CoSpot: A Cooperative VM Allocation Framework for Increased Revenue from Spot Instances

Syed M. Iqbal[*]
Amazon Web Services

Haley Li
University of British Columbia

Shane Bergsma
Huawei Cloud, Toronto Research Center

Ivan Beschastnikh
University of British Columbia

Alan J. Hu
University of British Columbia

ABSTRACT

Most large cloud operators offer a lower-priced, lower-priority alternative to regular (on-demand or reserved) virtual machines, commonly referred to as *spot instances.* Spot instances are opportunistically allocated to servers in order to utilize any residual cloud capacity, but are evicted whenever regular virtual machines need to use that capacity. This paper proposes *CoSpot*, a lightweight framework for cooperative allocation of regular virtual machines and spot instances, which allows for easy integration of arbitrary virtual machine and spot allocators. In our experiments, employing the framework achieves up to 245% improvement (average 34% improvement) in spot revenue, with *no loss* in virtual machine revenue, compared to the baseline VM and spot allocation without using our framework. We also derive and release a reusable workload with both virtual machines and spot instances, based on data previously shared by Microsoft Azure.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Social and professional topics → Pricing and resource allocation**.

KEYWORDS

cloud computing, virtual machine, spot instance, low priority VM, preemptible VM, resource allocation

[*]This work was performed while the author was affiliated with the University of British Columbia.

ACM Reference Format:
Syed M. Iqbal, Haley Li, Shane Bergsma, Ivan Beschastnikh, and Alan J. Hu. 2022. CoSpot: A Cooperative VM Allocation Framework for Increased Revenue from Spot Instances . In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3542929.3563499

And this more human love ... will resemble that which we prepare strenuously and laboriously: the love that consists in two solitudes that protect and delimit and greet each other.
— Rainer Maria Rilke, Letters to a Young Poet #7

1 INTRODUCTION

Most large cloud providers offer (at least) two different products: a regular (on-demand or reserved) virtual machine, which tenants can rent for as long as they need; and a lower-priority *spot instance*, which tenants can rent at lower cost, but which can be preempted/evicted at any time. For example (listed in order of current market share), Amazon Web Services offers *Spot Instances*, Microsoft Azure offers *Spot VMs* (formerly *Low Priority VMs*), Google Cloud offers *Preemptible VMs*, and Alibaba Cloud offers *Preemptible Instances.* For brevity in this paper, we will use the term *spot instance* for this general class of low-priority, preemptible VM, and simply *VM* for regular, high-priority VMs.

The business case for spot instances is straightforward: datacenters must be sized to handle peak VM demand, to keep regular VM customers happy. But, this results in idle datacenter capacity most of the time, and spot instances monetize the surplus resources that would otherwise be wasted — any revenue is better than nothing. Indeed, the original AWS spot instance pricing model (prior to November 2017 [19]) simply sold excess capacity to the highest bidder.

As a rapidly growing, global cloud provider, Huawei Cloud must provision capacity to support both peak demand as well as projected future (rapid) growth. We are therefore very interested in exploiting spare capacity through new spot products. Unfortunately, there is a lack of literature outlining practices for integrating spot products into existing IaaS

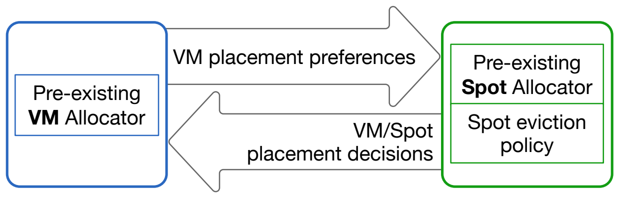


Figure 1: CoSpot allows pre-existing VM and spot allocators to make VM and spot placement decisions cooperatively. It is designed to be minimally disruptive to the existing code, and to rely only on information and capabilities that any allocator must have. In a nutshell, the VM allocator communicates preferred servers for a VM it needs to place to the spot allocator; the spot allocator chooses a server from that list according to its own preferences, and also communicates its spot placements to the VM allocator.

infrastructure (Sec. 6). There are no evaluations of tradeoffs between VM and spot revenue for different VM/spot allocation policies. There are also no existing published frameworks that enable explicit cooperation between VM and spot allocation policies or co-optimize VM and spot allocation to improve cloud operator revenue.

This paper addresses that gap by focusing on the management of spot instances from the cloud provider's perspective. Specifically, we attack the problem of VM/spot allocation: a tenant requests a virtual machine with specified computing requirements (e.g., number of cores, RAM, disk, etc.), and the allocator must place this virtual machine onto a physical server in the data center with sufficient unused resources to meet the request. For regular VMs, the tenant can continue to use the VM for as long as the tenant desires; for spot instances, the tenant's job might run to completion, but the allocator also has the problem/freedom of deciding which spot instance(s) to evict if their resources are needed to satisfy a request for a regular VM. From the cloud operator's perspective, the goal is to have an allocator that can satisfy an ongoing sequence of such requests, while optimizing some metric (e.g., maximizing revenue).

Our proposed solution is CoSpot, a novel framework by which independent VM and spot allocators can cooperate (Fig. 1). CoSpot allows for easy integration of *arbitrary* VM and spot allocators, an important consideration given the large investment that cloud operators have made in existing allocators. CoSpot may be especially valuable to newer operators who want to integrate spots into their existing framework, with minimal disruption to existing operations. However, large operators, who already have proprietary VM/spot allocators, might also benefit from the explicit and tunable cooperation enabled by CoSpot.

CoSpot presupposes the existence of (1) a VM allocator that places VMs on servers, (2) a spot allocator that does the same for spots (and that might reuse VM allocation logic), and (3) a spot eviction mechanism, by which the spot allocator decides on a set of spots to evict, to make room for a VM request. CoSpot introduces a minimal amount of communication between the two allocators: the VM allocator communicates a small set of its preferences to the spot allocator; the spot allocator communicates its choices back to the VM allocator. The key insight is that the VM allocator might be indifferent among a set of placement choices, whereas that flexibility might be valuable to the spot allocator. We emphasize that we are not proposing a new category of VM, changes to any SLAs, or a specific VM/spot allocator, but rather a framework that adds effective cooperation between *arbitrary, existing* VM/spot allocators. The goal is improved spot revenue with minimal changes to existing products/code.

We evaluate CoSpot across 54 different VM/spot allocator combinations. In all cases, CoSpot achieves substantial gains in spot revenue, compared to the baseline without CoSpot (where VM and spot allocators do not cooperate). For example, in a datacenter with only 1% spare capacity, CoSpot achieves at least (for the least-improving VM/spot allocator combination) 23% average improvement, and up to (for the most-improving combination) 59% average improvement, *with no loss in VM revenue.* I.e., CoSpot had no VM allocation failures while achieving longer spot instance lifetimes. The allocation latency with CoSpot was negligible in nearly all cases, and can always be limited by choosing appropriate framework parameters.

As an additional contribution, we also derive and release a *reusable* workload with both VM and spot instances, based on data logged by Hadary et al. [25] on Microsoft Azure. "Reusable" means that our synthesized workload contains concrete, consistent resource vectors for all VM, spot, and server types, allowing the workload to be reused to evaluate different allocation and scheduling algorithms, rather than being only a historical log from one proprietary cloud.

2 CONTEXT AND HYPOTHESES

We start with some context for our work: we establish the concept that all allocators can be viewed in terms of filtering and ranking of servers, and we explore the two hypotheses that underpin our work.

A VM/spot allocator is essentially solving an online, multi-dimensional bin-packing problem: online, because tenant requests arrive over time and must be satisfied quickly; multi-dimensional, because of the different resource types; and bin-packing, because we want to satisfy as many tenant requests as possible with the set of servers in our cloud. Since the allocator is solving a constrained optimization problem, any VM or spot allocator must face two issues for each request: feasibility and suitability. "Feasibility" means enforcing hard constraints, e.g., the allocator can place a VM/spot instance onto only servers with sufficient resources required by the instance. "Suitability" means soft optimization objectives: the allocator might prefer to place the instance on one server over another, to try to optimize, e.g., power consumption, fault tolerance, capacity for anticipated future requests, etc. For example, the OpenStack allocator architecture [51] applies "filters" to eliminate from consideration servers that don't meet the hard constraints, and then a "weigher", which scores each remaining server as to its suitability; Google's Borg allocator refers to the same steps as "feasibility checking" and "scoring" [63].

Similarly, when a VM allocation necessitates spot evictions, the spot allocator must choose a set of spot instances to evict that frees up enough resources for VM allocation (feasibility). But the spot allocator must have some mechanism to choose *which* spots it prefers to evict (suitability).

Accordingly, since all allocations must meet the feasibility requirements, **any VM or spot allocator is fully characterized by the ranking policy it uses to evaluate suitability;** the choice of ranking policy essentially defines the allocator. For the remainder of this paper, we will refer to different allocators solely by the ranking policy used. To give concrete examples, we will use the following 3 well-known, published ranking policies that are used in state-of-the-art VM allocators:

UTIL *(Utilization)* This policy ranks a server higher the more fully its cores are already utilized (ignoring spots). This implements the Best-Fit heuristic for bin-packing, by placing the VM request in the most full server on which it fits.

COS *(Cosine Similarity)* Many allocators use cosine similarity, or closely related measures, when placing VMs or jobs onto servers (e.g. [21, 22, 55]) and it is a known geometric heuristic for the vector bin packing problem [48]. If we consider the resource requirements of a VM request (e.g., cores, RAM, etc.) as one vector, and the available resources (residual capacity, ignoring spots) on a server as another, the ranking score is the cosine of the angle between the two vectors. The intuition is to place VMs on servers whose residual capacity is aligned with the resources requested. By itself, this policy did not work well in our experiments (described in detail in Sec. 4): VMs were spread around the datacenter, so larger VMs arriving later could not be placed because no server had sufficient capacity. We thus refined COS to rank any non-idle server (i.e., hosts at least one VM) over all idle servers. This is essentially the any-fit heuristic in bin-packing algorithms, where it is preferable to pack into existing open bins than to open a new empty bin.[1]

DPD *(Delta Perpendicular Distance)* Ke et al. [36] introduced DPD as a key component of the Fundy allocator, reporting better packing than COS-based prior work [22]. As with COS, the DPD ranking policy treats the VM request and the available resources on a server as vectors. The ranking score is a measure of how much the request vector would move the resource usage on the server toward a balanced usage of its resources. For example, a server with too many CPU-intensive-but-memory-light VMs would be a good place to allocate a CPU-light/memory-heavy VM request. As with COS, DPD did not work well by itself, so we added the same preference for non-idle servers as in COS.

The key insight behind our CoSpot framework is based on two hypotheses about how the ranking heuristics interact with the overall optimization objective.

Hypothesis 1*: Slightly degrading VM allocation rankings doesn't result in a large degradation of VM allocation quality.*

We can easily validate this hypothesis experimentally by simulating different VM allocation policies, but with an adjustable "knob" to degrade the rankings. Specifically, we choose an integer N, and instead of placing a VM request on the top-ranked server, we place it randomly on one of the top-N ranked servers. We then compare how many more VM requests cannot be allocated, versus the original, undegraded allocation policy.[2]

For the workloads in our simulation, we start with the Resource Central VM trace [10] (which doesn't have labeled spot instances), and the VMs only (ignoring spots) from the Protean trace [25]. For robustness, we randomly sample subtraces from these traces: each VM request in the original trace is included in each sampled trace with 1% probability (Bernoulli sampling in Sec. 4.2). From the Resource Central trace, we generate 50 such traces; from the Protean trace, we generate 10. We assume a server configuration large enough to accommodate the largest VM flavors in the traces, and then for each VM allocation policy (original, without degradation), we compute the minimum datacenter size using the methodology of Sec. 4.3 to generate datacenters with 0%–4% headroom. We then re-simulate each trace, but with $N = 1, \ldots, 32$, and plot the number of VMs that could not be allocated.

Fig. 2 shows the results for the Resource Central traces; the results for the Protean workloads are similar, but the plots

[1] Borg [63] and Protean [25] have also noted similar phenomena/solutions.
[2] This is intended as a quick experiment just to illustrate the first hypothesis. Our main experimental results to evaluate CoSpot are in Sec. 4.

are omitted for space reasons. We can see that the heuristics do work: degrading them by increasing N does increase the number of VM allocation failures. However, the number of failures is minuscule compared to the total number of VM requests (averaging 19,793 VMs in the Resource Central traces and 52,231 in the Protean traces). Furthermore, the number of VM allocation failures is usually zero if N is small, or if there's even a bit of spare capacity in the datacenter. The first hypothesis holds.

***Hypothesis 2**: The freedom created by not choosing the top-ranked server for VM allocation gives the spot allocator flexibility to achieve greater spot revenue.*

To evaluate this hypothesis, we need to propose a mechanism in which the VM allocator provides flexibility to the spot allocator, and where the spot allocator can exploit this flexibility. That is the main contribution of this paper and is described next.

3 THE COSPOT FRAMEWORK FOR COOPERATIVE VM/SPOT ALLOCATION

Rather than craft an arbitrarily complex framework for combined VM/spot allocation, we prune the design space via two principles.

First, we assume VMs are strictly more important than spots. VMs have both a higher price and higher service level expectations, e.g., AWS publishes a 99.99% uptime goal for their EC2 service [3]. Moreover, each failure to allocate a requsted VM hurts the cloud provider's reputation and could result in permanently losing a disappointed customer. Thus, we seek to minimize how much we perturb the VM allocator.

Second, our framework must work with *any* VM/spot allocators. Cloud operators have existing allocators, which they have optimized and tuned extensively, both for obvious objectives (revenue, energy, etc.) and proprietary ones, too. A useful cooperation strategy must respect and interoperate with this existing investment. Accordingly, CoSpot must rely only on information that any allocator must have readily available, and request only actions any allocator can perform.

What might these information/actions be? Obviously, each allocator needs basic information to perform allocations: total and residual capacity for each resource type on each server in the datacenter, as well as where each VM/spot instance is currently placed. Each allocator must be able to allocate and deallocate its instance type (VM or spot), and the spot allocator must be able to evict spot instances when the VM allocator requests. Beyond these obvious capabilities, we established in Sec. 2 that any allocator must have the ability to rank server choices. Thus, we construct CoSpot to rely on only these basic actions, and the two hypotheses from Section 2: that the VM allocator's top choices might

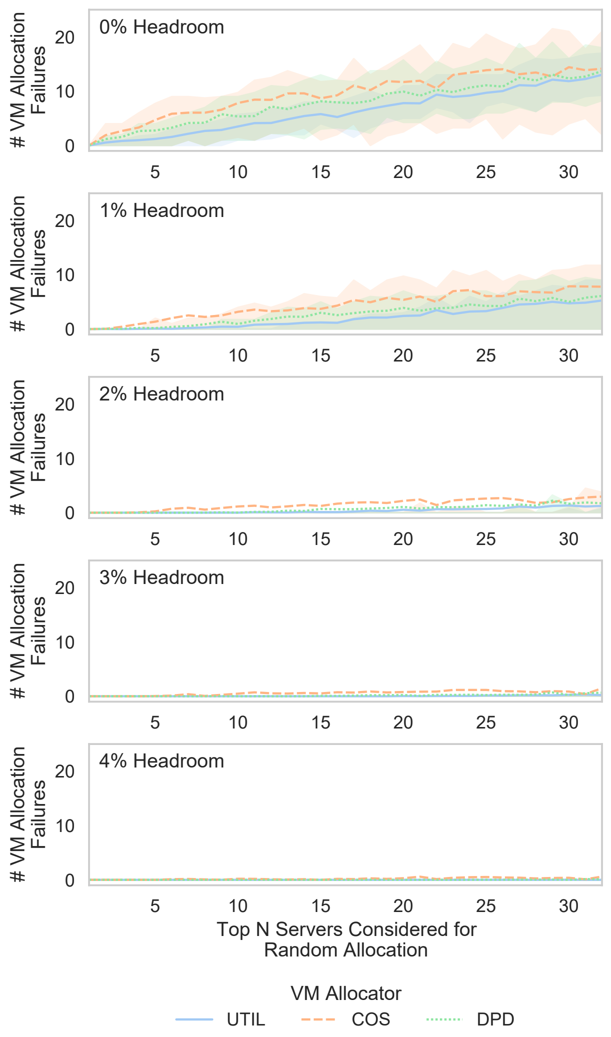


Figure 2: Loss of Allocation Quality as the VM Allocation Policy Is Degraded. The parameter N on the x-axis determines how much we degrade the VM allocator: instead of choosing the top-ranked server for each VM, the allocator randomly chooses one of the top-N ranked servers. ($N = 1$ means the VM is always placed on the top-ranked server, as would normally be done; $N = 2$ means the VM is placed on one of the top two server choices with equal probability; etc.) The y-axis is the number of VM requests that could not be allocated (due to resource fragmentation). The lines show the average # of VM allocation failures for each of the 3 VM allocation policies, over 50 randomly sampled workloads; the shaded regions show the interquartile range. Increasing N does result in more failures, but the number is small (compared to an average 19,793 VM requests in each workload sample), and is zero for small N and/or greater headroom.

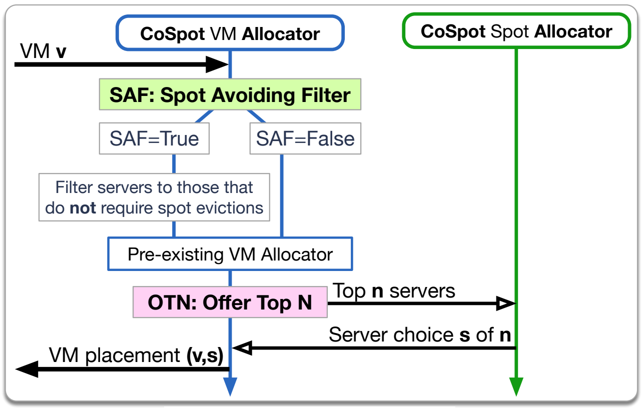
not differ much (Hypothesis 1), but this flexibility may be valuable to the spot allocator (Hypothesis 2).

Our design baseline is without cooperation: a VM allocator handles VM requests without regard to spots; the VM allocator communicates its decisions to the spot allocator; and the spot allocator handles spot requests, using datacenter capacity leftover by VMs, and also decides and performs spot evictions when a VM allocation (which ignored resources consumed by spots) exceeds residual capacity (including spots) on a server. CoSpot augments this baseline with a limited amount of communication between the two allocators (Fig. 1). This communication is controlled by two independent parameters that adjust how hard the VM allocator tries to cooperate with the spot allocator: (Fig. 3 gives a pictorial summary of how these two parameters work in CoSpot.)

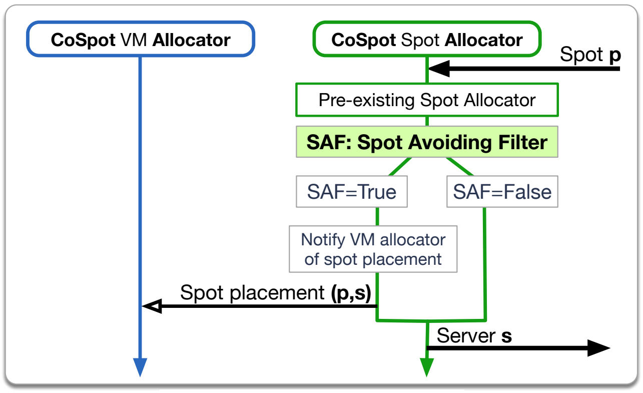
Offer Top N (OTN). The *OTN* parameter exploits the suitability ranking that any VM allocator must perform. It is a positive integer, and $OTN = n$ means the VM allocator, instead of placing a VM on its top-ranked server, will instead pass a list of its top n ranked servers to the spot allocator. The spot allocator then chooses among these n servers, all of which the VM allocator ranked highly, on which server it wants the VM allocator to place the VM. Varying n adjusts how much the VM allocator is willing to defer, among its top choices, to the spot allocator's preferences. Modifications to the pre-existing allocators are minimal: The VM allocator needs to perform ranking anyway, and the spot allocator can reuse whatever eviction suitability policy it already has, evaluated on all n choices, to choose the most suitable server, as well as what set of spots (if any are needed) to evict on that server.

Spot Avoiding Filter (SAF). The *SAF* parameter explores the more extreme position that avoiding spot evictions is preferable whenever feasible. It is a Boolean flag, and if true, the VM allocator will always rank higher any server where the VM can be allocated without requiring any spot evictions. For example, if *SAF* is true and $OTN = 1$, then the VM allocator will pick its top-ranked server among those that can accommodate the request without any spot evictions (if such a server exists), or else its top-ranked server regardless of spots (if no such server exists). If $OTN = n > 1$ (and *SAF* is true), the spot allocator gets to choose from a list of n servers, exactly as above, but this list will always have servers that avoid spot evictions ranked higher. Conceptually, this is equivalent to pre-filtering out all servers that require spot evictions, and falling back to spot eviction only if necessary. *SAF* is easily implemented in the VM allocator by allowing it to see (or track, by snooping the spot allocator's decisions) the spot allocator's residual capacity.

We use $OTN = 1$ and SAF = false as the baseline in our evaluation. This corresponds to the VM allocator optimizing purely for VMs, with no regard for the spot allocator. We will



(a) VM placement process



(b) Spot placement process

Figure 3: CoSpot Framework Overview. CoSpot assumes a pre-existing VM allocator, which chooses on which server to place an incoming VM request, and a pre-existing spot allocator, which does the same for spot requests. The CoSpot framework adds limited communication between the two allocators, to enable cooperation (Fig. 1). This communication is controlled via two independent parameters: *OTN* ("Offer Top N") and *SAF* ("Spot Avoiding Filter"). Subfigure (a) (above) shows the process of allocating a VM request. If *SAF* is true, the VM allocator will consider only servers where no spot evictions are required to allocate VMs and filter out the rest from consideration. (Not shown is that if there are no such servers, the VM allocator will fall back to the full set of servers.) When $OTN = n$ and the VM allocator is trying to allocate a VM, it communicates its top n choices of servers to the spot allocator, which selects its preference among them. Subfigure (b) (below) shows the process of allocating a spot request. The only change to the pre-existing process is that if *SAF* is true, the spot allocator communicates its placement decisions (or residual capacity) to the VM allocator so that the latter is capable of determining whether a given server can host an incoming virtual machine without requiring spot evictions.

Parameter	Values
Workload	10 Random Samples Each Using {Bernoulli, Batched} Sampling
Datacenter	0%, 1%, 2%, 3%, 4% Headroom
VM Allocation	UTIL, COS, DPD
Spot Allocation	sUTIL, sCOS, sDPD, vmaUTIL, vmaCOS, vmaDPD
Spot Eviction	FIFO, LPML, LPO
SAF	True, False
OTN	1, 2, 4, 8, 16, 32

Table 1: Summary of Experiments. We measure spot revenue gained and VM revenue lost (if any) for all 64,800 combinations of these parameters. We vary workload and datacenter headroom to assess robustness to workload variation and operating conditions. We vary VM/spot allocation and eviction policies to assess the generality of CoSpot to different allocator designs. ***SAF*** **and** ***OTN*** **are CoSpot parameters that control the degree of cooperation.**

also show results for the two VM/spot allocation algorithms proposed by López García et al. [39], which are the only comparable, published VM/spot allocation algorithms.

4 EXPERIMENTAL EVALUATION

Our experimental methodology is the cross-product of multiple layers. First, we seek to establish that CoSpot provides benefits for arbitrary VM and spot allocators. Hence, we evaluate 54 such combinations (= 3 VM allocators $\times$ 6 spot allocators $\times$ 3 spot eviction policies). For each combination, we sweep through 12 settings for *OTN* and *SAF*. For robustness, we repeat each of the above experiments for 10 workloads sampled using each of 2 different random sampling methods, and 5 different levels of spare capacity ("headroom") in the datacenter, yielding 64,800 total experiments. Each experiment was run on a purpose-built simulator, simulating each VM/spot request/deallocation/eviction in the workload, and computing VM and spot revenue. Table 1 summarizes the experiments. Details are below.

4.1 Revenue as Figure of Merit

Our goal is to maximize spot revenue while minimizing (or avoiding altogether) any loss to VM revenue. Different providers have varied and idiosyncratic pricing models (e.g., billing by the second or hour, depending on OS version; not being charged if interrupted in the first hour, or without enough notice, etc.), but in all cases, the revenue of a VM/spot is basically the product of how expensive the VM/spot is (bigger costs more) times how long the VM/spot ran. Accordingly, for simplicity and vendor-neutrality, if a VM/spot uses c cores for t seconds, we adopt the product ct as the figure of merit in our experiments, and refer to this as "revenue" for brevity. (We will revisit this topic briefly in Finding 8 in Sec. 5.) Accordingly, a VM/spot that is not allocated is a loss of revenue, as is a spot that is evicted before completion (we assume no migration of evicted spots). Also, note that we will always report both VM and spot "revenue", refraining from trying to compare them or combine them into a single figure of merit, because each cloud operator may choose different relative pricing of VMs and spots.

4.2 Workload Synthesis

A realistic evaluation requires a realistic workload. Fortunately, Hadary et al. [25] recently published a log of all VMs/spot instances on a portion of Microsoft Azure's cloud over a 14 day period (with deallocations tracked for 90 days). This is the only publicly available commercial cloud workload that has both VMs and spots identified.

Unfortunately, this original log is not usable for evaluating allocators, as the log does not contain the actual sizes of each VM/spot type; instead, it contains the *fractional resource usage* of a VM/spot type on a specific server type, and *only if that VM/spot-server combination occurred in the log*. If a different allocator placed the VM/spot on a different server, we would not know how much resources it would consume. For example, if VM v_1 uses half the cores[3] on server type t_1, we know nothing about how much resources v_1 would use on a different server type t_2.

To resolve this missing information we formulated an integer linear program (ILP), with variables for the amount of each resource in each VM/spot and server type, and constraints for each fractional resource usage datapoint in the log. Any solution to the ILP computes consistent resource amounts for every VM/spot and server type. Continuing our example, if VM v_2 uses all the cores on server type t_1, but only half the cores on server type t_2, then we can compute that v_1 would use only a quarter of the cores on t_2:

$$\begin{array}{rcl} v_1^{\text{cores}} & = & 0.5t_1^{\text{cores}} \\ v_2^{\text{cores}} & = & t_1^{\text{cores}} \\ v_2^{\text{cores}} & = & 0.5t_2^{\text{cores}} \\ \hline \therefore v_1^{\text{cores}} & = & 0.5t_1^{\text{cores}} = 0.5v_2^{\text{cores}} = 0.5\left(0.5t_2^{\text{cores}}\right) \end{array}$$

Surprisingly, the resulting ILP was infeasible, meaning the published log is inconsistent with *any possible* concrete resource quantities. Investigating, we discovered that some VM

[3] These numbers are for illustration only. The actual log provides ratios for cores, RAM, hard disk, SSD, and networking, but in this paper, we used cores and RAM only, as these were in use in all VM/spot and server types.

types had inconsistent resource requirements, e.g., one VM type apparently meant the bare metal server, which implies different resource amounts on different servers. We tried to compute irreducible infeasible subsets to identify such inconsistent VM types, but the ILP solver failed to complete. (We used the state-of-the-art Gurobi [24] solver.)

What eventually worked was a slack-based relaxation: We added one slack variable per constraint, e.g., the first constraint above becomes $v_1^{\text{cores}} = 0.5t_1^{\text{cores}} + s_i$, where the slack variable s_i is a new, completely unconstrained variable. These ensure that the ILP is feasible, and we can then ask the solver to minimize the sum of the absolute values of the slack variables. This new model determined which constraints required the largest slack to be satisfied. We removed the VM/spot types associated with these constraints one-by-one until no slack was required for a feasible solution. Solving the resulting model gave consistent, concrete core and RAM requirements for the remaining VM/spot types. Note that this solution is not necessarily the original VM, spot, and server numbers — they might be scaled arbitrarily — but the consistency means these numbers will behave exactly the same way for any allocation policy. In all, 18 out of 265 types were excluded, accounting for less than 6% of the original trace. The result is a reusable workload, with concrete core and RAM numbers for all VM/spot and server types, containing 94% of the originally logged data.

The synthesized workload is a single, very long (approx. 10 million create/delete events) trace. For experimentation, we found it desirable to have multiple (to avoid overfitting and assess robustness), shorter (to keep simulation tractable) traces. Accordingly, we created two sets of 10 smaller independent traces via two different random sampling methods: (1) **Bernoulli Sampling:** To create each independently sampled trace, each VM/spot in the original trace is included in each sampled trace with probability $p = 0.01$. The sample includes both allocation and deallocation events for the sampled VMs/spots. (2) **Batched Sampling** as advocated by Bergsma et al. [6]: Contiguous VM or spot requests from a single user are grouped into a *batch*, and the entire batch is included in a sampled trace with probability $p = 0.01$. This creates higher variance in the sampled traces, but better preserves the burstiness of the original trace.

The workload traces and associated scripts are available at https://github.com/DCResourceManage/cospot-socc2022

4.3 Datacenter Headroom

The best evaluation of a VM/spot allocator combination is when a datacenter is nearly at full utilization with just the VM workload: a datacenter overloaded with VMs has no room for spots, and a lightly loaded datacenter can trivially accommodate all spot requests. Accordingly, we tune the datacenter size to explore the behavior of CoSpot when dealing with a small, controlled amount of spare capacity.

First, we selected a single server type from the ILP solution that is large enough to accommodate any of the VM/spot types in the full workload. Then, for each randomly sampled workload (because samples differ in difficulty), and for each VM allocation policy (because policies differ in packing effectiveness), we used exhaustive simulation to find the smallest number of servers needed to handle all VM requests in the workload. **We define this to be the datacenter size that has 0% "headroom" or "spare capacity"**, as this is the smallest datacenter size that can handle all the VMs in the workload. Additionally, as it is unrealistic to expect even a heavily loaded datacenter to constantly operate at 100% utilization, we consider datacenters with an additional (within rounding) 1%, 2%, 3%, and 4% number of servers, which gives some spare capacity in the datacenter. This characterizes the behavior of CoSpot in the most challenging operating conditions, as the datacenter approaches full utilization.

4.4 VM Allocation Policies

As noted earlier, we wish to evaluate CoSpot against a wide variety of different VM/spot allocation/eviction policies. For VM allocation, our experiments use the 3 well-known, published, state-of-the-art policies described in Sec. 2: **UTIL**, **COS**, and **DPD**.

4.5 Spot Allocation Policies

The spot allocator does the same jobs as the VM allocator, except allocating spots instead of VMs onto servers. For our evaluation, we consider six policies for ranking which feasible server is most preferable:

***sUTIL*, *sCOS*, *and sDPD*.** are identical to UTIL, COS, and DPD, except they include spots when computing residual capacity and non-idle servers.

***vmaUTIL*, *vmaCOS*, *and vmaDPD*.** "vma" stands for "virtual machine avoiding". These are identical to sUTIL, sCOS, and sDPD, except they always prefer servers where no VM has been allocated to servers where there are VMs, instead of preferring non-idle servers. This is complementary to the preference for non-idle servers in UTIL, COS, and DPD, and captures the fact that spot allocation policies can be designed to try to avoid spot evictions.

4.6 Spot Eviction Policies

When eviction is necessary, the spot allocator needs a policy to choose which spots to evict. We consider three eviction suitability policies:

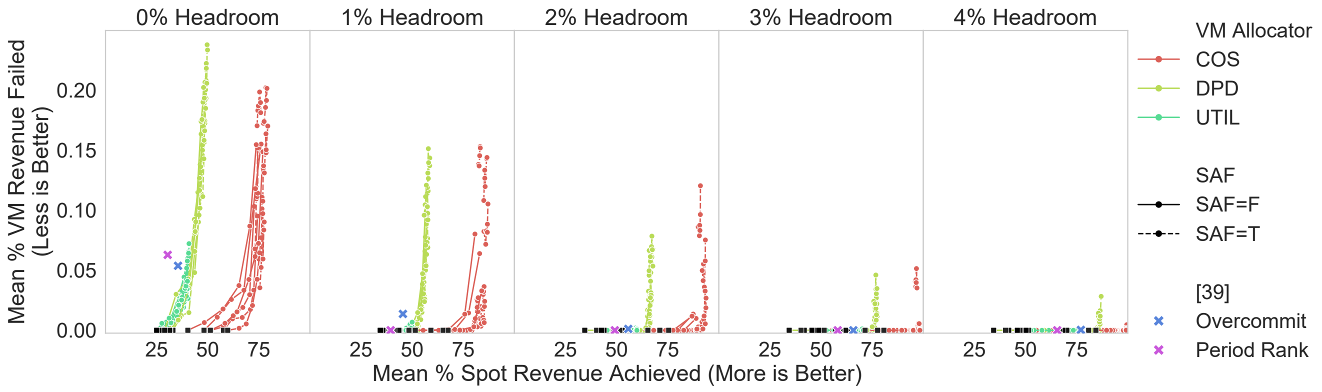


Figure 4: Spot Revenue Gained vs. VM Revenue Lost. These scatter plots summarize all 64,800 simulations. The x-axis is spot revenue achieved (for each of 5 different levels of datacenter headroom); the y-axis is VM revenue lost. Note that these revenues are given as percentages of the total *requested* in each sampled workload trace, ignoring evictions or resource fragmentation, so 100% might not actually be achievable. Each point represents a combination of VM allocator, spot allocator, and the tunable CoSpot parameters OTN and SAF, (216 points per headroom level), but for legibility, averaging the results over the 3 eviction policies (FIFO, LPML, LPO), the 2 sampling methods (Bernoulli, Batched), and the 10 samples per method. Colors indicate which VM allocator was used. Lines connect points that used the same VM/spot allocator combination, to highlight the trade-off as OTN varies: solid line for SAF false; dashed for SAF true. The baseline point ($OTN = 1$, $SAF = F$) for each allocator combination is plotted as a black square. Results for the two allocators from prior work [39] are plotted as Xs. The ideal result would be the bottom-right corner (100% spot revenue, with 0% VM revenue lost); a result Pareto-dominates any point above it and to its left. Note that over 76% of the simulations had 0 VM allocation failures, so most of the information in this plot is obscured in all the points on the x-axis. Nevertheless, some overall trends are apparent: CoSpot enables substantial increases in spot revenue, over both baseline as well as prior art, and usually with no loss in VM revenue; greater headroom allows greater gains with no loss in VM revenue; and tuning CoSpot parameters allows even more gains in spot revenue but with small losses in VM revenue.

***FIFO** (First-In-First-Out).* We use FIFO to represent a simple and fast eviction policy. The policy considers the removal of the spots in FIFO order: spots allocated earlier get removed first. For a given server, it tallies up the cores and RAM consumed by the spots until enough capacity has been freed for the VM allocation. The "score" for this set of evictions is the number of spots evicted; smaller is better.

***LPO** (Lifetime Prediction Oracle).* In contrast to FIFO, we use LPO to represent a heavyweight eviction policy that tries hard to optimize its decision. Fundamentally, eviction is difficult because the future is unknown. However, when simulating allocator designs, the full workload trace is available, so, in this case, the future *is* known. This policy models the (impossible-to-realize) gains from having perfect knowledge about spot lifetimes. This policy tries all subsets of spots running on a particular server. It computes the cores and RAM consumed by each subset, and if the collective resources consumed by the subset are sufficient such that the virtual machine request could be satisfied if these spots were evicted (feasibility), the future spot revenue lost by evicting this set is computed using the perfect knowledge from the workload trace of the spot end times. Finally, the subset with the least lost spot revenue is selected (suitability).

***LPML** (Lifetime Prediction Machine Learning).* is a realizable approximation to LPO. LPML is identical to LPO, except that it uses a neural network to predict a spot's remaining lifetime when considering evictions. We developed the network as a model trained on the workload trace to predict ending times for spots. The model contains a single hidden layer with ReLU activations. The training features for a spot request include workload information such as the spot type, the time and day of the week. We additionally augment the features with the lifetime of the previous completed VM from the same customer, since subsequent requests are often similar [25]. The model is trained to predict a hazard function for spot lifetimes, from which we compute the expected remaining lifetime when a spot is (or might be) evicted. The model was trained on the first three days of the full workload trace.

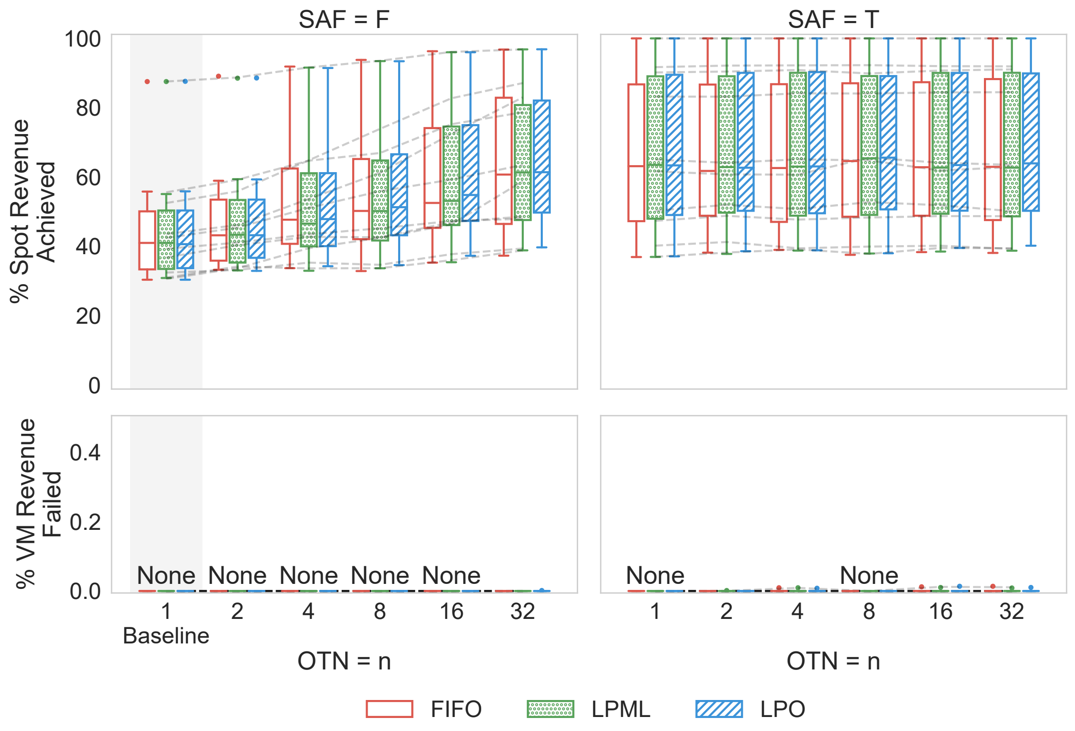


Figure 5: Results for a Typical VM/Spot-Allocator Combination (COS,sDPD) on a Datacenter with 2% Headroom, using Batch-Sampled Workloads. The y-axis is the percent of spot/VM revenue achieved (more is better) or failed (less is better) as a percentage of the total spot/VM revenue requested in the workload. Spot revenue is above; VM revenue is below. SAF and OTN are CoSpot's tunable parameters (Fig. 3 in Sec. 3). The left side of the graph is SAF false; the right side, SAF true. Within each side, the x-axis is OTN. The leftmost grouping is the baseline, when SAF is false and $OTN = 1$. The boxes show median and quartiles, the whiskers show min and max that are within 1.5 times the interquartile range from the lower and upper quartiles, and outliers beyond those points are plotted individually. The batch-sampled workloads vary markedly in difficulty, which makes the box plots very wide; the dashed grey lines connect data points from each sample workload, to better illustrate the effect of CoSpot's parameters on the different workloads. For example, one workload was much easier and generated all the spot revenue results at the top of each bar. By setting the SAF parameter to true, spot revenue improves significantly over the baseline, but with some losses in VM revenue. The choice of OTN has little effect when SAF is true. When SAF is set to false, tuning OTN can provide an increase to spot revenue without losses to VM revenue.

In our experiments, the LPML policy should not use its neural network until after the first three days of logging, because the network was trained on those days. As a workaround, the FIFO policy is used to make eviction decisions until the three day mark regardless of the specified eviction policy. After this point, the specified eviction policy kicks in. This degrades performance of both LPML and LPO (relative to FIFO), but provides an apples-to-apples comparison between the two.

5 EXPERIMENTAL RESULTS

With a large space of experimental parameters, it is hard to summarize all results. Yet a number of patterns were clear.

***Finding 1**: CoSpot increases spot revenue, over both baseline and prior art, at minimal or no loss to VM revenue.*

Agglomerating all 64,800 simulation runs, we achieved an average spot revenue gain of 35% and a maximum of 245%, compared to the baseline without cooperation ($OTN = 1$ and $SAF =$ false). Average VM revenue loss was 0.014%, and the maximum was 0.67%. Although the VM losses are tiny, tenants have much higher service level expectations for VMs, so we target *zero* VM revenue loss. If we include only simulation runs with zero VM revenue loss, average spot revenue gain drops to 34%, but the maximum is still 245%.

Note that these summary statistics, although concise, are somewhat contrived, e.g., a provider's allocator designs are

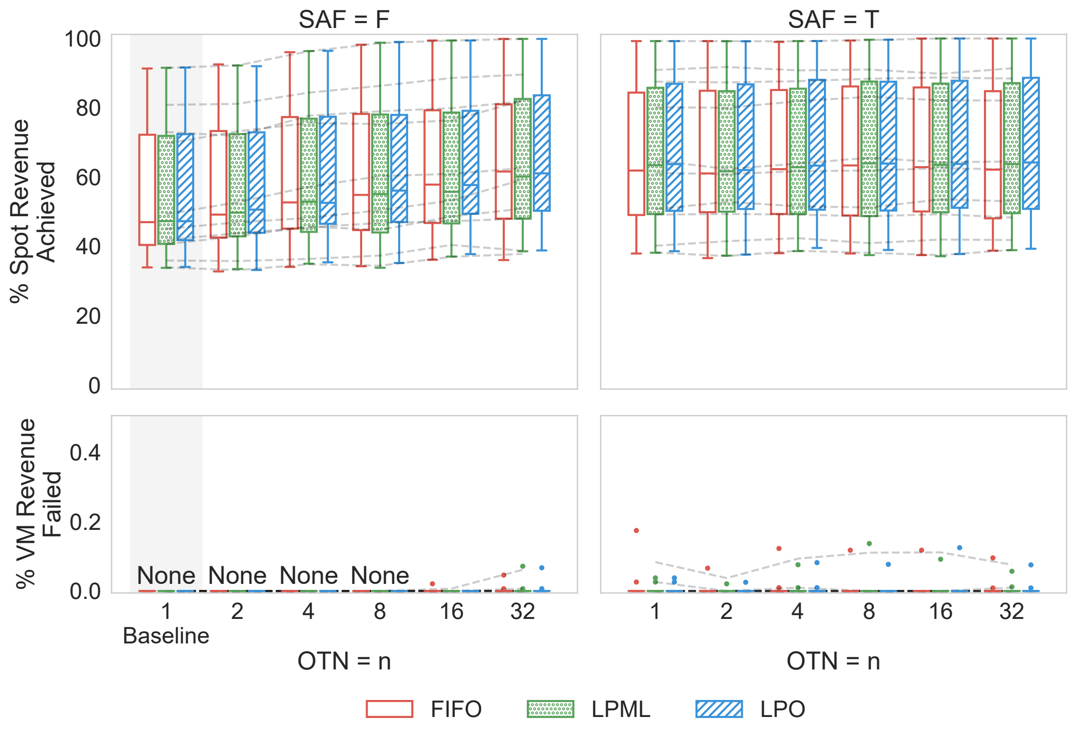


Figure 6: Results for the VM/Spot-Allocator Combination (COS,vmaDPD) on a Datacenter with 2% Headroom, using Batch-Sampled Workloads. This graph shows the same experimental parameters as Fig. 5, except that the spot allocation policy is changed from sDPD to vmaDPD. The vma policies try to reduce spot eviction without our cooperative framework, and do improve the baseline over Fig. 5. CoSpot further improves on this higher baseline.

not random, so it makes no sense to average over different policy combinations. A more nuanced summary would consider what gains are achievable under specific operating conditions by specific allocator combinations with some tuning of the CoSpot parameters. For example, with 1% headroom in the datacenter, we can achieve at least (i.e., the least improvement for any allocator combination) an average (over 10 batch-sampled workloads) spot revenue gain of 23% with an average VM revenue loss of 0% (i.e., no VM revenue lost on any of the 10 runs) by using parameters $OTN = 32$ and *SAF* false, for the allocator combination (COS,vmaUTIL,FIFO). The most-improving allocator combination with the 1% headroom datacenter and no VM revenue loss was (UTIL,sUTIL,LPML), which had an average spot revenue gain of 59%, by using parameters $OTN = 4$ and *SAF* true. However, there are 52 other allocator combinations at 1% headroom, and then 4 more headroom levels in our experiments, so we need to summarize broadly.

Fig. 4 is a scatter plot presenting the spot revenue gained vs. VM revenue lost in all experiments. (Details in figure caption.) It also shows how the spectrum of CoSpot results generally outperforms prior work [39]. Although the graph is very dense, some overall trends are visible: as noted already, that CoSpot enables substantial gains in spot revenue at minimal (usually zero) loss in VM revenue; that CoSpot does better when there is more headroom in the datacenter; and that tuning the CoSpot parameters allows controlling the VM revenue loss vs. spot revenue gain. We will explore these and other finds in more detail below.

***Finding 2**: SAF greatly improves spot revenue, but generally loses some VM revenue. Increasing OTN improves spot revenue, with gradual impact on VM revenue.*

Fig. 5 shows results for a typical VM/spot-allocator combination. (Details in caption.) Setting *SAF* to true greatly improved spot revenue, but for some workloads caused a minuscule loss of VM revenue. Varying OTN makes little difference when *SAF* is true — revenue gain levels off.

Fig. 5 also shows that when *SAF* is false, increasing OTN from 1 to 32 produces a gradual improvement in spot revenue, approaching that resulting from *SAF* being true, but with less (and controllable) loss of VM revenue.

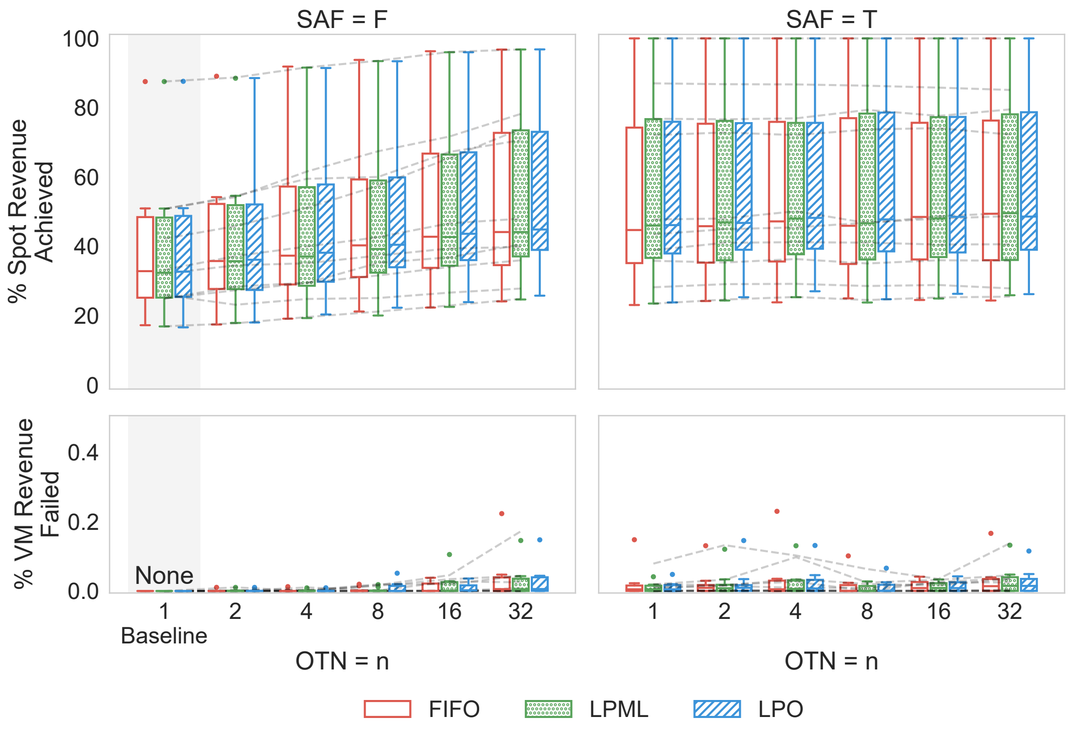


Figure 7: Results for VM/Spot-Allocator Combination (COS,sDPD) on a Datacenter with 0% Headroom, using Batch-Sampled Workloads. This graph shows the same experimental parameters as Fig. 5, except the datacenter is changed from 2% headroom to 0% headroom. With no headroom, even small perturbations to the VM allocator can produce loss in VM revenue. Comparing to Fig. 5 shows that a small amount of headroom in the datacenter gives CoSpot greater freedom to improve spot revenue.

***Finding 3**: The vma spot allocators start from a higher baseline, but still show improvement with CoSpot.*

Fig. 6 shows the same combination as Fig. 5, except the spot allocator has been changed to the vma (VM-Avoiding) version. The baseline performance is improved, which is not surprising, as the vma spot allocators are already "cooperating" in some sense with the VM allocator. However, CoSpot still yields further improvements in spot revenue.

***Finding 4**: Even FIFO works well as an eviction policy, with LPML and LPO working slightly better.*

This is not the focus of this paper, as we are not proposing or investigating spot eviction policies. However, our results suggest either that FIFO is good enough, or that a better eviction policy would need to be smarter than just finding the optimum set to evict at each point in time.

***Finding 5**: As datacenter headroom increases, there is greater benefit to greater cooperation.*

Fig. 7 shows results for the same combination as Fig. 5, except with 0% headroom instead of 2%. When the datacenter is already packed tightly with VMs, even minor perturbations to VM allocation can cause some loss in VM revenue, although potentially with large gains in spot revenue.

If we abuse the boxplots to group all 54 allocator combinations together in each column, we can generalize the comparison between Figs. 5 and 7, as shown in Fig. 8. Each grouping shows how CoSpot can better improve spot revenue and better avoid VM revenue loss as headroom increases. Even with a bit of spare capacity, CoSpot can increase spot revenue substantially, without detriment to VM revenue. Conversely, these results suggest that when a datacenter is fully- or overloaded with VMs, any cooperation with the spot allocator can (at least slightly) degrade VM revenue.

***Finding 6**: CoSpot performs less well on the burstier, batch-sampled workloads, but still improves spot revenue with minimal or no loss in VM revenue.*

We have emphasized the batch-sampled results, because they preserve the realistic burstiness of the original trace.

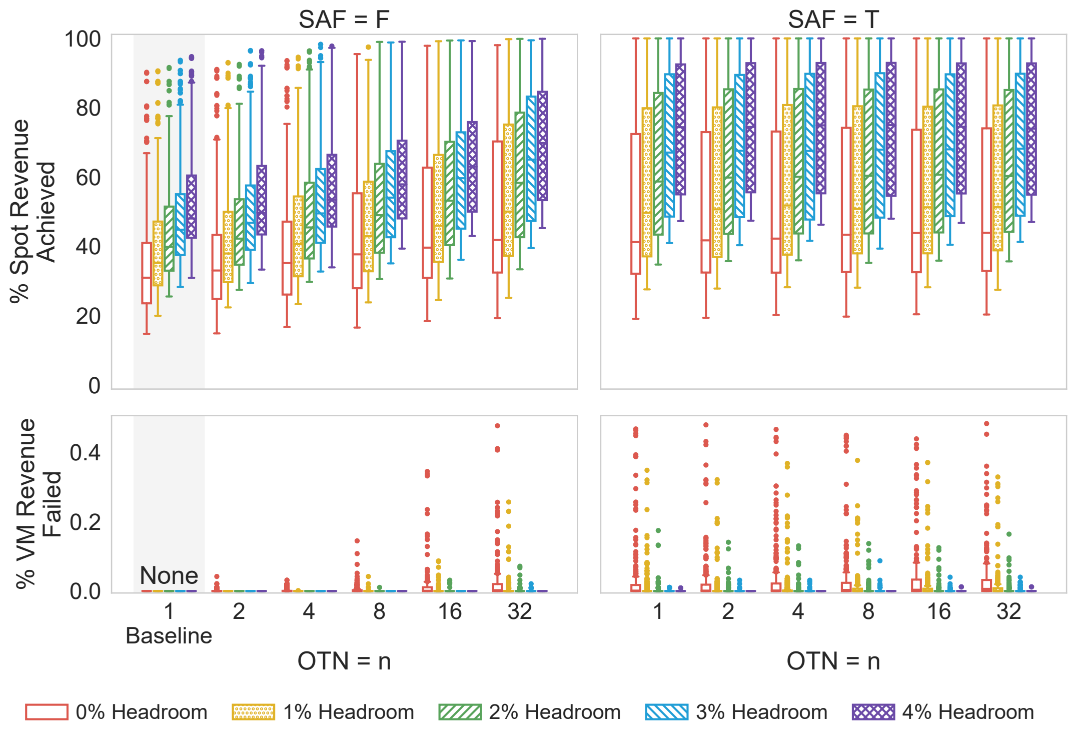


Figure 8: Combined Results Over All Allocator Combinations, using Batch-Sampled Workloads. This graph generalizes the comparison between Figs. 5 and 7. The graph groups results for all 54 VM/spot allocator combinations times 10 random workloads into each column. The groupings show how greater spare capacity enables greater gains in spot revenue and lower losses in VM revenue. However, even at 0% headroom, CoSpot can improve spot revenue, although with some VM revenue loss.

The Bernoulli-sampled workloads have less variability, resulting in cleaner graphs (omitted for space) and somewhat better results. Overall, restricting to only configurations with zero VM revenue loss (which eliminates some of the larger spot revenue gains), the average spot revenue gain over all configurations increased from 29% using batch-sampling to 37% using Bernoulli-sampling.

***Finding 7**: CoSpot's impact on allocation latency is negligible in nearly all cases, and always manageable.*

CoSpot had a 99th-percentile VM allocation latency of 24ms. There were a few outliers, due to the exhaustive subset search embedded in the LPML and LPO eviction policies, i.e., in cases where spot eviction is slow even without CoSpot. These can be eliminated by choosing a faster or more efficient eviction policy. E.g., using FIFO, the longest VM allocation latency was 18ms.

CoSpot has no effect on spot allocation latency — the spot allocator behaves exactly the same with or without CoSpot.

***Finding 8**: Using a realistic customer-oriented revenue function results in revenue losses of 0.5% at most.*

We repeated the above experiments with a revenue function that considered spots to be free if they were evicted within an hour of being launched, as in Amazon EC2 [4]. Spot revenue losses (compared to our original results) were within 0.5%, with a mean loss of 0.05%. Such realistic revenue functions can incentivize customers to use spots, with negligible revenue loss when used with CoSpot.

Takeaways: We have shown that CoSpot provides substantial spot revenue gains at minimal or no loss in VM revenue. These results are general across a wide range of VM and spot allocation policies, and robust to different operating conditions and workloads. For best results, a cloud provider should explore the parameter space through simulation with their own VM/spot allocators, server configurations, revenue policies, and historical workload traces to determine the optimal settings for *SAF* and *OTN*. In general, though, a cloud operator with very heavily loaded datacenters and demanding VM SLAs should stick to *SAF* = false and a small *OTN*, e.g., 2 or 4, and would still see gains in spot revenue; a cloud operator with more headroom might try SAF = true and/or larger values of OTN to gain much more spot revenue.

6 RELATED WORK

Deriving Greater Utility from Low Priority VMs. Much of the research on spot instances has been from the customer perspective, particularly centered around AWS's original bidding-based pricing model. This includes: (1) work that predicts future spot instance prices [1, 2, 9, 17, 29, 40, 56, 74, 75], (2) work that proposes bidding strategies for spot instances [9, 23, 34, 38, 40, 43, 52, 57, 60, 65, 70, 74, 75], and (3) work that aims to help users to migrate their work between VMs and spot instances, and avoid (or recover from) preemptions [30, 32, 33, 37, 44, 53, 58, 64, 67, 69]. After November 2017, AWS transitioned from bidding to a more stable (and generally higher [19]) fixed price for spot instances.

There have been proposals for how cloud providers may derive greater value from their spot instances, although these proposals do not directly concern spot allocation and preemption. Zhang et al. [72] suggest providers shut down idle machines in order to save energy, and study the interaction of energy saving and spot pricing. Shastri et al. [54] suggest cloud providers introduce spot instances with probabilistic guarantees on eviction rates. Dawoud et al. [12] and Ambati et al. [5] both propose a new VM class, which they call *elastic virtual machines* and *harvest virtual machines*, respectively, that grow and shrink as available resources fluctuate.

Scheduler Designs for High and Low Priority Workloads. For compute workloads beyond VMs, many approaches have been developed to enable low-priority workloads to share resources with high-priority jobs. Here, we focus on approaches used in large-scale compute clouds, such as those used in Facebook, Microsoft, and Google.

Centralized monolithic schedulers [20, 28, 71] are often used for HPC and big data workloads. In these workloads, jobs can remain pending in scheduling queues for some time. Schedulers are thereby able to optimize placement decisions by batch-scheduling multiple pending jobs together at the same time [18, 20, 28]. Demanding throughput and latency requirements in the IaaS context preclude these approaches from being used for VM and spot instance allocation [25].

Two-level schedulers [27, 62] handle course-grained resource management while letting application frameworks handle fine-grained scheduling duties. Distributed schedulers [45, 47, 49] use elaborate queue processing strategies at the target machine level. Hybrid schedulers [13, 14, 35] work by employing both centralized and distributed scheduling approaches. These two-level, distributed, and hybrid scheduling approaches have previously been avoided for VM and spot instance allocation, again because of the latency and throughput requirements of the IaaS layer [25]. In Omega [50], multiple schedulers leverage a shared state view of cluster resources to compete for placements "in a free-for-all manner." Facebook's Twine cluster management system [59] uses separate resource pools for different workload priorities, but moves hosts between pools dynamically based on demand. For spot allocation, this would introduce counter-productive fragmentation as each host could only run either spots or regular VMs.

For task-scheduling with large queues of pending requests, *scheduling priorities* are used to give precedence to high-priority workloads at the time of workload placement [7, 62, 63]. However, if tasks are long-lived, scheduling priority alone will not prevent low-priority workload from monopolizing resources and starving high-priority jobs. One solution is to reserve resources for known high-priority jobs in advance [11, 61]. Another is preemption: evict low-priority workload when the resources are needed. In Microsoft's Apollo [7], low-priority tasks "can be preempted or terminated if the server is under resource pressure," but no further details are provided. Google's Borg [63] also uses priority evictions. The paper briefly mentions that Borg tries to minimize "the number and priority of preempted tasks" when choosing machines for high-priority placements, but no details are public. Borg also has mechanisms to prevent tasks from the same job from getting evicted too often [61].

For VM allocation, there are few public details on mechanisms for jointly handling regular and preemptible VM instances. In general, many VMs allocators follow the OpenStack design [51], where candidate servers are first filtered using hard constraints and then scored according to certain soft preferences. Protean [25], the VM scheduler used in Microsoft Azure, also uses a filtering/scoring approach. Since Protean is reported to minimize the eviction rate of preemptible VMs, it presumably also uses a scoring rule that prefers placements not requiring evictions. However, this information is not publicly disclosed. Protean is similar to Sparrow [47], Apollo [7] and Omega [50] in that it uses multiple allocation agents over a shared inventory. CoSpot is entirely compatible with a multiple allocation agent approach, as each instance of an allocation agent can be architected to use CoSpot (with conflicts over the shared inventory being handled through normal mechanisms). For openly published allocation algorithms, Wu et al. [66] propose a framework for sharing servers between VMs and spot instances. In their approach, when VMs are allocated, the existence of spot jobs is simply ignored; evicting spots as needed. Conceptually, this corresponds to the baseline in CoSpot, with $OTN = 1$ and $SAF =$ false, but they are addressing an incomparable problem, in which servers can host at most one VM/spot at a time, and the goal is to minimize the overhead of migrating spots. As mentioned earlier, López García et al. [39] propose an allocator design that supports both VMs and spot instances and is directly comparable to CoSpot, and we compared our results against their two proposed scoring functions. However, we emphasize that CoSpot is not just a specific scoring function, but a framework to combine arbitrary VM and spot allocators.

The aforementioned prior works include both monolithic (a centralized system places both low and high priority workload) and polylithic (multiple schedulers operate in parallel on a shared inventory) architectures. CoSpot is capable of being employed in either architecture, in order to introduce explicit cooperation between schedulers. In the case of monolithic architectures, the division into VM vs. spot schedulers is logical rather than physical.

Some systems allow low-priority VMs to be *oversubscribed* on a host (for example, when the host is fully allocated, but actual host CPU usage remains low) [10]. As oversubscription is equivalent to logically adding more resources to certain hosts, it is compatible with CoSpot.

Resource Efficiency in Cloud Schedulers. There has been extensive prior work aimed at improving resource usage efficiency in cloud schedulers. These include techniques such as harvesting underutilized resources, reclaiming unused resources, resource heterogeneity, resource usage profiling and interference awareness, e.g., [5, 8, 10, 15, 16, 26, 31, 41, 42, 46, 63, 68, 73]. CoSpot can be considered in this category, as it attempts to minimize interference on spot instances from VMs while mitigating the ramifications of this minimization on VM SLA guarantees; however, we are the first to explore VM and spot allocator cooperation.

Takeaways: Prior work generally provides few details concerning allocation and preemption of spot workloads in large-scale compute clouds. There are no evaluations of trade-offs between VM and spot revenue for different allocation policies. There are also no existing published frameworks that enable explicit cooperation between VM and spot allocation policies or co-optimize VM and spot allocation to improve cloud operator revenue. CoSpot addresses these open issues.

7 CONCLUSION AND FUTURE WORK

We have introduced CoSpot, a lightweight framework for cooperative allocation of VMs and spot instances. CoSpot works with arbitrary, pre-existing VM and spot allocators. Experimentally, CoSpot performed well across 54 different allocator combinations, averaging a 34% gain in spot revenue across all experiments with zero loss in VM revenue. CoSpot did particularly well when the datacenter is near, but not at, full capacity. We are actively exploring ways to integrate CoSpot into our production VM allocators alongside potential new spot products. The simplicity of CoSpot allows us to expedite prototyping via simulation studies with our proprietary existing allocators and extended internal workloads.

For future work, there are many possibilities to extend or generalize CoSpot. For example, CoSpot's greater benefit as headroom increases suggests *dynamically* varying CoSpot parameters. The VM allocator always knows the datacenter's current load, so it would be easy to increase cooperation (e.g., increase OTN) when the load is very light, but let the VM allocator dominate as the datacenter approaches saturation. This should allow even greater spot revenue gains with even lower VM revenue losses. Similar in spirit, instead of a fixed N in OTN, the VM allocator could offer all servers whose score is within some epsilon of the best score. This idea requires that the ranking functions produce numerically meaningful scores (interval/ratio scale) instead of just a ranking (ordinal scale), but the advantage is that the number of choices would vary depending on how many are close to optimal. A natural generalization of CoSpot would be having more than two classes of VMs with different priorities. The CoSpot framework should extend in the obvious manner (higher-priority VM classes avoid evicting or offer multiple choices to the lower-priority allocator), but whether this would work well in practice would require extensive empirical evaluation with realistic workloads. A more open-ended direction for generalization is to explore richer cooperation between allocators. We have explicitly chosen to limit this cooperation (to leverage pre-existing allocators, to simplify the architecture), but greater cooperation could potentially better optimize the VM/spot revenues. Another important generalization would be to explore issues of job migration and reliability. For spot migration upon eviction, we would need a new revenue model, as migration incurs costs, but allows a spot to continue to generate revenue. For more general research on reliability and VM migration upon server failure, a necessary pre-condition would be the availability of realistic fault models and workloads. Finally, in the longer term, we believe there is considerable promise in more effective predictions of various quantities (e.g., revenue lost to eviction, future request arrival times), which would allow the allocators to make smarter decisions.

ACKNOWLEDGMENTS

We thank Nodir Kodirov for providing the codebase on which we built our simulation environment. We would also like to thank the anonymous referees and shepherd Zichen Xu for insightful suggestions. This work was supported by Huawei Technologies and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. 2013. Deconstructing Amazon EC2 Spot Instance Pricing.

ACM Trans. Econ. Comput. 1, 3, Article 16 (2013), 20 pages. https://doi.org/10.1145/2509413.2509416

[2] Sarah Alkharif, Kyungyong Lee, and Hyeokman Kim. 2018. Time-Series Analysis for Price Prediction of Opportunistic Cloud Computing Resources. In *7th Intl Conf on Emerging Databases*. 221–229.

[3] Amazon Web Services 2022. Amazon Compute Service Level Agreement. https://aws.amazon.com/compute/sla/. Accessed: 2022-09-23.

[4] Amazon Web Services 2022. Billing for interrupted Spot Instances — Amazon Elastic Compute Cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/billing-for-interrupted-spot-instances.html. Accessed: 2022-09-23.

[5] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symp on Operating Systems Design and Implementation (OSDI 20)*. 735–751.

[6] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. 2021. Generating Complex, Realistic Cloud Workloads Using Recurrent Neural Networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, 376–391. https://doi.org/10.1145/3477132.3483590

[7] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symp on Operating Systems Design and Implementation (OSDI 14)*. 285–300.

[8] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. 2014. Long-term SLOs for reclaimed cloud computing resources. In *Proceedings of the Fifth ACM Symposium on Cloud Computing (SoCC '14)*.

[9] Mohan Baruwal Chhetri, Markus Lumpe, Quoc Bao Vo, and Ryszard Kowalczyk. 2017. On Estimating Bids for Amazon EC2 Spot Instances Using Time Series Forecasting. In *2017 IEEE Intl Conf on Services Computing (SCC)*. 44–51. https://doi.org/10.1109/SCC.2017.14

[10] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *26th Symp on Operating Systems Principles (SOSP '17)*. 153–167. https://doi.org/10.1145/3132747.3132772

[11] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the Fifth ACM Symposium on Cloud Computing*. 1–14.

[12] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. 2012. Increasing Spot Instances Reliability Using Dynamic Scalability. In *2012 IEEE Fifth Intl Conf on Cloud Computing*. 959–961. https://doi.org/10.1109/CLOUD.2012.58

[13] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *7th Symp on Cloud Computing (SoCC '16)*.

[14] Pamela Delgado, Diego Didona, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid Datacenter Scheduling. In *USENIX Annual Technical Conference (ATC 15)*.

[15] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 77âĂŞ88. https://doi.org/10.1145/2451116.2451125

[16] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127âĂŞ144. https://doi.org/10.1145/2541940.2541941

[17] Shridhar G. Domanal and G. Ram Mohana Reddy. 2018. An efficient cost optimized scheduling for spot instances in heterogeneous cloud environment. *Future Generation Computer Systems* 84 (2018), 11–21. https://doi.org/10.1016/j.future.2018.02.003

[18] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *13th European Conf on Computer Systems (EuroSys'18)*.

[19] Gareth George, Rich Wolski, Chandra Krintz, and John Brevik. 2019. Analyzing AWS Spot Instance Pricing. In *2019 IEEE Intl Conf on Cloud Engineering (IC2E)*. 222–228. https://doi.org/10.1109/IC2E.2019.00036

[20] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symp on Operating Systems Design and Implementation (OSDI 16)*.

[21] Bhavesh N. Gohil, Sachin Gamit, and Dhiren R. Patel. 2021. Fair Fit—A Load Balance Aware VM Placement Algorithm in Cloud Data Centers. In *Advances in Communication and Computational Technology*. Springer, 437–451.

[22] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *2014 ACM Conf on SIGCOMM (SIGCOMM '14)*. 455–466. https://doi.org/10.1145/2619239.2626334

[23] Weichao Guo, Kang Chen, Yongwei Wu, and Weimin Zheng. 2015. Bidding for Highly Available Services with Low Price in Spot Instance Market. In *24th Intl Symp on High-Performance Parallel and Distributed Computing (HPDC '15)*. 191–202. https://doi.org/10.1145/2749246.2749259

[24] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[25] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symp on Operating Systems Design and Implementation (OSDI 20)*. 845–861.

[26] Jaeung Han, Seungheun Jeon, Young-ri Choi, and Jaehyuk Huh. 2016. Interference Management for Distributed Parallel Applications in Consolidated Clusters. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 443âĂŞ456. https://doi.org/10.1145/2872362.2872388

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symp on Networked Systems Design and Implementation (NSDI 11)*.

[28] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2019. Quincy: Fair Scheduling for Distrubuted Computing Clusters. In *27th ACM Symp on Operating System Principles (SOSP'19)*.

[29] Bahman Javadi, Ruppa K. Thulasiramy, and Rajkumar Buyya. 2011. Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In *2011 Fourth IEEE Intl Conf on Utility and Cloud Computing*. 219–228. https://doi.org/10.1109/UCC.2011.37

[30] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. 2016. Smart Spot Instances for the Supercloud. In *3rd*

Workshop on CrossCloud Infrastructures & Platforms (CrossCloud '16). Article 5, 6 pages. https://doi.org/10.1145/2904111.2904114

[31] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. 2020. Improving Resource Utilization by Timely Fine-Grained Scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 20, 16 pages. https://doi.org/10.1145/3342195.3387551

[32] Daeyong Jung, JongBeom Lim, and Heonchang Yu. 2014. Estimated Interval-Based Checkpointing (EIC) on Spot Instances in Cloud Computing. *J of Applied Mathematics* (5 2014). https://doi.org/10.1155/2014/217547

[33] JCS Kadupitiya, Vikram Jadhao, and Prateek Sharma. 2020. Modeling The Temporally Constrained Preemptions of Transient Cloud VMs. In *29th Intl Symp on High-Performance Parallel and Distributed Computing (HPDC '20)*. 41–52. https://doi.org/10.1145/3369583.3392671

[34] BogumiÅĆ KamiÅĎski and PrzemysÅĆaw Szufel. 2015. On optimization of simulation execution on Amazon EC2 spot market. *Simulation Modelling Practice and Theory* 58 (2015), 172–187. https://doi.org/10.1016/j.simpat.2015.05.008 Special issue on Cloud Simulation.

[35] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *USENIX Annual Technical Conference (ATC 15)*.

[36] Xiaodi Ke, Cong Guo, Siqi Ji, Shane Bergsma, Zhenhua Hu, and Lei Guo. 2021. Fundy: A Scalable and Extensible Resource Manager for Cloud Resources. In *IEEE Intl Conf on Cloud Computing*.

[37] Sunirmal Khatua and Nandini Mukherjee. 2013. Application-Centric Resource Provisioning for Amazon EC2 Spot Instances. In *Euro-Par 2013 Parallel Processing*. Springer Berlin Heidelberg, 267–278.

[38] Mikhail Khodak, Liang Zheng, Andrew S. Lan, Carlee Joe-Wong, and Mung Chiang. 2018. Learning Cloud Dynamics to Optimize Spot Instance Bidding Strategies. In *IEEE INFOCOM 2018 - IEEE Conf on Computer Communications*. 2762–2770. https://doi.org/10.1109/INFOCOM.2018.8486291

[39] Alvaro López García, Enol Fernández del Castillo, and Isabel Campos Plasencia. 2019. An efficient cloud scheduler design supporting preemptible instances. *Future Generation Computer Systems* 95 (2019), 68–78. https://doi.org/10.1016/j.future.2018.12.057

[40] Markus Lumpe, Mohan Baruwal Chhetri, Quoc Bao Vo, and Ryszard Kowalcyk. 2017. On Estimating Minimum Bids for Amazon EC2 Spot Instances. In *2017 17th IEEE/ACM Intl Symp on Cluster, Cloud and Grid Computing (CCGRID)*. 391–400. https://doi.org/10.1109/CCGRID.2017.76

[41] Jason Mars and Lingjia Tang. 2013. Whare-Map: Heterogeneity in "Homogeneous" Warehouse-Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 619âĂŞ630. https://doi.org/10.1145/2485922.2485975

[42] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 248âĂŞ259. https://doi.org/10.1145/2155620.2155650

[43] Michele Mazzucco and Marlon Dumas. 2011. Achieving Performance and Availability Guarantees with Spot Instances. In *2011 IEEE Intl Conf on High Performance Computing and Communications*. 296–303. https://doi.org/10.1109/HPCC.2011.46

[44] Ishai Menache, Ohad Shamir, and Navendu Jain. 2014. On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud. In *11th Intl Conf on Autonomic Computing (ICAC 14)*. 177–187.

[45] Phillip Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symp on Operating Systems Design and Implementation (OSDI 18)*.

[46] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 184âĂŞ200. https://doi.org/10.1145/3132747.3132766

[47] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *24th ACM Symp on Operating System Principles (SOSP'13)*.

[48] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. 2011. Heuristics for Vector Bin Packing. https://www.microsoft.com/en-us/research/wp-content/uploads/2011/01/VBPackingESA11.pdf.

[49] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient Queue Management for Cluster Scheduling. In *11th European Conf on Computer Systems (EuroSys'16)*.

[50] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conf on Computer Systems (EuroSys)*. 351–364.

[51] Omar Sefraoui, Mohammed Aissaoui, and M. Eleuldj. 2012. OpenStack: Toward an Open-source Solution for Cloud Computing. *Intl J of Computer Applications* 55 (2012), 38–42.

[52] Prateek Sharma, David Irwin, and Prashant Shenoy. 2016. How Not to Bid the Cloud. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

[53] Supreeth Shastri and David Irwin. 2017. HotSpot: Automated Server Hopping in Cloud Spot Markets. In *2017 Symp on Cloud Computing (SoCC '17)*. 493–505. https://doi.org/10.1145/3127479.3132017

[54] Supreeth Shastri, Amr Rizk, and David Irwin. 2016. Transient Guarantees: Maximizing the Value of Idle Cloud Capacity. In *SC '16: Intl Conf for High Performance Computing, Networking, Storage and Analysis*. 992–1002. https://doi.org/10.1109/SC.2016.84

[55] Jiyuan Shi, Fang Dong, Jinghui Zhang, Junzhou Luo, and Ding Ding. 2015. Two-Phase Online Virtual Machine Placement in Heterogeneous Cloud Data Center. In *2015 IEEE Intl Conf on Systems, Man, and Cybernetics*. IEEE Press, 1369–1374. https://doi.org/10.1109/SMC.2015.243

[56] Vivek Kumar Singh and Kaushik Dutta. 2015. Dynamic Price Prediction for Amazon Spot Instances. In *2015 48th Hawaii Intl Conf on System Sciences*. 1513–1520. https://doi.org/10.1109/HICSS.2015.184

[57] Yang Song, Murtaza Zafer, and Kang-Won Lee. 2012. Optimal bidding in spot instance market. In *2012 IEEE INFOCOM*. 190–198. https://doi.org/10.1109/INFCOM.2012.6195567

[58] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. 2015. SpotOn: A Batch Computing Service for the Spot Market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. 329–341. https://doi.org/10.1145/2806777.2806851

[59] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System

for Shared Infrastructure. In *14th USENIX Symp on Operating Systems Design and Implementation (OSDI 20)*. 787–803.

[60] ShaoJie Tang, Jing Yuan, and Xiang-Yang Li. 2012. Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In *2012 IEEE Fifth Intl Conf on Cloud Computing*. 91–98. https://doi.org/10.1109/CLOUD.2012.134

[61] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *EuroSys'20*.

[62] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *4th annual Symp on Cloud Computing*. 1–16.

[63] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *European Conf on Computer Systems (EuroSys)*.

[64] William Voorsluys, Saurabh Kumar Garg, and Rajkumar Buyya. 2011. Provisioning Spot Market Cloud Resources to Create Cost-Effective Virtual Clusters. In *Algorithms and Architectures for Parallel Processing*. Springer, 395–408.

[65] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. 2017. Probabilistic Guarantees of Execution Duration for Amazon Spot Instances. In *Intl Conf for High Performance Computing, Networking, Storage and Analysis (SC '17)*. Article 18, 11 pages. https://doi.org/10.1145/3126908.3126953

[66] Xiaohu Wu, Francesco De Pellegrini, Guanyu Gao, and Giuliano Casale. 2019. A Framework for Allocating Server Time to Spot and On-Demand Services in Cloud Computing. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 4, 4, Article 20 (2019), 31 pages. https://doi.org/10.1145/3366682

[67] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. 2016. Blending on-demand and spot instances to lower costs for in-memory storage. In *IEEE INFOCOM 2016 - The 35th Annual IEEE Intl Conf on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524348

[68] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 607âĂŞ618. https://doi.org/10.1145/2485922.2485974

[69] Sangho Yi, Artur Andrzejak, and Derrick Kondo. 2012. Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. *IEEE Trans. on Services Computing* 5, 4 (2012), 512–524. https://doi.org/10.1109/TSC.2011.44

[70] Murtaza Zafer, Yang Song, and Kang-Won Lee. 2012. Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs. In *2012 IEEE Fifth Intl Conf on Cloud Computing*. 75–82. https://doi.org/10.1109/CLOUD.2012.59

[71] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *5th European Conf on Computer Systems (EuroSys'10)*.

[72] Qi Zhang, Quanyan Zhu, and Raouf Boutaba. 2011. Dynamic Resource Allocation for Spot Markets in Cloud Computing Environments. In *2011 Fourth IEEE Intl Conf on Utility and Cloud Computing*. 178–185. https://doi.org/10.1109/UCC.2011.33

[73] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 755âĂŞ770.

[74] Han Zhao, Miao Pan, Xinxin Liu, Xiaolin Li, and Yuguang Fang. 2012. Optimal Resource Rental Planning for Elastic Applications in Cloud Market. In *2012 IEEE 26th Intl Parallel and Distributed Processing Symp.* 808–819. https://doi.org/10.1109/IPDPS.2012.77

[75] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. 2015. How to Bid the Cloud. In *2015 ACM Conf on Special Interest Group on Data Communication (SIGCOMM '15)*. 71–84. https://doi.org/10.1145/2785956.2787473