Optimizing the Cost of Executing Mixed Interactive and Batch Workloads on Transient VMs

PRADEEP AMBATI, University of Massachusetts Amherst
DAVID IRWIN, University of Massachusetts Amherst

Container Orchestration Platforms (COPs), such as Kubernetes, are increasingly used to manage large-scale clusters by automating resource allocation between applications encapsulated in containers. Increasingly, the resources underlying COPs are virtual machines (VMs) dynamically acquired from cloud platforms. COPs may choose from many different types of VMs offered by cloud platforms, which differ in their cost, performance, and availability. In particular, while *transient VMs* cost significantly less than on-demand VMs, platforms may revoke them at any time, causing them to become unavailable. While transient VMs' price is attractive, their unreliability is a problem for COPs designed to support mixed workloads composed of, not only delay-tolerant batch jobs, but also long-lived interactive services with high availability requirements.

To address the problem, we design TR-Kubernetes, a COP that optimizes the cost of executing mixed interactive and batch workloads on cloud platforms using transient VMs. To do so, TR-Kubernetes enforces arbitrary availability requirements specified by interactive services despite transient VM unavailability by acquiring *many more* transient VMs than necessary most of the time, which it then leverages to opportunistically execute batch jobs when excess resources are available. When cloud platforms revoke transient VMs, TR-Kubernetes relies on existing Kubernetes functions to internally revoke resources from batch jobs to maintain interactive services' availability requirements. We show that TR-Kubernetes requires minimal extensions to Kubernetes, and is capable of lowering the cost (by 53%) and improving the availability (99.999%) of a representative interactive/batch workload on Amazon EC2 when using transient compared to on-demand VMs.

ACM Reference Format:
Pradeep Ambati and David Irwin. 2019. Optimizing the Cost of Executing Mixed Interactive and Batch Workloads on Transient VMs. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 2, Article 28 (June 2019), 24 pages. https://doi.org/10.1145/3326143

1 INTRODUCTION

Container Orchestration Platforms (COPs), such as Kubernetes [9], Borg [22], Mesos [13], Docker Swarm [4], and others, have evolved into de facto cluster "operating systems" by automating the deployment of distributed applications encapsulated in containers, and managing the allocation of resources between them. COPs manage clusters of tens of thousands of machines, and serve as the primary interface users interact with to harness cluster resources. Thus, COPs must support the availability and performance requirements of a wide range of applications, including long-lived interactive services and non-interactive batch jobs, while also maintaining high cluster utilization. To do so, COPs include a rich set of functions for i) handling failures that occur at large scales, ii) spawning replacement containers, iii) load balancing network traffic, iv) provisioning new

Authors' addresses: Pradeep Ambati, University of Massachusetts Amherst, lambati@umass.edu; David Irwin, University of Massachusetts Amherst, irwin@ecs.umass.edu.

2476-1249/2019/6-ART28 $15.00
https://doi.org/10.1145/3326143

containers under constraint, i.e., autoscaling, and v) prioritizing resource requests (by revoking resources from lower priority applications and allocating them to higher priority ones).

COPs were originally developed for managing a mostly static set of dedicated physical machines in data centers. However, increasingly, the resources that underly COPs are virtual machines (VMs) dynamically acquired from Infrastructure-as-a-Service (IaaS) cloud platforms. These platforms offer many types of VMs under a variety of different contract terms, which differ in their cost, performance, and availability. In particular, transient VMs are an increasingly popular VM type, since they typically cost 50-90% less than on-demand VMs. However, in exchange for their low cost, cloud platforms reserve the right to reclaim transient VMs at any time to satisfy higher priority requests. Thus, while transient VMs' low price is attractive, their unreliability makes them unsuitable for COPs that must support long-lived interactive services with high availability requirements. As a result, prior work focuses primarily on optimizing only batch workloads for transient VMs, e.g., by tuning fault-tolerance mechanisms [11, 12, 18, 21, 25, 26].

To address the problem, we design TR-Kubernetes, a transient-aware COP that supports both batch jobs and interactive services with high availability requirements at low cost using transient VMs. To do so, TR-Kubernetes enables interactive services to explicitly specify their capacity availability requirement. For example, a distributed web service may specify that it needs the equivalent capacity of 500 `m4.large` servers from Amazon's Elastic Compute Cloud (EC2) with 5 nines of availability (99.999%). TR-Kubernetes's provisioning policy then selects a mix of different transient VMs, from among the hundreds offered by cloud platforms, that satisfies the capacity availability requirement with high probability. As we show, to enforce high availability using unreliable transient VMs, TR-Kubernetes must acquire *many more* transient VMs than necessary most of the time. Current transient VM prices are low enough that acquiring more transient VMs than necessary—in some cases many more—is still *much cheaper* than using on-demand VMs to satisfy the same capacity availability requirement. Thus, TR-Kubernetes often has much more capacity provisioned than its interactive services require. TR-Kubernetes automatically leverages this excess capacity to opportunistically execute delay-tolerant batch jobs at no additional cost. If interactive services ever require additional resources, due to increased demand or transient VM revocations, TR-Kubernetes leverages internal functions COPs already provide to revoke resources from the batch jobs and allocate them to the interactive services. This re-provisioning of containerized resources within a COP has much lower overhead than re-provisioning cloud VMs.

Maintaining excess capacity for interactive services also offers the additional benefits below.

- **Higher Availability**. TR-Kubernetes's excess capacity enables higher availability than is typically offered for on-demand cloud VMs. For example, EC2's SLA for on-demand VMs only provides refunds if availability is less than 99.99%, while TR-Kubernetes can satisfy arbitrarily high availability requirements, e.g., 5 nines or higher, albeit at an increasing cost.
- **Greater Responsiveness**. Interactive services often experience sudden workload spikes that require quickly provisioning additional VMs to satisfy the increased demand. However, since detecting a spike and provisioning additional cloud VMs may take on the order of minutes, while response time Service Level Objectives (SLOs) are generally on the order of milliseconds, sudden spikes may cause SLO violations in the interim. In this case, the excess capacity provides headroom to satisfy unexpected workload spikes without requesting new cloud VMs.

The advantages of maintaining excess capacity for interactive services are a complex function of a COP's interactive-to-batch workload mix, availability requirements, and transient VM prices. For example, satisfying higher capacity availability requirements requires maintaining more excess resource capacity at a greater cost. However, if these resources can effectively be used by batch jobs, then TR-Kubernetes remains cost-efficient. Conversely, a lower capacity availability requirement

requires maintaining fewer excess resources at a much lower cost, but results in fewer excess resources available to run batch jobs, which either increases job completion time (by requiring these jobs to wait until resources are available) or increases costs (by requiring the COP to provision new cloud VMs to run the jobs). We evaluate these tradeoffs on EC2 using a public workload trace [23].

Interestingly, the reason COPs include internal support for revocations, and thus transience, is the same reason cloud platforms offer transient VMs: to increase overall cluster utilization by enabling lower-priority (and delay-tolerant) batch jobs to run whenever resources are available, while also supporting higher-priority interactive tasks with strict performance and availability requirements. Thus, existing integrations of COPs with cloud platforms typically run interactive services on high-cost on-demand VMs, and run delay-tolerant batch jobs on low-cost transient VMs. In contrast, our hypothesis is that TR-Kubernetes can enable higher availability, better performance, and lower costs by running mixed interactive and batch workloads entirely on transient VMs. In evaluating our hypothesis, we make the following contributions.

TR-Kubernetes Design. We outline TR-Kubernetes's simple design, which mostly relies on existing COP mechanisms, e.g., for handling revocations, spawning replacement containers, autoscaling, etc., and existing cloud APIs, e.g., for allocating and replacing transient VMs. TR-Kubernetes's design then builds on these basic mechanisms by introducing i) an intelligent provisioning policy that selects transient VMs to ensure a specified capacity availability at a low price and ii) a background daemon that coordinates the graceful revocation of transient VMs.

Provisioning Policy. We present TR-Kubernetes's provisioning policy that selects unreliable transient VMs from among the hundreds offered by cloud platforms based on cost and availability estimates. As we show, our policy achieves a cost savings near that of the optimal cost-minimizing policy, while achieving an availability near that of the optimal availability-maximizing policy. Our comparison is based on an analysis of the price and availability of 350 spot VMs across all EC2 regions and availability zones (AZs), i.e., data centers, in the U.S. over a 3 month period.

Implementation and Evaluation. We implement and evaluate TR-Kubernetes's provisioning policy by minimally extending Kubernetes for AWS [5]. We show that transient VM revocations do not significantly affect interactive service performance and reliability. We also implement a trace-driven simulator to quantify TR-Kubernetes's benefits over long periods using a publicly-available workload trace [23]. We show that if interactive services comprise 50% of the workload and require 99.999% availability, then TR-Kubernetes lowers the cost of hosting these interactive services by 53% and executing the batch jobs by 30% compared to using on-demand VMs.

2 BACKGROUND

TR-Kubernetes is a Container Orchestration Platform (COP) that uses unreliable transient cloud VMs to execute mixed batch/interactive workloads at low cost. This section provides background on COPs, transient VMs, and our underlying assumptions.

2.1 Container Orchestration Platforms

There are many publicly-available COPs that offer similar functionality and support diverse workloads on large, mixed-use clusters, including Kubernetes [9], Mesos (with Marathon) [13], and Docker Swarm [4]. These platforms not only manage the allocation of cluster resources between distributed applications encapsulated in containers, but also provide a rich set of functions for supporting distributed applications and tightly integrate with IaaS cloud platforms. In addition, IaaS platforms now natively provide container orchestration platforms hosted on their VMs, such as Amazon's EC2 Container Service for Kubernetes (EKS) [3], Google's Kubernetes Engine (GKE) [2], and Azure Kubernetes Service (AKS) [1]. These native platforms are largely the same as the open-source platforms, but are configured automatically by the cloud provider and support elastic

autoscaling, i.e., by automatically allocating and releasing VMs as necessary. While our prototype extends Kubernetes on AWS, our work is applicable to any COP on any cloud platform.

A key assumption COPs make is that distributed applications that run on them can handle i) the failure or revocation of containers, and ii) the allocation of new replacement containers. This assumption enables COPs i) to scale up to many thousands of servers, where some failures are inevitable, by automatically replacing failed containers with new containers, and ii) to freely revoke containers from applications without affecting application correctness. COPs generally support two broad classes of applications—interactive services and batch jobs [22]. Interactive services must respond to user requests in real-time at millisecond-scale latencies, while batch jobs are tolerant to delays and thus typically run in the background without strict deadlines. Internal support for revocations enables COPs to allocate available resources to batch jobs, while also enabling them to revoke these resources and re-allocate them to interactive services if their demand spikes.

Thus, distributed applications that run on COPs are designed to handle container failures, revocations, and re-allocations. Interactive services are typically designed to be stateless and often leverage load balancers natively built into COPs that spread requests among services' active containers. These load balancers automatically update the active set of containers as VM failures and revocations occur. In contrast, batch jobs either restart from the beginning, or from the most recent checkpoint, on a VM revocation. Many distributed big data frameworks, such as Spark [27], Naiad [15], TensorFlow [7], etc., that cache large datasets in volatile memory include built-in functions for checkpointing job state. Optimizing this checkpointing based on job characteristics and VM revocation rates is the subject of active research [17, 21, 25, 26]. As discussed in §6, batch jobs that run on TR-Kubernetes may leverage such checkpointing policies.

Since TR-Kubernetes extends an existing COP in Kubernetes, it makes the same assumptions as above. In particular, we assume distributed applications are designed to handle container failures and revocations. We also assume i) interactive services are stateless and leverage Kubernetes's internal load balancer to spread their load across the active set of containers, which is dynamically updated as revocations occur, and ii) stateful batch jobs must re-start, either from the beginning or from their last checkpoint, on each revocation. Note that, as with Kubernetes, TR-Kubernetes does not natively support stateful interactive services that cannot gracefully handle revocations. In §6, we discuss related work that TR-Kubernetes could incorporate to support such services.

2.2 Transient Cloud VMs

Transient VMs are available temporarily for an uncertain amount time, as platforms may revoke them at any time with little warning. As discussed above, since COPs support revocations, they implicitly allocate transient VMs to lower-priority jobs, which are generally batch jobs. IaaS cloud platforms offer transient VMs to customers for similar reasons as COPs: they enable cloud platforms to earn revenue from their idle capacity, while retaining the flexibility to reclaim it to satisfy requests for higher-priority on-demand VMs, which they try not to revoke. Each of the major cloud platforms—Google Cloud Platform (GCP), Microsoft Azure, and Amazon EC2—now offer transient VMs. GCP and Azure offer transient VMs for a fixed price, called preemptible VMs and low-priority batch VMs, respectively, while EC2 offers spot instances for a variable spot price.

TR-Kubernetes uses estimates of each transient VM's cost, availability, and revocation frequency to select VMs that satisfy the capacity availability requirements of an interactive service. Note that the unavailability of transient VMs is dictated by the presence of high-priority jobs, which causes platforms to preempt VMs executing low-priority jobs, and not hardware failures. Prior work on optimizing batch jobs for transient VMs derives availability estimates from historical usage, internal logs, or public price traces. In particular, EC2 publicly releases price traces for its transient spot VMs, which enables users to infer their unavailability, since it correlates with high

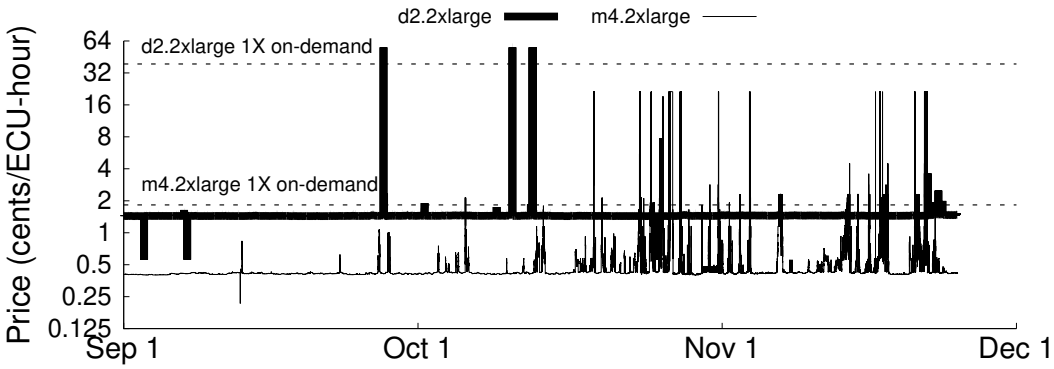


Fig. 1. Comparison of `d2.2xlarge` and `m4.2xlarge` spot prices in *us-west-1c* from 2017/9 - 2017/12.

prices. For example, Figure 1 shows a spot price trace for two transient VMs from September to December 2017, which includes the latest change in EC2's spot pricing algorithm to smooth price fluctuations. The dotted lines indicate each VM's on-demand price: the transient VM is unavailable when the price is above the dotted line, and a revocation occurs whenever the spot price crosses the dotted line. Importantly, each VM type exhibits its own availability characteristics. Cloud platforms include hundreds of VM types in each region. In EC2, `us-east-1` alone has 375 types of VMs.

To simplify using transient VMs, cloud platforms include tools to automate their allocation and replacement. For example, EC2 offers Spot Fleet, which enables users to specify a policy to automate the replacement of revoked transient VMs (either to minimize cost or maximize diversity). Unlike TR-Kubernetes, Spot Fleet's policy is agnostic of application availability and capacity requirements. We compare TR-Kubernetes's provisioning policy with Spot Fleet in the next section and in §5. In addition, EC2 users can submit persistent requests for transient VMs which direct EC2 to automatically re-acquire a revoked VM once it becomes available again after a revocation. Persistent requests simplify TR-Kubernetes's implementation by requiring its provisioning policy to specify only the initial VM selection, and automating all subsequent VM allocations and revocations. Persistent requests also enable spot hibernation, a new feature where EC2 automatically checkpoints VM memory state prior to a revocation and restores it on re-allocation. Note that spot hibernation only works when resuming on the same transient VM, as with persistent requests. We evaluate TR-Kubernetes with and without automated checkpointing via spot hibernation in §5.

3 DESIGN

We first outline TR-Kubernetes's extensions to Kubernetes and then present its transient-aware provisioning policy, which enables interactive services to specify a capacity availability requirement.

3.1 TR-Kubernetes Design

TR-Kubernetes's design relies heavily on existing functions built into Kubernetes, as well as other COPs. The primary difference is that TR-Kubernetes enables users to specify a *capacity availability requirement* for an aggregate amount of computational capacity in their service description for an interactive service. Figure 2 depicts TR-Kubernetes's extensions to Kubernetes. These include an offline tool that runs TR-Kubernetes's *provisioning algorithm* to generate service descriptions, which specify the transient VMs necessary to satisfy the capacity availability requirement. This service description is then submitted along with the job via the Kubernetes command-line tool. Since our prototype runs on EC2, users specify capacity in terms of EC2 Compute Units (ECUs), which is a relative measure of a VM's integer processing power [8]. As an example, an interactive

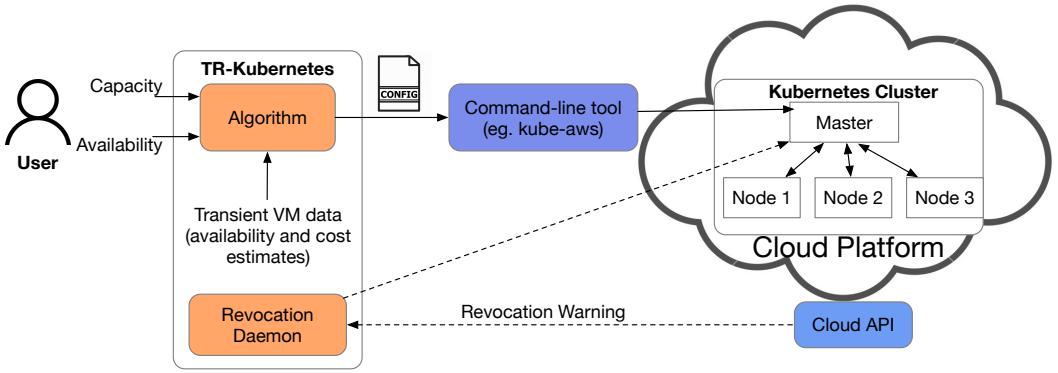


Fig. 2. A depiction of TR-Kubernetes architecture.

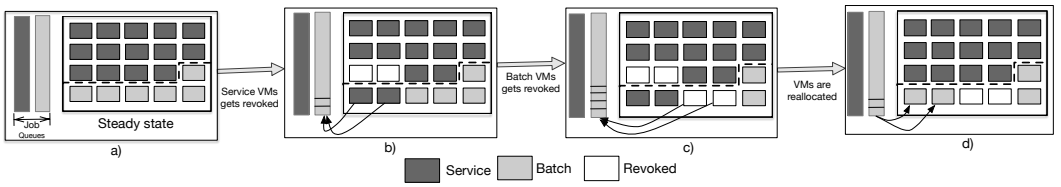


Fig. 3. Depiction of states, events, and state transitions when responding to transient VM revocation and re-allocation.

service might specify that it requires 10,000 ECUs with an availability of 99.999%. TR-Kubernetes's provisioning policy (§3.2) then determines which transient VMs to request and how many based on availability estimates, such that they satisfy the capacity availability requirement at the lowest cost.

Since Kubernetes is already widely used by major enterprises, a key goal of TR-Kubernetes is to minimally alter its operational model by only requiring users to additionally specify a service's target capacity availability. TR-Kubernetes otherwise preserves Kubernetes's existing approach for specifying service descriptions, which requires services to specify their resource requirements, and not their performance requirements, e.g., for average or tail latency. However, we do evaluate TR-Kubernetes's effect on performance and tail latency in §5. Once transient VMs are allocated based on a service description, TR-Kubernetes relies on functions already built into Kubernetes on AWS, an open source implementation of Kubernetes designed to run on VMs dynamically acquired from EC2. Kubernetes on AWS supports EC2 services, such as Auto Scaling and Spot Fleet, designed to automate resource allocation and revocation. As a result, Kubernetes on AWS can handle revocations by simply detecting them as failed VMs and removing them from the cluster.

TR-Kubernetes also employs an external *revocation daemon* that interacts with cloud APIs and Kubernetes to detect imminent revocations and proactively remove revoke VMs to minimize failed requests. In addition, if a revoked VM reduces the capacity of an interactive service below the requirement in its service description, Kubernetes's replication controller can automatically spawn replica container *pods*, i.e., co-located groups of containers, on the remaining VMs to replace the resources on the revoked VM. If necessary, to ensure the required capacity, Kubernetes will revoke resources internally from lower-priority batch jobs. Of course, if there are no lower-priority resources left to revoke, TR-Kubernetes may not satisfy a service's capacity requirement for brief periods, which increases its unavailability. However, our provisioning algorithm described in §3.2 is designed to ensure these periods preserve the capacity availability requirement over the long term.

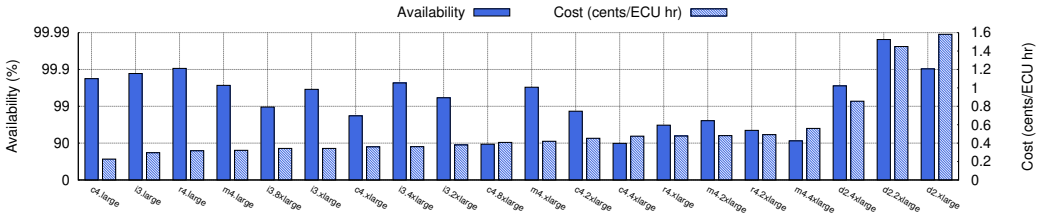


Fig. 4. Comparison of average transient VM cost and availability in *us-west-1c* from 2017/9-12.

Kubernetes includes a built-in load balancer, which distributes interactive service requests across the active container pods, automatically removes pods on failed/revoked VMs from the active set, and automatically adds newly scheduled container pods to the active set. As a result, Kubernetes's internal functions already largely mask failures/revocations from the external users or applications of interactive services. If EC2 revokes a VM executing a batch job, then Kubernetes re-schedules and queues the revoked batch job. TR-Kubernetes can either queue the job to run on the next available resource (of those remaining), or, if no resources are available, provision a new transient or on-demand VM to execute the job immediately. Kubernetes on AWS natively supports the reactive autoscaling above for interactive services or batch jobs to replace failed/revoked VMs with on-demand cloud VMs, and can also be configured to use transient VMs. Likewise, Kubernetes on AWS is also able to re-incorporate re-allocated transient VMs (from a persistent request) that were previously revoked. In this case, when the transient VM starts again (possibly with a new IP address and DNS name), as part of its boot-up procedure, it contacts the Kubernetes master and re-registers itself as a new worker.

To illustrate, Figure 3 depicts various states, events, and state transitions that may occur with TR-Kubernetes. The left figure (a) represents a steady state with 20 transient VMs such that the interactive service capacity requirement is only 14 transient VMs. If any VMs from the interactive service are revoked, Kubernetes internally revokes resources from batch tasks and allocates them to the interactive service to maintain its capacity requirement, and places the batch jobs in the queue (b). If any VMs running batch jobs are revoked, as in (c), jobs running on these VMs also get queued and are re-scheduled to run once VMs are re-allocated, as in (d).

3.2 Provisioning Algorithm

TR-Kubernetes enables users to specify an availability requirement for a specified capacity in their service description for an interactive task. As discussed in §2, since we assume interactive services are stateless and leverage Kubernetes's built-in load balancer to distribute requests across VMs, it assumes any composition of VMs that satisfies the ECU requirement is acceptable, as the load balancer will distribute requests evenly based on each VM's resource capacity. Of course, there are many such VM combinations with different costs that could satisfy a fixed ECU requirement, such as 10,000 ECUs. For example, 1000 `m5.large` on-demand VM's, which have 10 ECUs, at a cost of $96 per hour (or $0.0096/ECU/hr), or 34 `m5.large` and 28 `m5.24xlarge` on-demand VM's (345 ECUs) at a cost of $132.29 (or $0.013229/ECU/hr). The latter option costs more, in part, because the `m5.24xlarge` has slightly more memory per ECU (1.1MB/ECU to 0.8MB/ECU). Note that TR-Kubernetes's current policy specifies capacity in terms of ECUs, and enables applications to only set a minimum aggregate memory threshold.

The differences in cost between different combinations of on-demand VMs are small, since on-demand VMs (within each VM class) are consistently priced in proportion to their ECU and memory capacity. The cost differences, however, are often much higher for transient VMs. For example, Figure 1 shows the `m4.2xlarge`'s spot price and the `d2.2xlarge`'s spot price over the

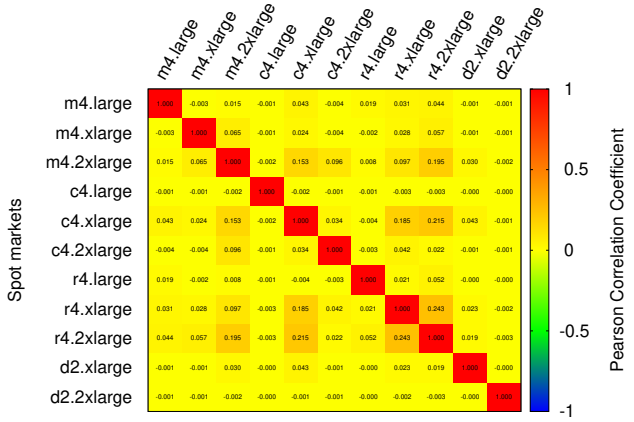


Fig. 5. Heatmap showing the correlation in availability periods of spot VMs in `us-west-1c` from September to November 2017. The figure shows that availability periods are largely independent across spot VMs.

same three-month period. The figure shows that the `m4.2xlarge` is often much cheaper (by 4×) than the `d2.2xlarge`, but is also occasionally more expensive. However, even though the `m4.2xlarge` is cheaper on average, its spot price is more variable. In this case, the capacity of `d2.2xlarge` and `m4.2xlarge` is similar, with 28 and 26 ECUs, respectively. Figure 4 illustrates the complexity of selecting transient spot VMs in EC2 that are both low cost and satisfy a target availability. The figure shows that cost and availability are not correlated, such that the VM with the lowest price is not necessarily the most available, and that there are a wide range of options with different tradeoffs. Note that availability is on a log scale to emphasize that small differences are important.

Our provisioning policy addresses this problem by selecting transient VMs to jointly optimize both cost and availability subject to the availability target. To do so, TR-Kubernetes maintains a table of price and availability estimates for each transient VM. Given the table, computing the aggregate availability of different capacities for a pool of transient VMs is non-trivial, especially if transient VM availability is highly correlated. Fortunately, Figure 5 shows that availability for transient VMs is not highly correlated in EC2. While we show only 11 VM types from `us-west-1c` here, our analysis across the other regions, zones, and VM types yields similar results.

In this case, we characterize each transient VM's availability as a time-series with a value of 1 if the VM is available, and 0 if it is unavailable. Here, we assume a spot VM is available if its spot price is less than the corresponding on-demand price, and unavailable otherwise. We then compute the Pearson Correlation Coefficient (PCC) between all pairs of VM types in Figure 5. With PCC, 1 represents a total positive linear correlation, −1 represents a total negative linear correlation, and 0 represents no linear correlation. The heat map shows that the availability periods are either zero or are close to zero for all pairs of different types. Given this, in our analysis below, we assume availability periods are independent. Also note that transient VMs of the same type are perfectly correlated, since a single price dictates their availability *en masse*.

3.2.1 Computing the Availability of a Target Capacity. Before describing our provisioning algorithm, we first outline how to compute the availability of a target capacity C across a pool of transient VMs with different availabilities. Since transient VMs of a given type are either available or unavailable, our approach, described below, essentially computes the probability of all possible available/unavailable combinations, and then sums the probabilities of all combinations that yield a capacity $\geq C$. To do so, we first denote each transient VM's availability as p_i and its capacity as c_i.

We can then represent a transient VM i as a polynomial of degree $n_i \times c_i$, where we have n_i for each transient VM i. This polynomial $Q_i(x)$ is given by the following equation, which we use below.

$$Q_i(x) = (1 - p_i)x^0 + p_i x^{n_i c_i} \tag{1}$$

This representation conveys that with probability $(1-p_i)$, transient VMs of type i are unavailable and yield zero capacity, and with probability p_i, they are available and yield $n_i \times c_i$ capacity. Here, the exponents of x represent the capacity of transient VMs of type i, while their coefficients represent the probability that a certain capacity is available (either zero or $n_i \times c_i$). We use this polynomial to indirectly represent the probability mass function of transient VMs of type i. Of course, any single VM type is either available or unavailable, which yields a simple form for $Q_i(x)$. To compute the availability of an arbitrary capacity C for a pool of different VM types, we require a representation for the entire pool's probability mass function, which is more complex, since it must capture the capacities and availabilities of different transient VMs.

Before outlining the probability mass function for a pool of transient VMs, we first consider a trivial example. Our example includes just two transient VMs, each of capacity 1 ECU with availability 90%, where our aggregate capacity availability requirement is 1 ECU at 99%. Thus, selecting one or more of any single transient VM type yields only 90% availability of at least 1 ECU, since we assume the availability of transient VMs of the same type are perfectly correlated. In this case, while we have 2 ECUs 90% of the time, 10% of the time we have 0 ECUs, which violates our capacity availability requirement of 1 ECU at 99%. If we pick one instance of each type, we can compute the pool's availability at the full capacity of 2 ECUs by multiplying the individual availabilities together, since they are independent (from Figure 5), which yields $0.9 \times 0.9 = 81\%$ availability for 2 ECUs. Similarly, we can compute the availability (or unavailability) of 0 ECUs by multiplying the unavailability of each type, yielding $0.1 \times 0.1 = 1\%$ availability. Since this case specifies only a capacity of 1 ECU, we can directly compute the availability of the pool offering 1 or more ECUs by simply subtracting the availability of 0 ECUs from 100%, which yields $1 - 0.01 = 99\%$ availability, which satisfies the availability requirement above.

We can generalize this example and apply it to any number N of transient VM types of different capacities and availabilities, for any capacity and availability requirement, by using the polynomial representation of each transient VM type i described above. In this case, for N different transient VM types, with n_i of each type, we derive our representation of the probability mass function of the transient VM pool by simply multiplying the polynomials of each transient VM, as they are independent. This gives the polynomial representation Q_{pool}(x) for the transient VM pool below.

$$Q_{pool}(x) = \prod_{i=1}^{N} Q_i(x) \tag{2}$$

Similar to above, $Q_{pool}(x)$ represents the probability mass function of the entire transient VM pool, where x's exponents represent the capacity for some set of transient VMs, and the coefficients represent the probability that this set of transient VMs are the only ones that are available. A term exists in $Q_{pool}(x)$'s expanded polynomial for every possible available/unavailable combination of the pool's transient VMs. From this equation, we can compute the availability at a target capacity m ($m \leq C$) by simply adding the coefficients of x's exponents, where the respective exponent is $\geq m$, as these are the combinations of transient VMs that yield greater than the capacity requirement. Applying this methodology to the trivial example above enables us to compute the availability of 1 ECU in a different, but more general, way, which does not rely on the capacity requirement being 1 ECU. In particular, we have $Q_1(x) = 0.1x^0 + 0.9x^1$ and $Q_2(x) = 0.1x^0 + 0.9x^1$, where multiplying these two gives $Q_{pool}(x) = (0.1x^0 + 0.9x^1) \times (0.1x^0 + 0.9x^1) = 0.01x^0 + 0.18x^1 + 0.81x^2$

and the availability at target capacity of 1 is the sum of coefficients of x^1 and x^2, which yields $0.18 + 0.81 = 0.99$.

Note that the methodology above is equivalent to taking the convolution of multiple independent binomial random variables, where we represent i) each transient VM type's availability as an independent binomial random variable $A_i \ \forall i \in [1, N]$ and ii) the transient VM pool as the sum of the respective transient VMs' binomial random variables. More formally, we use the probability mass function below of independent binomial random variables $A_i \ \forall i \in [1, N]$ to represent the ith transient VM's capacity availability where p_i, n_i and c_i represent the availability, number, and capacity of the ith transient VM in the pool, respectively.

$$f_{A_i}(x) = \begin{cases} 0, x < 0 \\ 1 - p_i, x = 0 \\ p_i, x = n_i \times c_i \end{cases} \tag{3}$$

We then represent the pool of transient VMs as the random variable Z below, which is the sum of the random variables of the individual transient VM types in the pool.

$$Z = A_1 + A_2 + \ldots + A_N \tag{4}$$

Since each $A_i \ \forall i \in [1, N]$ are independent of each other (from Figure 5), we can compute the probability mass function of Z by taking the convolution of probability mass functions of A_i $\forall i \in [1, N]$ as below, which is equivalent to Equation 2.

$$f_Z = \prod_{i=1}^{N} f_{A_i} \tag{5}$$

3.3 Greedy Algorithm

The approach above enables us to compute the aggregate availability of any specified capacity for a particular pool of transient VMs. We next outline how TR-Kubernetes selects transient VMs for the pool to minimize cost, while satisfying the target level of availability for the specified capacity. The problem is complex, since there are hundreds of transient VM types within each cloud region (including each AZ), and we may select multiple instances of any one transient VM. As a result, there are an exponential number of possible pools that satisfy the capacity availability requirement.

Our problem appears similar to a multi-dimensional unbounded knapsack problem, where the VMs are akin to items, ECUs and availabilities are akin to weight dimensions, VM pools are akin to knapsacks, and costs are akin to item value. However, there are two primary differences that prevent applying common techniques, such as dynamic programming, to the problem. In particular, in unbounded knapsack, the size of the each knapsack (along each dimension) is known in advance, whereas we do not know the final number of ECUs required for a given target availability (as it depends on the target and the individual VM availabilities). In addition, the availability dimension is not strictly additive based on §3.2, since the increase in availability from adding a new transient VM to a pool, i.e., the "knapsack," depends on the other transient VMs already in the pool.

Thus, we instead initially employed a simply greedy algorithm that selected transient VMs in order of their lowest cost (per ECU-hour) per marginal increase in availability of the specified capacity relative to the currently selected pool of transient VMs. If the currently selected pool has less than the target capacity, then we compute the marginal increase in availability of the capacity of the currently selected group. This approach favors low cost transient VMs that yield high availabilities. Note that we select based on the marginal increase in availability with respect to the currently selected group, and not the absolute availability of the transient VM. For example, if we select a transient VM of type i with an availability of 99%, the marginal increase in availability

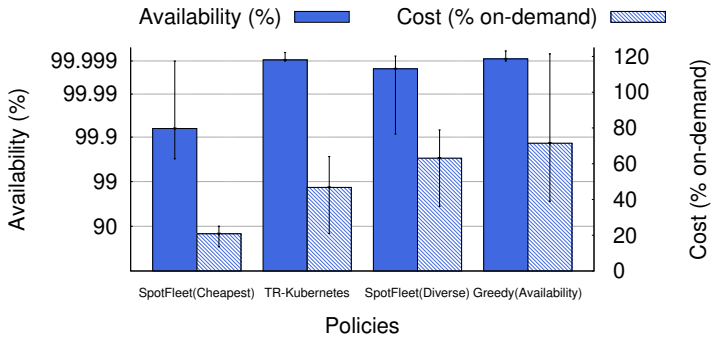


Fig. 6. Comparison of availability and cost of different provisioning policies for a target capacity of 5000 ECUs at a target availability of 99.999%.

from selecting another transient VM of type i is 0, since the availability of the same type is perfectly correlated with all other VMs of that type.

When we directly employed the greedy approach above, we found that it did not work well because it applied the same weight to cost and availability. However, transient VM availability in cloud platforms is currently high (above 90%) and users typically reason about and specify availability based on small orders of magnitude, i.e., some number of 9s of availability. In contrast, users reason about and specify cost based on much larger orders of magnitude, i.e., reducing cost by 50% is significant, while reducing it by 1% or 0.1% is generally not. By weighting cost and availability equally, our greedy sorting criteria above tends to allocate the next globally cheapest transient VM, since most transient VMs currently have availability greater than 90%. For example, if we have two transient VMs where their cost and availability is ($0.0047/ECU-hour,99%) and ($0.005/ECU-hour, 99.99%), respectively, the policy above will favor the former even though the latter has two orders of magnitude better availability at only a marginally higher cost. As a result, we modified our initial approach above by quantifying availability on a logarithmic scale using the equation below, assuming transient VM availability is <100%, where p_i is the availability of transient VM's of type i and σ_i is its cost per ECU-hour.

$$\log(\frac{1}{p_i}) \times \sigma_i \tag{6}$$

Our algorithm greedily selects transient VMs based on the criteria above one by one until the pool satisfies the specified capacity availability requirement, or its cost exceeds the cost of using on-demand VMs to satisfy the requirement, in which case TR-Kubernetes requests on-demand VMs. Note that our approach naturally diversifies transient VM selection across types, since always selecting the same transient VM (even if it has the highest availability) does not result in a marginal increase in the pool's availability. We compare our greedy algorithm with other approaches in Figure 6. In this case, we set our required capacity equal to 5000 ECUs with a target availability of 99.999%, i.e., 5 nines. We also limit transient VMs in a pool to be from the same AZ. The bars are the average availability and cost across all 14 EC2 AZs in the U.S., where the error bars indicate the maximum and minimum AZ. We plot the cost as a percentage of the on-demand cost for the equivalent VMs, and plot availability on a log scale.

We compare with multiple other provisioning policies, including those used by Spot Fleet, EC2's tool for automatically managing transient spot VMs. Spot Fleet (cheapest) is a greedy policy that greedily selects the cheapest VM until it reaches the ECU capacity; Spot Fleet (diverse) always selects the next-cheapest VM until it reaches the ECU capacity, but does not select the same VM twice (until it has selected 10 different VM types); and Greedy (availability) greedily selects the VM that results in the greatest increase in marginal availability until it reaches the capacity availability

requirement. Note that the former two policies are application-agnostic and do not consider an availability target, while the latter policy does, as it is essentially a variant of TR-Kubernetes's policy above that only considers marginal availability and not cost.

The graph shows that picking the cheapest transient VMs with Spot Fleet (cheapest) results in the lowest cost, but a much lower availability (3 nines) as well. In contrast, Spot Fleet (diverse) and Greedy (Availability) result in a high availability, but also a higher cost (60-80% of the on-demand cost). Further, Spot Fleet (diverse) results in slightly less than the availability target on average with a high variance, while Greedy (availability) has a high variance in cost (with cost sometimes exceeding the on-demand cost). In contrast, TR-Kubernetes achieves the target availability at the lowest cost at less than 50% of the on-demand cost on average.

3.4 Supporting Multi-Tier Services

Our greedy algorithm above selects transient VMs to satisfy the capacity availability requirement of a single interactive service. However, in many cases, interactive services operate as one of multiple tiers of a larger interactive service, such as 3-tier web application that includes web, application, and database server tiers. In Kubernetes, users specify multi-tier services by submitting a separate service description for each tier that specifies its resource requirements. Since a key goal of TR-Kubernetes is to minimally alter Kubernetes, we adopt the same approach with users independently specifying the capacity availability requirement of each tier. A key problem with such independent resource specifications is they do not allow users to specify availability for the aggregate multi-tier service. However, we show that we can derive this availability from the requirements of the tiers.

We consider a multi-tier service to be available if the capacity requirement of each of its tiers is concurrently satisfied. If the availability of the transient VMs allocated in each tier is independent *and* the capacities are the same, then the availability of the aggregate service is simply $(p_i)^k \forall i \in [1, k]$, where p_i is the availability of the ith of k tiers. In this case, to enable an availability of p for the multi-tier service, users should specify an availability of $\sqrt[k]{p}$ for each tier. Thus, users must specify a higher availability for each tier relative to their desired availability for the multi-tier service. If each tier's availability is independent, but the capacity is different, then we can simply treat each tier as a single transient VM equal to the specified capacity with availability p_i, and use the same approach as in §3.2.1 to compute the availability of the multi-tier service.

However, note that even when we determine each tier's transient VM allocation independently based on the greedy algorithm above, the resulting allocation is entirely dependent across tiers. That is, if each tier's capacity and availability requirement are the same, then *each tier's allocation will be exactly the same*, as our algorithm is deterministic. In this case, if each tier's availability requirement is p, then the availability of the multi-tier service will also be p, regardless of the number of tiers. This also holds if the capacity of each tier is different, but their availability requirement is the same, since any tier's transient VM allocation will be a subset of the allocation for all tiers with a higher capacity. If both the capacity and availability of each tier are different, then the availability of the multi-tier service is bounded by the tier with the lowest availability.

Summary. TR-Kubernetes does not alter Kubernetes's independent specification for each tier of a multi-tier service. The availability of a multi-tier service is bounded by the tier with the lowest availability, regardless of each tier's capacity requirement; if each tier specifies the same availability, then that is the availability of the multi-tier service, also regardless of each tier's capacity requirement. To illustrate, Figure 7 shows the availability of a three-tier service when the capacities are each equal to 400 ECUs, and a more realistic case where they are different (with capacities of 800 ECUs for the front-end, 400 ECUs for the middle tier, and 200 ECUs for the back-end). In each case, the availability requirement is 99.99%. We run the experiment using availability data over a 3 month period across all 14 AZ's in the U.S. and report averages and error bars.

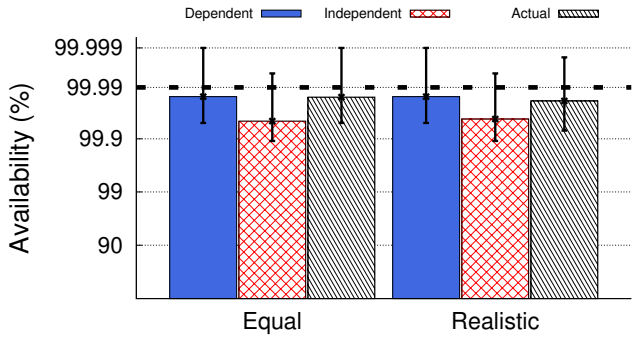


Fig. 7. Availability of a three-tier service when the capacities are equal, and when they are different, assuming dependent allocations, independent allocations, and in practice.

The graph shows the actual availability, as well as the availability if the transient VM allocations between tiers had entirely dependent and independent availabilities. Note that the latter results in a lower service availability of $(0.9999)^4 = 0.9996$. However, the graph shows that, as expected, transient VM allocations for each tier are highly dependent, resulting in the actual availability being equal to that of the dependent allocation (and near the 99.99% target).

4 IMPLEMENTATION

We implement TR-Kubernetes by minimally extending Kubernetes on AWS [5], and have publicly released our extensions.[1]. We also implement a job simulator to evaluate our approach over long periods of time using traces of transient VM prices and availability.

Prototype. By design, TR-Kubernetes minimally extends Kubernetes on AWS, as it is designed to be simple and exploit much of Kubernetes's (and other COPs') rich set of existing functions. The prototype uses Python bindings with the Boto3, numpy, and PyYAML python libraries. Kubernetes on AWS already enables users to create, update, and destroy Kubernetes clusters dynamically on AWS, either programmatically or using a command-line tool. While Kubernetes includes most of the functions TR-Kubernetes requires, they must be correctly configured. For example, the software supports both dynamic acquisition of on-demand VMs and transient spot VMs via Spot Fleet, specifically the diversified policy used in Figure 6. To create a cluster, the software first generates a template configuration file, where the user has to fill in certain cluster parameters, such as the EC2 key pair, DNS name, type of VMs to use, and the number in each node pool. Kubernetes on AWS acquires the cluster from EC2 based on this configuration file. TR-Kubernetes also configures the load balancer and replication controller for interactive services; in the latter case, it must configure the replication controller and the per container resource requirements to match the target capacity.

TR-Kubernetes interposes on the allocation process by selecting the transient spot VMs, i.e., worker nodes, that the cluster should use by autogenerating the Kubernetes configuration file. That is, we take the target capacity and availability specified by the user, execute our algorithm from the previous section, and autogenerate the configuration file that specifies the transient VM types and quantities that yield the target capacity at the target availability. Since Kubernetes on AWS supports using Spot Fleet and not requests for individual VMs, we add each transient spot VM as a separate Spot Fleet request in the configuration file. Users leverage an external tool offline to generate the configuration file above for each interactive service they submit to TR-Kubernetes. To illustrate the implementation, Figure 8 depicts a sample configuration file that TR-Kubernetes generates. For simplicity, we show only the portion of the file corresponding to the worker node-pool configuration.

[1] https://github.com/sustainablecomputinglab/tr-kubernetes.git

```
worker:
  nodePools:
  - name: nodepool-spot-m5.large
    SpotFleet:
      iamFleetArn: arn:aws:iam::XXXXXX:role/aws-ec2-spot-fleet-tagging-role
      targetCapacity: 100
      launchSpecification: {weightedCapacity: 10, instanceType: m5.large}
  - name: nodepool-spot-c5.large
    SpotFleet:
      iamFleetArn: arn:aws:iam::XXXXXX:role/aws-ec2-spot-fleet-tagging-role
      targetCapacity: 110
      launchSpecification: {weightedCapacity: 10, instanceType: c5.large}
  - name: nodepool-spot-m5.4xlarge
    SpotFleet:
      iamFleetArn: arn:aws:iam::XXXXXX:role/aws-ec2-spot-fleet-tagging-role
      targetCapacity: 183
      launchSpecification: {weightedCapacity: 61, instanceType: m5.4xlarge}
  - name: nodepool-spot-c5.2xlarge
    SpotFleet:
      iamFleetArn: arn:aws:iam::XXXXXX:role/aws-ec2-spot-fleet-tagging-role
      targetCapacity: 155
      launchSpecification: {weightedCapacity: 31, instanceType: c5.2xlarge}
```

Fig. 8. TR-Kubernetes-generated configuration file.

The revocation daemon described in §3.1 is implemented in python, and registers to receive the revocation warning from a cloud platform via platform-specific APIs. Upon receiving a warning, the daemon marks the soon-to-be revoked VM as un-schedulable in the Kubernetes master (and load balancer) to ensure it does not schedule any new containers on the VM. Next, the daemon uses the Kubernetes master to label the container pods running on the VM as "under maintenance," which prevents the load balancer from directing any additional requests to the VM but allows the current requests and/or jobs to finish. Note that this revocation daemon, like the offline tool above, operates externally to Kubernetes and requires no changes to its codebase.

Simulator. In addition to our prototype, we also built a job simulator to enable analysis of TR-Kubernetes cost savings over long periods of time [23]. Our simulator schedules tasks in first-come-first-serve (FCFS) order. We assume the CPU is the bottleneck for tasks, such that if we cannot meet the task's constraints, we allow our simulated scheduler to allocate lower-capacity VMs. However, due to the CPU bottleneck, this causes the task running time to slow down by a proportionate factor in comparison to the original running time. We do not permit the scheduler to schedule jobs on higher-capacity VMs than in the original trace, since we do not know how many additional resources the jobs can use. The simulator records each job's submit, start, and finish times on the simulated cluster, which comprises transient VMs based on our policy from the previous section with costs corresponding to their spot price traces.

5 EVALUATION

We evaluate TR-Kubernetes at small scale on EC2 using our prototype, and at large scale over a long period using publicly-available spot price traces and a month-long production job trace from Google [16]. Our prototype results focus on quantifying the effect of revocations on an interactive services's performance and reliability. We also use spot price traces to quantify the ability of TR-Kubernetes to satisfy different capacity and availability requirements, and the resulting cost compared to using on-demand VMs. We also quantify the amount of excess resources that these allocations yield, which TR-Kubernetes automatically uses to satisfy batch jobs at no additional cost. We then use the job traces to evaluate the performance of batch jobs using TR-Kubernetes

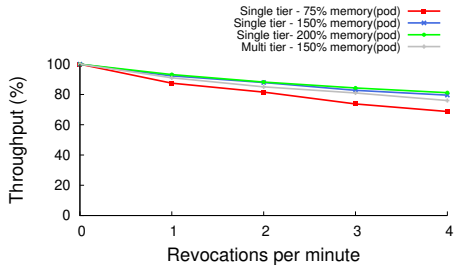


Fig. 9. Distributed web server throughput as a function of revocation frequency for different working set sizes.

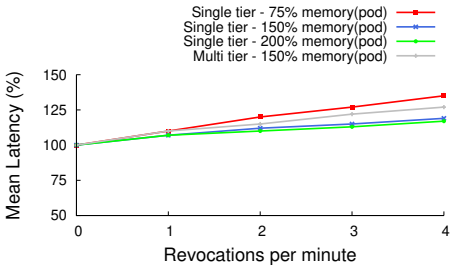


Fig. 10. Distributed web server mean latency as a function of revocation frequency for different working set sizes.

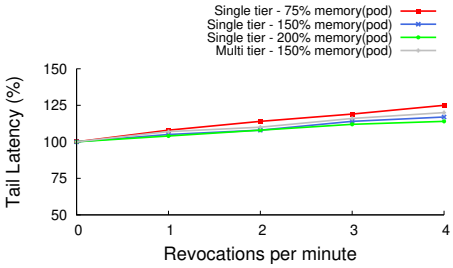


Fig. 11. Distributed web server tail latency (99%) as a function of revocation frequency for different working set sizes.

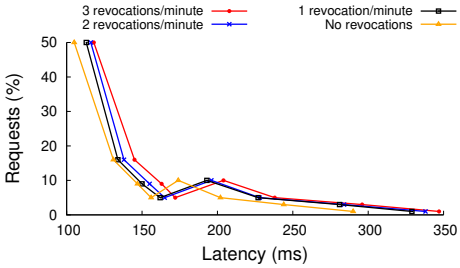


Fig. 12. Latency distribution of a distributed web server for different revocation rates.

compared to using i) on-demand VMs for the entire workload or ii) on-demand VMs for interactive services and transient VMs for batch jobs. The former is the typical setup for COPs currently deployed on cloud platforms (as EC2's Kubernetes service *requires* the use of on-demand VMs), while the latter is a natural improvement, since it runs batch jobs on transient VMs.

We run all simulation experiments using spot price data from all 14 EC2 AZs in the U.S., and report error bars that represent the minimum and maximum of each metric. Our spot price data covers 3 months from September to December 2017, which is after the latest change in EC2's spot price algorithm. We consider all spot VM types except for GPUs and FPGAs, which are not general-purpose and require support from applications. We also only consider the latest (4th) generation of VM types, as of September 2017.

5.1 Prototype Results

Our prototype results focus on the application performance and reliability impact of revocations. Since substantial prior work has optimized batch jobs for revocations, e.g., by tuning fault tolerance mechanisms [11, 12, 18, 21, 25, 26], our evaluation focuses on interactive service performance. Note that batch jobs run on TR-Kubernetes can leverage this prior work. For these experiments, we use a distributed web server that serves static content as a representative application. The server uses Kubernetes's built-in load balancer to distribute requests across 10 server replicas (running Nginx) hosted on `t2-medium` VMs in EC2.

Throughput. We first examine application performance under increasing revocation frequencies. Since replacements for revoked VMs must re-warm their cache, frequent revocations have the potential to degrade server throughput and latency. In this case, we seed our replicas with a number of image files, and use the ApacheBench (ab) web server benchmarking tool, such that

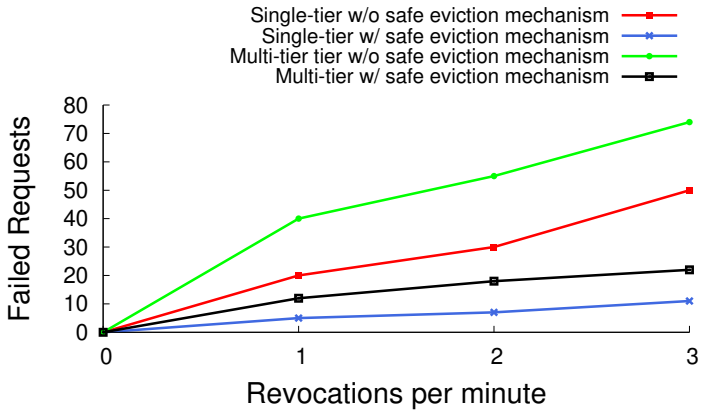


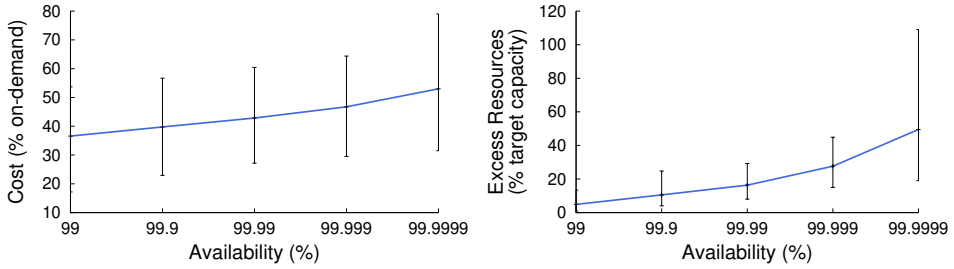
Fig. 13. Failed requests (HTTP 200) as a function of revocation rate with and without the revocation daemon.

each benchmark run consists of a total of 2000 image requests from 4 clients, which each run for an average of 8 minutes. We vary the size of the static content, i.e., image files, in each experiment such that either i) all the content fits in the VM's memory, ii) only some of the content fits in the VM's memory, requiring some content to be served from disk, or iii) most of the content resides on and must be accessed from disk. We then vary the revocation frequency from 0 revocations/minute to as high as 4 revocations/minute.

Figure 9 presents the results, which show that the larger the size of the static content relative to memory, the lower the effect of revocations on throughput. When the content fits in memory, all content is served from the memory cache, such that when a revocation occurs and flushes this cache, there is a significant decrease in throughput as the cache re-warms. This effect is less pronounced as the size of the content increases, as larger fractions of requests must be served from disk instead of the cache. Even in the worst case, significant throughput degradation requires much higher revocation frequencies, e.g., a revocation every minute or less, than on current cloud platforms where revocations occur at the scale of tens of minutes [26] to hours [17, 18]. In the spot price data (from all 14 EC2 AZs) we studied, the average mean time to revocation (MTTR) for the VM types we considered was around 150 hours across all the 14 AZs with the maximum and minimum around 200 and 80 hours respectively. Of course, dynamic content would not be subject to these caching penalties due to revocations. Here, we assume revocations occur when spot price exceeds on-demand price. Of course, dynamic content would not be subject to these caching penalties due to revocations. Thus, the performance impact of even high revocation frequencies on interactive server-based applications is likely to be low.

We also execute this experiment in a multi-tier configuration, where a second backend tier stores the content. In this case, revocations in each tier can degrade performance. As a result, the multi-tier throughput is slightly lower than the equivalent single tier case. However, as before the revocation penalty is low, resulting in only a small difference even under extreme revocation rates.

Latency. We next examine the effect of revocations on service latency. As noted above, frequent revocations result in churn that can degrade performance. To illustrate, we use the same distributed web server configuration as above, and plot mean latency, tail latency (99%), and latency distribution for the requests in Figures 10, 11, and 12 respectively. Figures 10 and 11 show that the revocation rate and working set size have similar effect on both mean latency and tail latency (99%) of the interactive service. Interestingly, the mean latency is slightly higher than the 99% tail latency due to the small number (~0.002%) of failed requests, as we discuss below, that experience long timeouts on each revocation that have a disproportionate impact on the mean. Figure 12 shows that increases



(a) Normalized Cost vs. Target Availability (b) Excess resources vs. Target Availability

Fig. 14. The cost relative to on-demand (a) and excess resources relative to the target capacity (b) for a target capacity of 5000 ECUs. Error bars indicate maximum and minimum of each metric across all AZs in the U.S.

in the revocation rate slightly increase the latency distribution by shifting it to the right. Even so, the difference between no revocations, and 1 revocation/minute, which is an excessively high revocation rate on current cloud platforms, is small. Since revocation rates this high are unlikely in practice, revocations are unlikely to have a significant effect on average or tail latency. For brevity, we omit results from similar experiments using multiple tiers that yield the same result.

Reliability. For interactive services with high throughput, simply treating VM revocations as failures can cause outstanding requests on the VM to fail. As discussed in §3, TR-Kubernetes's revocation daemon uses an advance warning of the revocation to gracefully coordinate VM removal to minimize the number of failed requests. To demonstrate, we use the same distributed web server setup as above with 10 server replicas, but serving out a simple static html page. We again use ApacheBench (ab) to benchmark the server by configuring 20 clients to send 450,000 request over a 5 minute period (or 1500 requests/second across all clients). We then vary the rate of VM revocation, and measure the number of failed requests both with and without TR-Kubernetes's revocation daemon. A failure indicates an HTTP 200 status code, which returns when the response length is less than expected. Revocations cause this HTTP failure when the VM shuts down before completing a response.

Figure 13 shows the results. We see that when vanilla Kubernetes treats revocations as failures, the number of failed requests increases linearly with the revocation rate. In contrast, TR-Kubernetes's revocation daemon decreases the number of failed requests by >5×. These few failed requests occur when the revocation daemon interacts with the Kubernetes master to make the VM un-schedulable, as this action is likely not atomic and must coordinate with both the Kubernetes scheduler and the load balancer. If we do not mark the VM as unschedulable, it results in no failed requests (as long as the VM is not revoked). In either case, the number of failed requests is small compared with the overall number of requests, even at high revocation rates. For example, in this case three revocations per minute translates to replacing 30% of the server replicas each minute, which for TR-Kubernetes results in only 9 of the 450,000 requests failing (or 0.002%) when using a single tier.

As before, the same trends are evident when we evaluate using multiple tiers. However, since a failed request at any tier fails the entire request, the number of failed requests in the multi-tier case is slightly higher (increasing the total number of failed requests by only 5-10). This demonstrates that even high revocation rates have a small impact on interactive service reliability.

5.2 Cost and Availability Analysis

We next analyze the potential cost and availability of an interactive service using TR-Kubernetes over 3 months using traces of EC2 spot prices to infer realistic cost and availability characteristics.

Our cost analysis also includes the excess resources an interactive service must acquire to maintain its target capacity availability requirement.

Cost Analysis. Figure 14 quantifies both the cost (a) and excess resources (b) on the y-axis for different availability targets on the x-axis for a capacity requirement of 5000 ECUs. For context, this capacity is equal to ~333 `m5.xlarges` or ~167 `m5.2xlarges`. We normalize our cost to the cost of satisfying the target 5000 ECU capacity using the on-demand VM with the lowest cost per ECU-hour. The figure shows that, as expected, the cost increases as we increase the availability target (a), although the increase in cost is not substantial when going from 2 nines availability (at 38% the on-demand price) to 6 nines availability (at 53% of the on-demand price). While there is some variance across AZs, as indicated by the error bars, in all cases, the cost for even 6 nines availability, which translates to 31.5 seconds of capacity below the requirement per year, is less than the cost of the equivalent on-demand VMs.

Since EC2's SLA for on-demand VMs only requires refunds if availability is less than 99.99% due to failures, if we assume 99.99% as the on-demand and transient availability due to failures, i.e., where external users cannot connect to the VM, then TR-Kubernetes's redundancy can actually enable it to achieve higher availabilities, assuming hardware failures are independent, i.e., there is not a data center-wide outage. We can modify our provisioning policy to consider unavailability due to failures by simply reducing the availability by a factor of 0.01%. We could also include on-demand VMs at this availability in our policy.

To summarize, Figure 14(a) shows that TR-Kubernetes can achieve *higher availabilities* than using on-demand VMs at a *lower cost*, ranging from 20% to 80% of the on-demand price depending on the availability requirement. Figure 14(b) then shows the amount of excess unreliable transient VM capacity we must purchase to enforce our availability requirement. As expected, the higher the availability requirement, the more excess unreliable capacity we must purchase to enforce it. This excess capacity is the reason for the cost increase in Figure 14(a). At 5 nines availability, we must purchase 30% more capacity than necessary, and for 6 nines availability, we must purchase 50% more capacity. As discussed below, TR-Kubernetes leverages this excess capacity to execute low-priority batch jobs. If we are able to utilize this excess capacity instead of purchasing on-demand (or transient) VMs to execute these batch jobs, then the cost to execute our mixed interactive and batch workload would decrease further relative to Figure 14(a), as we increase the availability target.

Note that 99% availability yields very little excess capacity, which indicates that many transient VMs are mostly available (with near 99% availability). One second-order effect of adopting TR-Kubernetes's optimizations, and other similar transient VM optimizations, is an increase in transient VM value and potentially price (due to increases in demand). Such increases may reduce transient VM availability, increase its cost, and require more excess capacity. However, while the magnitude of these metrics may change, the tradeoff between them will remain the same. Transient VMs with lower availability will require more excess transient capacity to enforce the same availability requirement. If that excess capacity can be productively used to run batch jobs that would otherwise run on on-demand or transient VMs, then we can amortize the cost of these excess resources across the on-demand VMs required to execute both the interactive and batch workloads.

Availability Analysis. The analysis above uses the actual availability of the transient VMs based on their price data over the 3 month trace period. As a result, our provisioning algorithm always satisfies the capacity availability requirement. However, in practice, spot prices and availability are not known *a priori*, requiring us to first estimate spot prices and availability based on recent history. To understand how these estimates affect our accuracy at satisfying the availability target, we experimented with different prediction window durations in Figure 15. The window specifies the interval length over which we estimate VM availability, such that a window of seven days estimates availability over the previous seven days. We again use a capacity of 5000 ECUs and an availability

target of 99.99%. The graph then compares the realized availability based on the predicted estimates with the target availability. We run the experiment for every window duration within the 3 month period across all 14 AZ's in the U.S. and report averages and error bars (representing the minimum and maximum availability).

Figure 15 shows that the realized availability across prediction windows remains relatively stable, but tends to be one 9 less than the target. The error derives from changes in VM prices and availability over time. In contrast, the oracle that assumes future knowledge of availability and prices always yields an availability strictly greater than the target (since it continues adding VMs until it reaches the target). One way to account for this error is to leverage prior work on spot price prediction to attempt to better predict future prices, rather than use simple historical estimates. Another way to address this problem is to simply set the target higher than required, which we show in the figure by also including the availability for a target for 5 nines. The 5 nines target achieves at or near the 4 nines availability for all window durations. Note that, Figure 14(a), which plots normalized cost versus availability, quantifies the cost of inaccurate predictions when combined with Figure 15. In that graph, specifying a higher target by 1 nine (to account for inaccurate predictions) reduces savings by ~5% relative to on-demand prices.

5.3 Job Performance

We also compare TR-Kubernetes with two other approaches for provisioning cloud-backed COPs to satisfy mixed-use workloads: i) dynamically allocate on-demand VMs for all interactive and batch jobs and ii) dynamically allocate on-demand VMs for interactive services and transient VMs for batch jobs. While the previous section only examined availability, cost, and excess resources based on price data, we also quantify performance, in terms of completion time, for batch jobs in this section. For our workload, we use a Google cluster trace that provides data for a 12.5k machine cluster over a month-long period [16]. Each job in the trace comprises one or more tasks, each of which includes a set of resource requirements for scheduling (packing) the tasks onto machines. The tasks can be high-priority production tasks or low-priority non-production tasks. The resource requirements of each task are normalized in the trace to the highest capacity of the resource on any machine. Since the highest capacity resource is not listed in the trace, we assumed it was a state-of-the-art machine at the time of the trace (in 2011), e.g., an 8-core 8-socket Intel Xeon processor (64 core machine).

Since we use ECUs to quantify capacity, we equate this 8-core machine to 26 ECUs based on EC2's VM listings in 2011, and translate the normalized values in the job trace to ECUs. Based on this, our job trace analysis shows that all non-production batch jobs collectively require ~76k ECUs to complete in 29 days. In addition, the vast majority of the batch jobs in the trace are relatively short. To illustrate, Figure 16 shows the cumulative distribution of job lengths showing the high fraction of short jobs versus long jobs.

Job Performance Analysis. We examine job performance in the trace when run on the excess resources after satisfying the capacity availability requirements of the interactive services. Of course, the amount of excess resources directly affects job performance, so we measure performance for different ratios of required service capacity to required batch capacity in our workload. For example, a 1:1 ratio for the cluster trace dictates that the interactive services require the same number of ECUs as the batch jobs on average. The larger the ratio, the more excess resources are available to run batch jobs, and the better their performance. We quantify performance in terms of job completion time normalized to its completion time, i.e., the time from job submission to completion, in the original job trace. As before, we run experiments for all 14 EC2 AZs in the U.S. over 3 months of price data, and report the average with error bars indicating the maximum and minimum AZ.

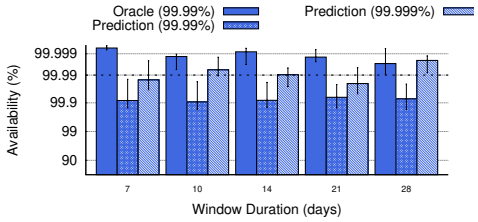


Fig. 15. Unpredictable transient VM availability (due to price changes) affect TR-Kubernetes's accuracy at meeting its availability target.

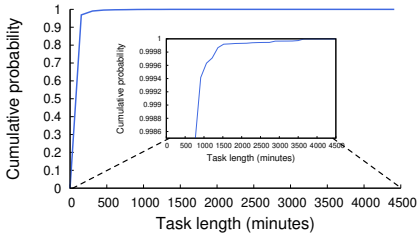
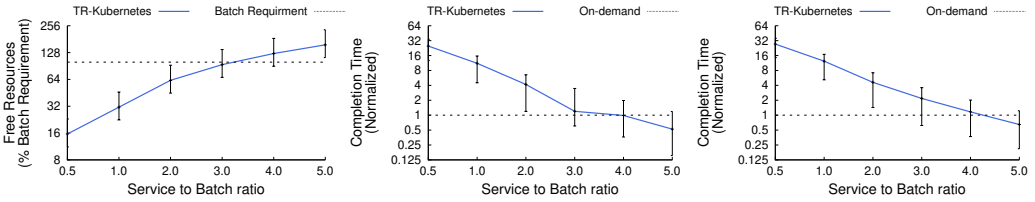


Fig. 16. CDF of the batch job lengths in job trace.



(a) Excess Capacity (b) Running Time (ckpt) (c) Running Time (no ckpt)

Fig. 17. The excess capacity as a function of the service-to-batch capacity ratio for the Google trace for a 99.999% availability target (a). The job performance, both with (b) and without checkpointing(CP) (c), as a function of the service-to-batch capacity ratio, normalized to the completion time in the original trace.

Figure 17 shows the job performance results from our simulator as a function of the ratio of the average ECUs required by the interactive service workloads versus the batch workloads. In this case, we set our target availability equal to 5 nines. Figure 17(a) plots the excess capacity as a function of the percentage of the batch workload that we can execute without acquiring additional resources (plotted on a log scale). The figure shows that as the fraction of our interactive service workload increases linearly, we get a super-linear increase in the batch workload we can execute using the excess capacity. At roughly a ratio of 3, we can execute 100% of our batch workload for free, and with a ratio of 1, we can execute roughly one-third of the batch workload for free.

Figure 17(b) and (c) then plot the job performance in terms of completion time if we only use the excess capacity to execute batch jobs, rather than always request additional on-demand or transient VMs to execute queued jobs. In this case, we normalize completion time to the completion time of the original trace. The only difference in the figures is whether or not the batch jobs are checkpointed using the spot hibernation mechanism mentioned in §2.2. The results show that, at least for the Google trace, such checkpointing does not significantly improve job performance due to the large number of short jobs, as shown in Figure 16, with few jobs exceeding 100 minutes. Of course, a different trace with longer jobs would yield a different result. However, this result may motivate why such checkpointing is not built into (or natively supported by) Kubernetes, as it is complex to implement and does not improve performance for workloads dominated by short jobs.

As expected, a service-to-batch ratio of 3 results in the same completion time as in the original trace with increasingly longer completion times for lower ratios (due to the decrease in available excess resources). The log scale here works in reverse, with job times becoming substantially longer as the ratio lowers, with roughly 10× slowdown with a ratio of 1. While this relative slowdown is high, we emphasize that these experiments restrict us to only using the excess resources acquired from maintaining interactive service availabiliy, and due to the many short jobs, the absolute

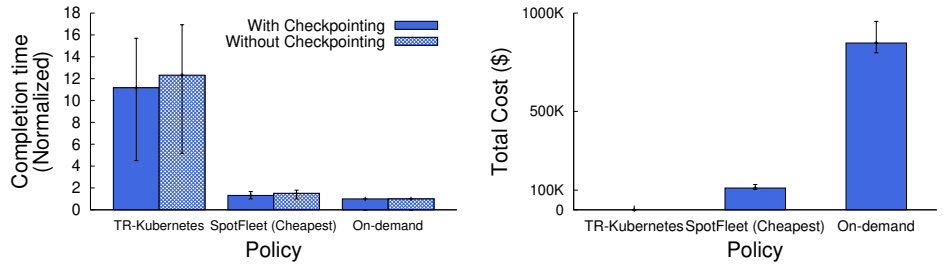


Fig. 18. Comparison of TR-Kubernetes with a policy that uses transient VMs and on-demand VMs to execute the batch workload. We plot the completion time (left) and the cost (right) where we set the service-to-batch ratio equal to 1 and the availability target to 99.999%. We assume TR-Kubernetes only executes batch jobs on excess resource capacity. Results are the average across all 14 AZs in the U.S. with error bars to indicate the minimum and maximum.

slowdown may not be as unreasonable as it appears, i.e., as delaying a 1 minute job to 10 minutes may not be too onerous. In contrast, beyond a ratio of 3, we have more resources than in the original trace. Since the original trace had to queue some jobs, we are able to make productive use of these resources by reducing queue times and our normalized completion time. Of course, we will eventually reach a point where we cannot make productive use of these excess resources by running batch jobs, i.e., once queue times are at 0 then we can no longer improve performance. At this point, the additional excess resources will no longer improve cost-efficiency.

Cost Comparison. Finally, we compare our job performance and cost with i) using on-demand VMs to satisfy the interactive service requirement (with no redundancy) and to execute the batch jobs on demand and ii) using on-demand VMs to satisfy the interactive service requirement (with no redundancy) and transient VMs to execute the batch jobs on demand. Figure 18 shows the result.

Again, the experiment setup is the same as above (with a 99.999% availability target), and where we have set the service-to-batch ratio equal to 1. As in the previous experiments, Figure 18(left) shows that restricting ourselves to excess resources results in a slowdown for batch jobs. However, Figure 18(right) shows estimates of the absolute cost to execute the batch workload based on our assumptions above (of 76k batch ECUs over the month). By comparison, using on-demand VMs and transient VMs both yield performance near that of the original trace. However, using TR-Kubernetes's excess capacity to run batch jobs is essentially free, as it incurs no additional cost, while we estimate using on-demand VMs would cost near $1M.

Using the cheapest available transient VMs directly to execute the batch workload would decrease the cost by 90% to roughly $100k, as it reflects the average discount offered by transient VMs. While 90% is a significant relative reduction, $100k is still a substantial sum that is much greater than $0. Given TR-Kubernetes's simplicity, and assuming users can quantify their capacity and availability requirements for interactive services, the potential cost savings TR-Kubernetes offers is worth the overhead of deploying/configuring it. While these results indicate that significant savings are possible, the exact savings and job performance are a function of the workload.

6 RELATED WORK

There has been a significant amount of work on optimizing transient VMs recently. Most of this work has focused on optimizing batch applications, as transient VMs arose from the desire to revoke resources from low-priority, delay tolerant batch jobs and give them to high-priority, latency sensitive services. However, even though batch applications are delay-tolerant, high rates of revocation incur high performance penalties, especially for distributed big data frameworks that maintain intermediate data in volatile memory, which is lost on each revocation, e.g., as in

Spark [27], Naiad [15], Tensorflow [7], etc. Thus, prior work primarily focuses on optimizing the selection of transient VMs and configuring of fault-tolerance mechanisms, such as checkpointing, replication, and migration, to minimize the performance impact of revocations [14, 17–19, 21, 24, 26, 28]. In general, TR-Kubernetes differs from this work in its focus on supporting the availability requirements of interactive workloads on transient VMs, and using the excess resources from doing so to execute delay-tolerant batch jobs.

In particular, Flint modifies Spark [27] to support latency-sensitive batch jobs on transient VMs [17]. Flint's policy selects the transient VM type that minimizes the expected cost to run a Spark job when considering its resource usage, price, revocation rate, and the overhead of checkpointing. However, Flint observes that, while using a single transient VM type results in the minimum expected cost, it yields a wide variance in job running time (or latency), since any revocation event concurrently triggers the revocation of all VMs. Flint reduces this latency variance by extending its selection policy to diversify across different transient VM types (by selecting some non-cost-optimal transient VMs). Importantly, our work is orthogonal to Flint (and similar prior work cited above) that focuses solely on optimizing batch jobs, as this work could be directly applied to batch jobs submitted to TR-Kubernetes. Unlike TR-Kubernetes, this prior work does not address interactive applications or support high availability requirements.

There has also been some research that focuses on running interactive workloads on transient VMs. For example, SpotCheck [19] operates at the virtualization layer and maintains a synchronized in-memory backup of the transient VM memory state, such that if a revocation occurs, the backup VM can seamlessly take over [20], similar to VM-based primary-backup systems, as in Remus [10]. As a result, a key advantage of SpotCheck is that it never loses an application's in-memory state, and thus can support stateful interactive services that maintain session state. SpotCheck keeps costs low by highly multiplexing the backup server among many transient VMs and thus amortizing their cost. While effective at entirely masking the effect of revocations from interactive services, SpotCheck is complex to implement, requiring the use of nested virtualization and the propagation of individual memory writes to the backup server. By contrast, TR-Kubernetes is simple, and leverages much of the functionality already present in COPs, which makes it more likely to be adopted in practice. However, unlike SpotCheck, TR-Kubernetes only supports stateless interactive services using Kubernetes's built-in load balancer. Since SpotCheck is a virtualization mechanism, TR-Kubernetes could re-implement it at the container layer to support stateful interactive services.

Similarly, Tributary [6] applies an approach called "spot dancing," which leverages spot VMs to support latency SLOs for interactive services at low cost, by intentionally selecting and bidding on spot VMs to cause revocations. Tributary exploits a rule in the EC2 billing model that does not charge users if revocations occur within their first hour. Thus, the approach attempts to maximize its number of free cycles. Tributary does have similar dynamics as TR-Kubernetes in that it provisions many more resources than necessary, which provides headroom for interactive services to absorb demand spikes. However, Tributary does not ensure availability requirements or leverage additional resources to execute batch jobs. Tributary also does not build on existing COPs to simplify its use or adoption. More generally, our approach is not specific to spot VMs or EC2's billing model, as we only use the spot price traces to derive transient VM availability estimates. As a result, TR-Kubernetes is applicable to any variant of transient VMs with such a characterization of availability.

7 CONCLUSION

This paper presents TR-Kubernetes, a minimal extension of Kubernetes that executes mixed interactive and batch workloads on unreliable transient VMs dynamically acquired from cloud platforms. To do so, we design a greedy provisioning algorithm that satisfies a capacity availability requirement

at low cost. We implement TR-Kubernetes on EC2's variant of transient VMs, and evaluate its performance, reliability, cost, and availability using publicly available benchmarking tools, availability and cost data, and workload traces. We show that, when compared to running interactive services on on-demand VMs, TR-Kubernetes is capable of lowering costs (by 53%) and providing higher availability (99.999%).

Acknowledgements. We would like to thank our shepherd Shaolei Ren and the anonymous reviewers for their thoughtful feedback, which improved the quality of this paper. This work is supported by NSF grant #1802523 and the Amazon AWS Cloud Credits for Research program.

REFERENCES

[1] 2017. Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/container-service/.
[2] 2017. Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine/.
[3] 2018. Amazon Elastic Container Service for Kubernetes. https://aws.amazon.com/eks/.
[4] 2018. Docker Swarm. https://docs.docker.com/engine/swarm/.
[5] 2018. Kubernetes on AWS. https://kubernetes-incubator.github.io/kube-aws/.
[6] 2018. Tributary: spot-dancing for elastic services with latency SLOs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/atc18/presentation/harlap
[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.
[8] C. Babcock. 2015. Amazon's 'Virtual CPU'? You Figure it Out, In *InformationWeek*.
[9] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. 2016. Borg, Omega, and Kubernetes. *ACM Queue - Containers* 14, 1 (January-February 2016).
[10] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*.
[11] B. Ghit and D. Epema. 2017. Better Safe than Sorry: Grappling with Failures of In-Memory Data Analytics Frameworks. In *HPDC*.
[12] A. Harlap, A. Tumanov, A. Chung, G. Ganger, and P. Gibbons. 2017. Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets. In *European Conference on Computer Systems (EuroSys)*.
[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI*.
[14] B. Huang, N. Jarrett, S. Babu, S. Mukherjee, and J. Yang. 2015. Cumulon: Matrix-Based Data Analytics in the Cloud with Spot Instances. *Proceedings of the VLDB Endowment (PVLDB)* 9, 3 (November 2015).
[15] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738
[16] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2011. *Google cluster-usage traces: format + schema.* Technical Report. Google Inc., Mountain View, CA, USA. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.
[17] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. 2016. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *European Conference on Computer Systems (EuroSys)*.
[18] P. Sharma, D. Irwin, and P. Shenoy. 2017. Portfolio-driven Resource Management for Transient Cloud Servers. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.
[19] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. 2015. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *European Conference on Computer Systems (EuroSys)*.
[20] R. Singh, D. Irwin, P. Shenoy, and K.K. Ramakrishnan. 2013. Yank: Enabling Green Data Centers to Pull the Plug. In *Symposium on Networked Systems Design and Implementation (NSDI)*.
[21] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. 2015. SpotOn: A Batch Computing Service for the Spot Market. In *Symposium on Cloud Computing (SoCC)*.
[22] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *European Conference on Computer Systems (EuroSys)*.
[23] John Wilkes. 2011. More Google cluster data. Google research blog. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[24] Z. Xu, C. Stewart, N. Deng, and X. Wang. 2016. Blending On-Demand and Spot Instances to Lower Costs for In-Memory Storage. In *International Conference on Computer Communications (Infocom)*.
[25] Y. Yan, Y. Gao, Z. Guo, B. Chen, and T. Moscibroda. 2016. TR-Spark: Transient Computing for Big Data Analytics. In *Symposium on Cloud Computing (SoCC)*.
[26] Y. Yang, G. Kim, W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B. Chun. 2017. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *European Conference on Computer Systems (EuroSys)*.
[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*.
[28] L. Zheng, C. Joe-Wong, C. Tan, M. Chiang, and X. Wang. 2015. How to Bid the Cloud. In *ACM SIGCOMM Conference (SIGCOMM)*.