The exponential utility function[2] captures the steep drop-off in price once servers are not 100% available. We also plot the current price-points for the Google economy class described above and GCE preemptible instances, which cost $\sim$70% of the on-demand price for all servers. Finally, we plot the equilibrium price, which is simply the line y=x. The difference between our utility function and the equilibrium price is the normalized discount from using transient servers.

We use the utility function above in Section V to estimate the potential revenue from offering transient guarantees. Note that, to verify transient guarantees, large-scale users can average their performance across a large number of requests. Ensuring small-scale users can verify transient guarantees poses a more challenging problem. While such verification is outside the scope of this paper, crowd-sourced techniques [30] are a promising direction.

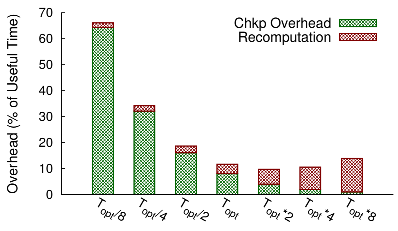
IV. Implementation

We implemented a cluster simulator in python to evaluate the performance improvements that transient guarantees offer. The simulator takes a fixed server capacity as input (where each server has a specified memory size), as well as a trace of requests for on-demand servers. Each request specifies a job to run that includes the number of servers the job requires,

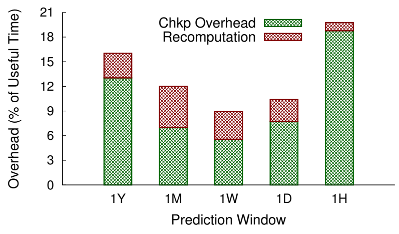
[2]Exponential utility is derived from the function $y = 101^{(x/100)} - 1$.



(a) Baseline Policies



(b) Checkpoint Interval



(c) Inaccurate Predictions

Fig. 6: Overhead due to different checkpointing policies for the `m1.large` spot market in the U.S. East region of EC2.

job submission time, and job duration. We assume any excess capacity is then allocated to the pool of transient servers. Each transient server incurs an overhead by periodically checkpointing its memory state. We assume that transient guarantees specify an MTTR, which transient servers use to compute the optimal checkpointing interval (based on t_{opt}). Our simulator implements the equal-split and greedy-split algorithms.

The simulator is designed to operate on publicly-available job traces from a production Google cluster [31]. The trace contains job characteristics, server configurations, and scheduling decisions on a cluster of 12.5k servers over a 29-day period. We make a few simplifying assumptions using this job trace in our evaluation. First, we normalize the heterogeneous servers in the trace based on the smallest server type, and assume a cluster with homogeneous servers. If a server runs multiple jobs concurrently, we assume it runs multiple VMs. We also rank cluster nodes from 1 to N, and schedule jobs in rank order (as opposed to following Google's scheduling decisions), such that if k jobs are active, they occupy servers 1 to k. These assumptions enable the simulator to avoid trace-specific scheduling, bin-packing, and cluster management decisions that are not central to evaluating transient guarantees.

V. Evaluation

Our evaluation first examines the overhead and performance degradation due to revocation for EC2 spot instances based on historical spot prices. We then analyze the demand pattern of a production Google cluster trace [31] over a month-long period to quantify the benefits of using transient guarantees.

A. EC2 Spot Instance Performance

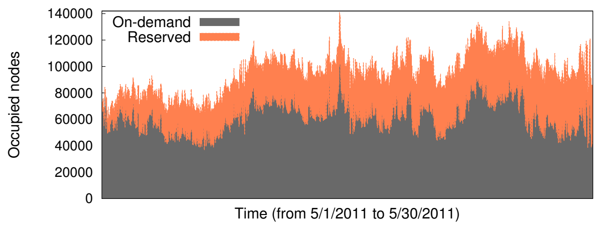
We evaluate spot instance performance for a representative EC2 spot market—the `m1.large` instance type running Linux in one availability zone of the U.S. East region over 2014. Note that we focus on EC2, since GCE offers no information on the revocation characteristics of preemptible instances. The `m1.large` market represents one of the most popular configurations of the most popular instance types in the most popular region of EC2. In this analysis, we assume a bid equal to the on-demand price, as in prior work [15], [17], since users can nearly always switch to using on-demand servers if their spot instances are revoked. Based on spot price data from 2014, we observe 555 revocations over the year with an MTTR of $\sim$15 hours. We assume a modest-sized memory footprint of 16GB, which takes $\sim$10 minutes to checkpoint and restore based on our benchmarks using EBS magnetic disks.

Figure 6a plots the overhead of spot instances in the `m1.large` market for three different fault-tolerance policies: an oracle policy that minimizes overhead by checkpointing immediately before each revocation, a periodic policy that checkpoints at the optimal (OPT) periodic interval [7] ($t_{opt} \sim$ 2 hours in this case) assuming the MTTR is known, and a static policy that checkpoints once each hour. The latter policy is proposed in prior work, since EC2 bills on an hourly basis [32]. The figure shows that the static per-hour checkpointing consumes 24% of the useful server-time, and the optimal periodic policy consumes 12% of the useful server-time. While the oracle consumes less than 1% overhead, it is not viable in practice as future demand is not precisely known.

Figure 6b then shows the impact on overhead of incorrectly setting the checkpointing frequency when computing the optimal periodic checkpointing interval from Figure 6a. In this case, the optimal checkpointing frequency is near $2{*}t_{opt}$, since the optimal formula is only a first-order approximation and incorrectly assumes revocation interarrival times are Poisson distributed. However, recall that with EC2, users actually do not know future revocation characteristics. The graph shows that selecting a checkpointing frequency too short can result in significant additional overhead that further reduces performance. Finally, Figure 6c shows the impact of mispredicting the revocation rate on overhead. In this case, we use the optimal periodic checkpointing interval, but where we compute the MTTR based on spot price history over different size past windows, e.g., the last hour, day, week, month, and year. The graph shows that overhead varies widely (from 9% to 21%) depending on the prediction window we select.

Our analysis demonstrates the overheads due to revocation in EC2's spot market are already non-trivial. However, while the overhead generally varies from 10-20%, spot instance prices are 70-90% less than on-demand servers. As a result, spot instance prices are currently well below their equilibrium price. However, volatility in EC2's spot market is unbounded. As more users exploit the arbitrage opportunities available in the spot market, we expect the average spot price to rise and to become more volatile and less predictable. This volatility will increase the overhead of spot instances and decrease their performance. Transient guarantees address this issue.

Result. *Volatility in EC2's spot market, which is unbounded*



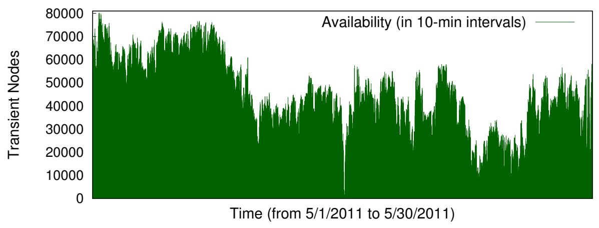
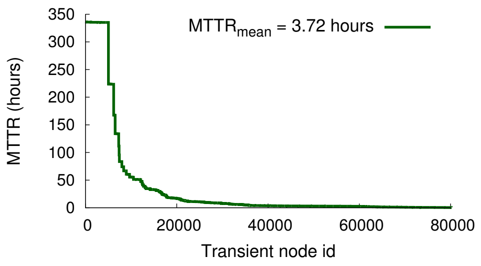
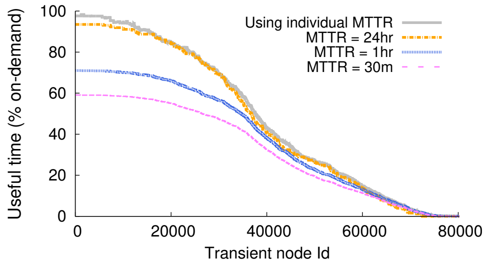


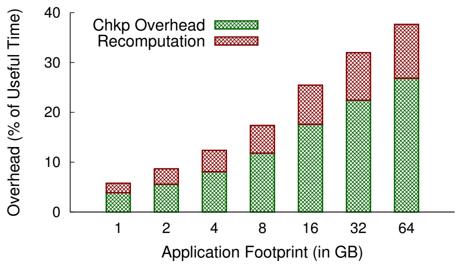
Fig. 7: Workload for a production Google cluster [31] and the resulting availability of transient servers.



(a) MTTR Distribution



(b) Performance



(c) Impact of Application Footprint

Fig. 8: Revocation characteristics (a) of transient servers in 7. Performance of individual transient servers (b) at 16GB memory footprint, and the impact on aggregate performance (c) when application footprint varies.

and unpredictable, already reduces the performance of spot instances by 9-24% relative to on-demand servers.

B. Transient Guarantee Performance

Using the Google cluster traces [31], Figure 7(a) plots the server allocation pattern for two classes of jobs – high-priority (akin to requests for reserved instances) and low-priority (akin to requests for on-demand instances). The peak (normalized) server capacity required for executing both job classes is ∼141k VMs. We assume the cluster is sized precisely for its peak capacity (such that at some time t all servers are necessary to satisfy demand). Thus, the white space at the top of the graph indicates the varying amount of idle server capacity that is available to offer as transient servers.

Figure 7(b) then plots the idle server capacity over time, where only those servers that are unused for at least 10 minutes are considered. Since it takes a few minutes to allocate a new VM, we do not consider <10 minutes of idle time useful. The availability of idle capacity ranges from 0 to ∼80k VMs, where we assume each server has 16GB of memory.

Figure 8a then shows the mean-time-to-revocation (MTTR) for each transient server in the cluster. While the average MTTR across all transient servers is 3.72 hours, we note that the top 10% of servers has an MTTR >84 hours and the top 50% has an MTTR >17 hours. Thus, as discussed earlier, a large fraction of transient servers experience much longer periods of availability than reflected in the average MTTR.

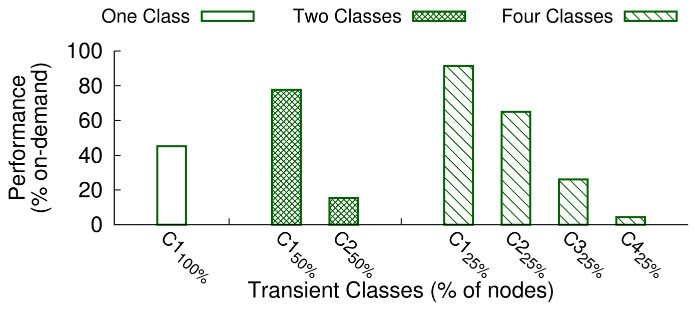
Next, we derive the maximum amount of useful server-time (modulo fault-tolerance overhead) for each transient server, assuming that we offer each server with a transient guarantee based on its own unique MTTR. We also plot the useful server-time assuming an MTTR of 24 hours, 1 hour and 30 minutes across all transient servers. Figure 8b shows that, in this case, using a MTTR of 24 hours achieves near the optimal useful server-time, while using a MTTR of 30 mins reduces the useful server-time by ∼40% across all servers in the cluster. Since only 7% of transient servers have an MTTR in the range of 30 minutes, this results in excessive checkpointing overhead for the vast majority of servers.

Finally, Figure 8c shows that as server memory footprints increase, they spend an increasing amount of time on fault-tolerance overhead. However, we observe that this increase is sublinear based on Equation 2), as the x-axis is on a log scale. As a result, halving the application size from, say, 16GB to 8GB, only decreases the overhead from ∼25% to ∼18%.

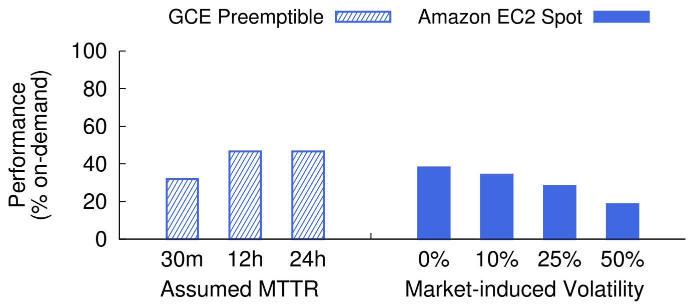
Result: *Since idle capacity varies over time, transient servers exhibit a wide range of characteristics. Checkpointing transient servers based on incorrect revocation characteristics results in significant performance losses (up to ∼40%), especially for the least volatile (and most valuable) servers.*

Figure 9a next quantifies the benefit of partitioning transient servers into multiple classes with different transient guarantees. This graph employs the equal-split policy to partition transient servers into 1, 2, and 4 classes. In each case, the y-axis quantifies the average useful time of a server in each class (modulo checkpointing overhead). The graph demonstrates how separating transient servers into different classes enables platforms to offer differentiated quality-of-service for different transient servers. We see that, as we offer more transient classes, the increase in the performance of higher classes is significantly more than the decrease in the performance of lower classes, thereby increasing the overall performance of the cluster. For example, moving from a single class configuration to a two class configuration results in an overall decrease in fault-tolerance overhead across all transient servers of 13.5%. This reduction in overhead is due to more accurately specifying revocation characteristics in multiple classes.

Figure 9b then compares the performance of transient



(a) Transient Guarantees



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

Fig. 9: Performance of transient servers in the Google job trace when using transient guarantees with different partitionings (a) and performance when allocating using the approaches taken with GCE and EC2 (b). Performance is in terms of the percentage of useful time (modulo overhead) compared to an equivalent on-demand server.

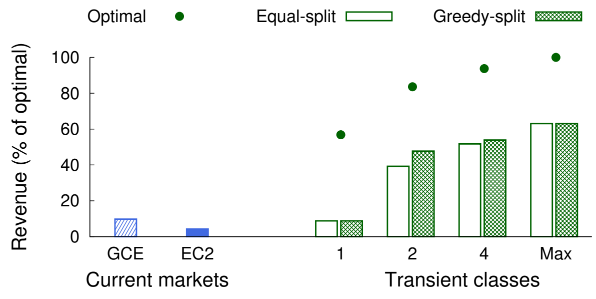


Fig. 10: Revenue comparison when selling transient servers for GCE, EC2, and using transient guarantees.

guarantees with the current approaches used by GCE and EC2. Since GCE does not reveal any information about preemptible instances, we consider three distinct fixed-interval checkpointing policies that assumes a MTTR of 30 minutes, 12 hours, and 24 hours. For EC2, we use the commonly employed one hour checkpointing strategy [32]. Since EC2's spot market is strictly more volatile than our approach, we add differing levels of market-induced volatility to the Google job trace for EC2. In this case, adding X% volatility means that we inject X% more revocations than are dictated solely by variations in the trace's demand for servers. These additional X% revocations imitate users outbidding each other for transient serverss. The graph shows that the performance of GCE and EC2 is less than that with transient guarantees. GCE's performance (assuming a 24 hour MTTR) is near that of offering a single class with transient guarantees, but has 13% higher overhead than offering two separate classes. EC2 also performs worse than a single class with transient guarantees, largely because of the additional market-induced volatility. For example with an additional 25% volatility, its performance is 28% less than using transient guarantees with a single class.

Finally, we evaluate the potential increase in revenue from offering multiple classes of servers with transient guarantees compared to GCE and EC2. For this graph, we use the exponential utility function from Figure 5 to assign prices to transient servers based on their performance. Figure 10 then shows the aggregate revenue from selling GCE preemptible instances, EC2 spot instances, and using transient guarantees at these prices. The dot represents the optimal revenue if transient servers were sold at their equilibrium price. In all cases, we assume saturating demand, such that all transient servers are sold. The y-axis quantifies the revenue as a % of the maximum, where every transient server is priced at its equilibrium price, for each approach on the x-axis. The maximum number of transient classes represents the optimal where each transient server has its own class and revocation characteristics. For GCE and EC2 we select the top performing configuration from Figure 9. The figure shows the potential revenue of offering multiple classes of transient servers.

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

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

VI. Related Work

Prior work focuses on optimizing existing offerings of transient servers from a user's perspective. Much of this work exploits particular details of EC2's spot market. For example, there is substantial prior work on analyzing EC2 spot price characteristics [8], [33], designing optimal bidding policies [12], [13], [10], and modifying particular applications to gracefully handle transient servers using fault-tolerance mechanisms, such as checkpointing [32], [17]. Our work differs from this work in that we take a platform's perspective, and then analyze how best to provide transient servers to users to maximize their performance, while still allowing platforms the freedom to revoke transient servers when necessary.

Related work that takes a similar platform-centric perspective proposes offering a new economy class of on-demand servers that have slightly lower availability (>98.9%) [27]. The work analyzes data from multiple Google production clusters and shows that a large fraction of servers (6.7-17.3%) are idle with high probability for long multi-month time periods. Our work generalizes and expands upon this idea by defining the concept of a transient guarantee, and then showing how to partition idle capacity into an arbitrary number of transient server classes to maximize the performance of transient servers. Importantly, we also identify the relationship between a transient server's performance and its volatility and predictability, and define its equilibrium price to capture its value relative to an on-demand server.

Our analysis assumes transient servers use checkpointing to handle revocations. While the vast majority of applications run on transient servers are batch applications that store in-memory state, some applications that are stateless and need not checkpoint in-memory state. In this case, the equilibrium price of the transient server would equal the on-demand price. Recent work takes advantage of particular properties of EC2 spot instances, such as its two minute revocation warning, to asynchronously copy memory state to a backup server [15]. The overhead of these approaches is not only dependent on volatility and predictability, but also the length of the warning. We leave a consideration of the overhead (and value) of the warning time to interactive applications as future work.

VII. Conclusion

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

Acknowledgements. We would like to thank our shepherd Rich Wolski and the anonymous reviewers for their feedback, which improved the quality of this paper. This work is supported by a grant from the NSF (CNS-#1422245) and a Google Faculty Research Award.

References

[1] L. Columbus, "Roundup of Cloud Computing Forecasts and Market Estimates Q3 Update," Forbes, September 27th 2015.

[2] T. Qureshi, "Cost-effective Batch Processing with Amazon EC2 Spot," AWS Compute Blog, August 7th 2015.

[3] T. Trader, "Amazon Web Services Spotlights HPC Options," HPCWire, August 11th 2015.

[4] D. Pellerin, D. Ballantyne, and A. Boeglin, "An Introduction to High Performance Computing on AWS," Amazon Whitepaper, September 1 2015.

[5] Scientific, "Scientific Computing Using Spot Instances," https://aws.amazon.com/ec2/spot/spot-and-science/, August 2016.

[6] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large scale Cluster Management at Google with Borg," in *SoCC*, November 2015.

[7] J. Daly, "A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps," in *Future Generation Computer Systems*, vol. 22, no. 3, 2006.

[8] O. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 Spot Instance Pricing," in *ACM Transactions on Economics and Computation*, vol. 1, no. 3, September 2013.

[9] X. Ouyang, D. Irwin, and P. Shenoy, "SpotLight: An Information Service for the Cloud," in *ICDCS*, June 2016.

[10] L. Zheng, C. Joe-Wong, C. Tan, M. Chiang, and X. Wang, "How to Bid the Cloud," in *SIGCOMM*, August 2015.

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

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

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

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

[15] P. Sharma, S. Lee, T. Guo, , D. Irwin, and P. Shenoy, "SpotCheck: Designing a Derivative Cloud on the Spot Market," in *Eurosys*, April 2015.

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

[17] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy, "SpotOn: A Batch Computing Service for the Spot Market," in *SoCC*, August 2015.

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

[19] P. Sharma, D. Irwin, and P. Shenoy, "How Not to Bid the Cloud," in *HotCloud*, June 2016.

[20] G. Banga, P. Druschel, and J. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *OSDI*, February 1999.

[21] D. Williams, H. Jamjoom, and H. Weatherspoon, "The Xen-Blanket: Virtualize Once, Run Everywhere," in *EuroSys*, 2012.

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

[23] F. Lardinois, "Spotinst, which helps you buy AWS spot instances, raises 2M Series A," TechCrunch, March 8th 2016.

[24] J. Novet, "Amazon pays 20m-50m for ClusterK, the startup that can run apps on AWS at 10% of the regular price," VentureBeat, April 29th 2015.

[25] A. Barrett, "The Need for Speed: This Week on Google Platform," Google Cloud Platform Blog, June 10th 2016.

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

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

[28] C. Lee and A. Snavely, "Precise and Realistic Utility Functions for User Centric Performance Analysis of Schedulers," in *HPDC*, June 2007.

[29] J. Wilkes, "Utility Functions, Prices, and Negotiation," Hewlett-Packard, Technical Report, July 2008.

[30] K. Walsh and E. G. Sirer, "Experience with an Object Reputation System for Peer-to-Peer Filesharing," in *NSDI*, May 2006.

[31] C. Reiss, J. Wilkes, and J. Hellerstein, "Google Cluster-usage Traces: Format + Schema," Google Inc., Technical Report, Nov 2011.

[32] W. Voorsluys and R. Buyya, "Reliable Provisioning of Spot Instances for Compute-Intensive Applications," in *Advanced Information Networking and Applications (AINA)*, 2012.

[33] B. Javadi, R. Thulasiram, and R. Buyya, "Statistical Modeling of Spot Instance Prices in Public Cloud Environments," in *UCC*, December 2011.