SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud

Yawen Wang
Stanford University
yawenw@stanford.edu

Kapil Arya
Microsoft Research
Kapil.Arya@microsoft.com

Marios Kogias
Microsoft Research
t-makog@microsoft.com

Manohar Vanga
Nokia Bell Labs, Germany
manohar.vanga@nokia-bell-labs.com

Aditya Bhandari
Microsoft
adityabh@microsoft.com

Neeraja J. Yadwadkar
Stanford University
neeraja@cs.stanford.edu

Siddhartha Sen
Microsoft Research
sidsen@microsoft.com

Sameh Elnikety
Microsoft Research
samehe@microsoft.com

Christos Kozyrakis
Stanford University
kozyraki@stanford.edu

Ricardo Bianchini
Microsoft Research
ricardob@microsoft.com

Abstract

We can increase the efficiency of public cloud datacenters by harvesting allocated but temporarily idling CPU cores from customer virtual machines (VMs) to run batch or analytics workloads. Even small efficiency gains translate into substantial savings, since provisioning and operating a datacenter costs hundreds of millions of dollars per year. The main challenge is to harvest idle cores with little or no impact on customer VMs, which could be running latency-sensitive services and are essentially black-boxes to the cloud provider.

We introduce ElasticVM, a new VM type that can run batch workloads cheaply using mainly harvested cores. We also propose SmartHarvest, a system that dynamically manages the number of cores available to ElasticVMs in each fine-grained time window. SmartHarvest uses online learning to predict the core demand of primary, customer VMs and compute the number of cores that can be safely harvested. Our results show that SmartHarvest can harvest a significant amount of CPU resources without increasing the 99th-percentile tail latency of latency-critical primary workloads by more than 10%. Unlike static harvesting techniques that rely on offline profiling, SmartHarvest is robust to different primary workloads, batch workloads, and load changes. Finally, we show that the online learning in SmartHarvest is complementary to systems optimizations for VM management.

ACM Reference Format:
Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456225

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456225

1 Introduction

Motivation. Most datacenters continue to operate at low resource utilization [29, 36, 38, 45, 53, 55, 71] despite many efforts to rightsize applications [11, 60, 66] and build schedulers, load balancers, and cluster managers with improved bin-packing capabilities [21, 26, 28, 29, 34, 35, 54, 61]. This problem is even more acute in public clouds as customers often oversize their virtual machines (VMs) to be able to handle load spikes, failover scenarios, and growth in demand for their services [26]. Even when the cloud platform can dynamically scale the number of VMs, customers often leave plenty of spare capacity in case load increases faster than the platform can react. This overprovisioning prevents degradations in user experience (e.g., an excessive increase in service tail latency [27, 42, 44, 56]) but also massively underutilizes the platform's resources.

To make matters worse, public clouds differ from other datacenters in that VMs are almost always black boxes to

the platform. Cloud customers are rarely willing to provide application-level performance information about their workloads (via explicit interfaces) or allow the platform to deeply inspect their VMs (via event or system call counters). In the absence of application-level information, platforms must (1) assume any "regular" VM may be latency-sensitive (i.e., any VM type not explicitly identified as lower priority should be treated as latency-sensitive); and (2) rely on lower-level proxies for performance, such as CPU utilization, to estimate VMs' resource requirements. Moreover, using profiling to estimate how a workload will behave in the long term is typically infeasible in the cloud: offline profiling requires inputs that are often unavailable until VMs run in production, whereas one-time online profiling requires knowing when each black-box VM has shown representative behavior.

These differences mean that techniques for increasing datacenter utilization are often not directly applicable to public clouds, including several recent proposals to harvest spare resources from a latency-sensitive service and use them for batch workloads (e.g., machine learning training, batch data analytics, Web crawling) [36, 37, 44, 45, 61, 71]. For instance, PerfISO [36] proposed leaving a fixed number of cores idle, called an *idle buffer*, to absorb bursts in CPU usage by the latency-sensitive service while harvesting the rest of the idle cores. Heracles [45] used a feedback-based controller to reassign resources based on service tail latency. In the public cloud context, we could create best-effort (low-priority) VMs that harvest resources from regular (high-priority) VMs for batch processing. Unfortunately, the prior works require extensive offline workload profiling (e.g., PerfISO), rely on application-level performance metrics (e.g., Heracles), and/or could threaten the tail performance of latency-sensitive services (e.g., proposals that do not reserve any buffer resources). Applying the ideas from these works would require keeping the *same* substantial number of resources idle in *every* server, so that even the most bursty regular VMs could quickly spike into them until the system can take enough resources away from the low-priority VMs. This would clearly be inefficient.

Our work: SmartHarvest and ElasticVM. We propose SmartHarvest, a system that increases cloud platform utilization by *dynamically* harvesting spare resources from regular (aka "primary") VMs for a co-located ElasticVM. Spare resources are *resources allocated to the primary VMs but are temporarily idling*. The ElasticVM is a new type of low-priority VM allocated with a minimum set of resources. It expands and contracts beyond this minimum (i.e., receives more or fewer physical resources) based on the availability of spare resources. We focus on harvesting idle CPU cores, as cores are typically the scarcest and most expensive resource of cloud servers. To take advantage of additional cores, the workload running in the ElasticVM must have enough software threads. When physical cores are taken away from it, the virtual CPUs simply multiplex on the remaining physical cores. Providing service-level objectives (SLOs) for workloads running on harvested resources [13, 22] and building software frameworks that run well on harvested resources [13, 23, 57] are orthogonal challenges to the one we target, which is maximizing the amount of harvested cores.

Instead of keeping a large number of idle cores in every server, SmartHarvest protects the performance of primary VMs by continuously *monitoring* and *predicting* the resources they will need in the near future in each server, and assigning physical cores accordingly. This *online learning approach* allows SmartHarvest to dynamically maximize the cores assigned to the ElasticVM, while minimizing the negative impact on the co-located primary VMs.

SmartHarvest's predictions are based solely on observed core utilizations and are applicable across a wide range of workloads without the need for application-level metrics. Given recent core utilization measurements, SmartHarvest predicts the next peak number of physical cores required by primary workloads. SmartHarvest formulates the prediction as a *multi-class classification* problem [2, 7, 17], where each feasible number of cores is a separate class. The classification is *cost-sensitive*: rather than simply labeling the correct class, it uses a cost function to penalize underpredictions more severely than overpredictions. Underpredicting primary CPU usage leads to excessive harvesting at the cost of performance loss for the primary workloads. On the other hand, overpredicting primary CPU usage reduces the amount of harvesting but does not impact performance of the primaries. Cost-sensitive, multi-class classification is lightweight and requires no offline training data. SmartHarvest also uses a *safeguard* mechanism that temporarily constrains harvesting when frequent underpredictions occur.

Implementation and results. We implement SmartHarvest for servers running the Hyper-V hypervisor. We create an agent that runs in user-space in the host OS (root partition) and manages core assignments. The agent uses a single core to frequently monitor core utilization. At the end of a learning window, the agent predicts the peak core utilization for the next window and uses Hyper-V's cpugroup mechanism to apportion the cores for that window. We tuned the learning window conservatively to preserve the performance of sensitive primary workloads with sub-millisecond tail latency requirements. Since it may take a relatively long time for a cpugroup change to take effect (on the order of 10ms), we also explore an implementation that reassigns cores faster. This implementation relies on changes we make to Hyper-V to effect the cpugroup change faster (on the order of 100μs) via interprocessor interrupts (IPIs).

We evaluate SmartHarvest using four latency-sensitive primary workloads at various loads: a real Web search service, Memcached, and two representative TailBench benchmarks [41]. In the ElasticVM, we run two batch applications,

a real machine learning workload and TeraSort, or a synthetic application that consumes as many cores as it is given. We explore the sensitivity of our results to a variety of learning and system parameters, including the learning window length, the cost function, and how fast cores are reassigned (default cpugroups versus IPI-based mechanism).

Our results demonstrate that SmartHarvest is effective at harvesting cores for batch workloads while protecting the performance of primary VMs. In all our experiments, SmartHarvest limits the increase in the tail latency (P99) of the primary workloads to 10% or less, without using any offline profiling. While tuning SmartHarvest for primary applications with sub-millisecond latency requirements reduces the number of cores harvested for less sensitive workloads, having a single SmartHarvest configuration that works across multiple types of primary workloads is a major advantage. In constrast, there is no single setting of a fixed buffer policy like the one used in PerfISO [36] that works well and safely across different primary workloads, offered loads, and batch workloads running on harvested resources.

Finally, we show that, while online learning is more critical for slower core reassignment mechanisms, it still provides significant benefits with the faster, IPI-based reassignment mechanisms. Online learning avoids frequent incorrect reassignments that impact performance even if they are fast. This result illustrates a fundamental synergy between machine learning (ML) and systems mechanisms in the nascent area of "ML for Systems".

Contributions. In summary, our main contributions are:

1. A novel system and VM type for dynamically harvesting cores that have been allocated to primary VMs but have been temporarily idling in public cloud servers.
2. A robust and general online learning approach for continuously predicting the use of cores in black-box primary VMs.
3. An evaluation of the benefits of harvesting with online learning using real and synthetic workloads across a range of scenarios.
4. An illustration of the synergy between online learning and system optimizations in the "ML for systems" space.

2 Background & Motivation

Resource harvesting. A common approach for increasing server utilization has been to harvest allocated but idling resources from a latency-sensitive service and use them for batch workloads. Spare cores have received the most attention, as CPUs are the most expensive resource and potentially frequent changes in their idleness makes harvesting quite difficult. Prior work has focused on two main aspects:

1. Protecting the performance of the latency-sensitive service in the face of relatively slow mechanisms for core reassignment, e.g. [36]. This work has relied on reserving an idle buffer of cores for the service to spike into, while cores can be taken away from the batch workload.
2. Improving core reassignment mechanisms [53], which at the extreme could obviate the need for an idle buffer. However, this work has relied on knowledge of service characteristics and their communication (e.g., number of cores needed to handle an incoming message).

Along the same lines but in a virtualized environment, we focus on harvesting idle cores from (potentially latency-sensitive) primary VMs for a co-located lower-priority ElasticVM. The primary VMs can be from many different customers, have a variety of configuration characteristics, dynamically change behavioral patterns, and arrive/depart at any time. Prior work [26] has shown that public cloud platforms have an abundance of allocated but idle cores, as a large percentage of VMs exhibit low CPU utilization. For example, 75% of VMs exhibit 25% or lower average CPU utilization over their lifetimes [26]. Our harvesting techniques retain a dynamic buffer of idle cores, but we also explore faster cross-VM core reassignment mechanisms. The *key challenge* we address is maximizing core harvesting while preventing degradation to the performance of co-located primary VMs, under the practical constraints of public clouds.

An earlier paper [13] solves the easier problem of harvesting "unallocated" cores, i.e. cores that have not been rented to any VMs. When only unallocated cores are harvested, harvesting does not impact the performance of primary VMs any more than co-locating an additional primary VM would. Importantly, this prior work describes how to modify existing cluster scheduling frameworks to take advantage of harvested resources for batch workloads. ElasticVMs can leverage the same frameworks without modification.

Finally, note that ElasticVMs are substantially different from spot VMs [12, 15, 25]. Spot VMs only consume a *fixed* number of *unallocated* resources and are evicted/killed as soon as the cloud platform needs any of those resources for primary VMs. They cannot borrow unused physical cores that have been allocated to other VMs. In contrast, ElasticVMs are capable of *dynamically* harvesting *allocated but idling* cores, and thus need to manage core assignments more carefully. Additionally, an ElasticVM is not evicted when the platform needs harvested cores back as it has a minimum size and can grow or shrink dynamically from this minimum. To receive the same average number of cores as an ElasticVM, spot VMs would incur many more expensive and intrusive evictions as they cannot be shrunk (they are evicted instead).

Online adaptation. A key drawback of techniques that use an idle buffer of cores is that the ideal buffer size may vary *spatially*, from server to server, and *temporally*, as a result of behavior changes in one or more co-located primary VMs. To overcome this drawback, we need to dynamically select the best buffer size over time, and do so independently at each

server. Simple history-based schemes, like Exponentially Weighted Moving Average (EWMA), are commonly used to estimate future information based on recent data. However, they are ineffective at foreseeing sharp CPU usage bursts. This motivates us to adopt an *online learning approach [18]* that learns workload CPU usage patterns and sizes the buffer adaptively. The *key challenge* is devising lightweight learning techniques that can accurately predict core usage at a fine temporal grain with the aim of maximizing harvesting while protecting the performance of primary VMs. Accurate prediction is also helpful with the design of a robust batch service on top of harvested resources [13].

3 SmartHarvest

We now describe the principles that guide our design, the concept of ElasticVMs, the SmartHarvest architecture and operation, and its online learning component.

3.1 Design Principles

We design SmartHarvest based on the following guidelines. First, harvesting should not negatively affect the performance of (black-box) primary VMs by retaining any cores that they need. Second, we want to dynamically harvest as many idle cores from primary VMs as possible in order to allow the batch workloads to make maximum progress. These two properties are difficult to achieve in the presence of arbitrary workload mixes and arbitrary load variations in the primary workloads. Thus, the third desirable property is that SmartHarvest should support dynamic core usage monitoring and fast core reassignments. Moreover, reassignments should be of full cores (vs hyperthreads) to prevent a batch workload from sharing a core with a primary workload, as this causes higher tail latencies [68]. Finally, for SmartHarvest to be practical, it should work consistently well for any primary and batch workloads, without the need for profiling or application-level metrics, and without relying on application-specific characteristics. These design principles allow us to address the challenges raised in Section 2.

3.2 ElasticVMs

We create a new type of VM for batch workloads, called an ElasticVM. Each ElasticVM has a "minimum" physical resource allocation, e.g. 1 core, 8GB of memory, and 10GB of SSD space. However, its actual number of assigned physical cores grows beyond this minimum, whenever there are unallocated cores (cores that have not been committed to any primary VM) and/or idle cores that have been harvested from primary VMs on the same server. For this paper, the memory and SSD allocations stay fixed throughout the execution of the ElasticVM. Figure 1 shows an example core assignment, where the ElasticVM has a minimum size of 1 physical core, but also receives 2 unallocated cores and harvests 3 idle cores from primary VMs.

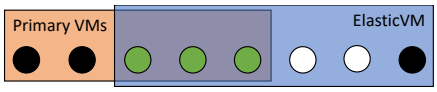


Figure 1. Core assignment in a server: the blue rectangle represents the ElasticVM, whereas the orange rectangle represents one or more primary VMs. The circles represent cores, with dark fill corresponding to allocated cores and white fill to unallocated cores that have been assigned to the ElasticVM. The green cores have been allocated to primary VMs but are currently being harvested.

Users can deploy ElasticVMs to the cloud platform in the same way as they deploy any other VM. The platform treats an ElasticVM as if it has a fixed size equal to its minimum resources, except for an agent running on each server that is responsible for managing the VM core assignments transparently to the rest of the platform.

We assume the ElasticVM is core-hungry and has enough work to utilize all cores assigned to it; thus we do not monitor or predict its core needs, and focus instead on harvesting as many primary VM cores for it as we can. The maximum number of physical cores an ElasticVM can reach is the total number of physical cores in the server. Thus, we configure an ElasticVM to have as many *virtual* cores as the total number of *physical* cores in a server, regardless of how many physical cores it is actually assigned. This obviates the need for changes to the software that runs on the ElasticVM: if the ElasticVM receives fewer physical cores than virtual cores, the hypervisor will simply multiplex any active virtual cores onto the assigned physical cores.

The software within the ElasticVM may adapt its parallelism to the number of available cores (e.g., via thread pools), or schedule computations on each ElasticVM based on this number. For example, cluster scheduling frameworks like Apache YARN [58] or Kubernetes [21] can be adapted to take advantage of harvested cores transparently to the workloads they schedule [71]. Our techniques are oblivious to what exactly runs on ElasticVMs.

We refer to ElasticVM as being lower priority than primary VMs for two reasons: (1) a primary VM has paid for, and thus has priority over, its allocated cores—the ElasticVM can only use an allocated core if the primary VM does not currently need it; and (2) the ElasticVM may be killed if the cloud provider needs even its minimum resources for a primary VM. Pricing for ElasticVMs is beyond the scope of this work, but can follow well-known strategies for spot-instance (low-priority VMs) or serverless VMs (pay for what you use). In addition to being offered as a low-cost VM type to run workloads without strict QoS requirements from external cloud tenants, ElasticVMs may also be used for internal batch services over which the cloud provider has full control.

The provider can define a family of ElasticVMs with different configurations. For simplicity, we assume a single ElasticVM type and focus only on maximizing the harvesting of cores from arbitrary primary VMs to a (single) ElasticVM on

the server. Additionally, we assume there are no unallocated cores in the server. Managing unallocated cores is easier, as they only have to be relinquished by the ElasticVM when they are allocated to a newly arriving primary VM [13].

We considered allowing ElasticVMs to dynamically harvest spare memory as well, but doing so is substantially more complex due to the overheads involved. For example, reallocating a physical memory page from one VM to another involves CPU work for bookkeeping and updating several system tables, as well as zeroing the page for security. A primary VM may need its harvested memory pages back, but reclaiming the pages may involve costly IOs. We focus on core harvesting in this paper, but our online learning approach to predicting resource usage should be generally applicable to harvesting other types of resources.

3.3 SmartHarvest Architecture & Operation

Functionality. SmartHarvest improves server utilization by taking advantage of idle cores allocated to primary VMs to run low-priority workloads in an ElasticVM. To do this with minimal impact on the performance of the primary VMs, SmartHarvest maintains an idle buffer of cores that is ready to immediately absorb load increases in the primary VMs. The cores in the idle buffer are allocated but not currently used by any primary VM. SmartHarvest tries to only reserve as many idle cores as needed, so it periodically predicts the peak number of cores required by the primary VMs at a fine, sub-second time-granularity. Based on this prediction, it reassigns cores. Instances of SmartHarvest on different servers operate entirely independently.

Architecture. SmartHarvest comprises two main components: an ElasticVM and a software agent we call EVMAgent. The agent is responsible for managing the cores assigned to the ElasticVM, beyond its minimum size, using calls to the hypervisor. The agent is also responsible for dynamically sizing the idle buffer, by executing the online learning algorithm (Section 3.4) that predicts the number of needed primary cores. For comparison, the agent can also be configured to manage a fixed-size buffer of idle cores without any predictions, as proposed by PerfIso [36].

Operation. Algorithm 1 summarizes the operation of the EVMAgent. The agent splits time into "learning windows". During each window (lines 4–10), the agent frequently polls the hypervisor for the number of busy primary cores and records this data for the learning algorithm to use. A primary core is conservatively considered busy if it has an active software thread running on it at the time of the query. If at any point during the window all primary cores are found to be busy (line 7), then SmartHarvest has run out of cores in the idle buffer, indicating that the learning algorithm may have underpredicted the peak number of needed cores. In this case, the agent immediately enforces a short-term safeguard for the next window by expanding the primary VMs' assignment and contracting the ElasticVM's assignment (lines 12–13 and lines 21–22).

```
1  while True do
2      safeguard_invoke = False
3      primary_usage.clear()
       /* Poll the primary VMs' core usage and check for
          violations of the short-term safeguard.            */
4      while time_elapsed ≤ learning_window do
5          primary_busy = GetPrimaryBusy()
6          primary_usage.append(primary_busy)
           /* If all primary cores are busy, we may have
              underpredicted the primary VMs' peak usage.    */
7          if primary_busy = primary_cores then
8              safeguard_invoke = true
9              break
10         end
11     end
12     if safeguard_invoke then
           /* Trigger the short-term safeguard.              */
13         primary_cores = GetSafePrimaryCores()
14     else
           /* Assign a cost to our prediction based on actual
              peak primary usage. Use this to train model.   */
15         costs =
               ComputeCost(primary_cores, primary_usage.max())
16         ModelTrain(features, costs)
           /* Use primary VMs' usage data to generate features
              and predict the next peak.                     */
17         features = ComputeFeatures(primary_usage)
18         primary_cores = ModelPredict(features)
19     end
       /* Never assign fewer than there are busy primary cores,
          to avoid preempting work.                          */
20     primary_cores = max(primary_cores, primary_busy +1)
       /* Assign cores and wait for it to take effect. This call
          also implements the long-term safeguard            */
21     harvest_cores = total_cores − primary_cores
22     SafeAssignCoresAndWait(harvest_cores, primary_cores)
23 end
```

Algorithm 1: EVMAgent logic, which combines the online learner and safeguard from Section 3.4.

When a window completes and the short-term safeguard is not engaged, the agent runs the learning algorithm to predict the peak number of cores needed by primary VMs for the next window (lines 14–18). The agent ensures that this is always greater than the number of currently busy primary cores, to avoid preempting active primary software threads (line 20). Having decided the number of peak primary cores for the next window, the agent then assigns any remaining primary cores to the ElasticVM (lines 21–22). This implicitly sizes the idle buffer: the number of cores in the buffer is the difference between the predicted primary peak and the current number of primary busy cores.

The call to `SafeAssignCoresAndWait` implements a long-term safeguard that disables harvesting for a few seconds

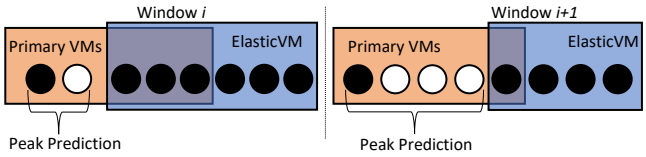


Figure 2. Two consecutive windows: dark circles correspond to busy cores, and white circles correspond to idle buffer cores.

when it is deemed injurious to the primary workload. Otherwise, the agent issues the cpugroups call and sleeps for a short period because the core reassignment operation takes a little time to take full effect.

The learning algorithm uses the CPU usage data collected about the primary VMs during the learning window (line 6) in two ways: as input features to predict the peak primary cores needed for the next window (lines 17–18), and as cost feedback for how good the current window's prediction was (lines 15–16). We discuss this in detail in the next section.

Figure 2 illustrates two consecutive learning windows. Before the first window (left), SmartHarvest had predicted that at most 2 cores would be needed by the primary VMs, so it sized the idle buffer to 1 core. During the first window, SmartHarvest observes the usage of the primary VMs and uses its learning algorithm to predict a peak of at most 4 needed cores for the next window. Just before the start of the second window (right), SmartHarvest resizes the buffer to 3 cores to accommodate the predicted peak.

3.4 Harvesting with Smart Decisions

The EVMAgent from Algorithm 1 uses online learning to decide how many cores from the primary VMs to assign to the ElasticVM in each time window. This continuous, windowed learning approach is necessary for two reasons. First, the load on primary VMs may vary over time. Second, core reassignments need time to take effect, making it infeasible to reassign cores every time the usage of the primary VMs changes. EVMAgent chooses the number of cores for the ElasticVM once per window. If the primary VMs exhaust their assigned cores, the agent invokes a safeguard to reclaim cores from the ElasticVM.

Online learner. The online learner uses a cost-sensitive multi-class classification algorithm to predict the peak core usage of the primary VMs in each time window. Despite the large number of options along several dimensions (prediction target, input features, type of feedback received, learning algorithm, and model representation), the properties and constraints of our problem led to fairly natural choices.

Prediction target: As the ElasticVM should only use cores the primary VMs are unlikely to need, a natural prediction target is the *peak* number of primary cores in the next window. Given that number, we can assign any cores from the primary VMs above the predicted peak to the ElasticVM. This implicitly sizes the idle core buffer, which grows and shrinks between the predicted peak (which remains fixed during a window) and the current primary VMs' usage (Figure 2).

Features: Since SmartHarvest runs in a public cloud, it only has black-box access to the primary VMs, which severely restricts the types of features it can use. Besides static properties of the VMs, EVMAgent can only observe the external resource utilization of the VMs by calling the hypervisor. We focus on CPU utilization; application performance metrics or service-level objectives are completely opaque to the agent.

EVMAgent collects the CPU utilization of primary VMs in the current learning window and computes the following five features for training and predictions: *the min, max, average, standard deviation, and median CPU usage*. We identify these features using a feature selection technique [3] that trains a decision tree using offline data to rank features according to their importance in deciding the predicted value. We considered additional features (e.g., different CPU usage percentiles, delta between consecutive CPU peaks), but eliminated them during feature selection because they did not improve prediction accuracy. The number of features was intentionally kept small to reduce training and prediction overheads and enable millisecond learning windows.

Feedback: The feedback we receive for predictions determines the learning paradigm we can apply. If the primary VMs use fewer cores than the predicted peak of a window, we learn their actual peak usage, so we can apply a supervised learning algorithm (the actual peak usage is the correct "label"). But if the primary VMs use all of the predicted peak cores, then we never learn the actual number of peak cores they would have used (we get no feedback)! Not coincidentally, this is precisely when EVMAgent invokes the safeguard mechanism discussed below (lines 12 to 13, Algorithm 1). The safeguard helps restore the supervised feedback in the next window, allowing the agent to continue training the online learner (lines 14 to 18).

Learning algorithm: Supervised learning algorithms can broadly be classified into regression (what is the relationship between x and y) or classification (which class y does input x correspond to). In our setting, a regression—such as linear regression or time-series analysis methods like ARIMA [19]—would predict a continuous value for the peak usage of the primary VMs. This is awkward because a continuous value forces the regressor to accommodate "cliffs" around the target values. Moreover, an underprediction of the peak usage is far worse than an overprediction; a single regression target does not naturally allow us to minimize such a differentiated cost function.

In contrast, a *cost-sensitive multi-class classification* algorithm [2, 17] allows us to train a separate predictor for each core count (class) and select the class with lowest cost during predictions. This accommodates our notion of differentiated costs, because it allows us to specify any cost value for each class without worrying about the relationship between these

values. Since the costs are continuous, we can run a separate regression for each class with cost as the predicted value. We describe our cost function in Section 4. In general, we assign lower costs to classes that are equal or closely above the correct class, and higher costs to all other classes. This skews the learner towards making small overpredictions, and heavily penalizes it for underpredicting primary usage (which triggers the safeguard).

Model: In theory, we can use any model we want for the per-class regressors, ranging from linear models to decision trees to neural networks. These models vary in their training and prediction time, and not all of them can be efficiently trained in an online manner. Given our requirement of a lightweight learner that needs to make predictions every few milliseconds, we opt for a simple linear model.

Safeguards. The online learner is designed to predict slightly more than the peak core usage of the primary VMs. However it is still possible that it under-predicts the CPU peak. When this happens, the primary VMs do not get all cores they need and their performance degrade. We design a two-level safeguard mechanism to mitigate this problem.

The first-level, short-term safeguard is triggered whenever the usage of the primary VMs matches their predicted peak (i.e., the idle core buffer becomes empty). Two major problems occur when there is no idle core in the buffer. First, the primary VMs may have wanted to use more cores but were unable to do so. Second, the online learner no longer receives supervised feedback for its prediction because we do not know what the true peak usage of the primary VMs would have been. Thus, the learner is unable to properly update the learned model. The first-level safeguard addresses this failure by proactively expanding the number of cores assigned to primary VMs for the next learning window instead of following the online learner's prediction. Lines 7–10, lines 12–13 and lines 20–22 in Algorithm 1 implement this logic; the function `GetSafePrimaryCores` computes the new core assignment to the primary VMs as one plus the peak number of cores used by them over the past one second, or the total number of cores originally allocated to them if this number is smaller. While the safeguard is in place, EVMAgent continues to monitor the primary VMs' core usage. Since the safeguard has expanded the number of cores assigned to primary VMs, there is a good chance we will observe their actual peak usage, and hence collect supervised learning feedback. This feedback is particularly important because it comes from a period where the learner made an underprediction.

Some VM workloads may temporarily experience unpredictable CPU usage due to high swings in load. To avoid performance degradation from frequent misprediction of peak CPU needs, we use a second-level, long-term safeguard that temporarily disables harvesting altogether. Since primary VMs are treated as black-boxes, we use vCPU dispatch wait times available from the hypervisor as a proxy for VM quality of service (QoS). Ready vCPUs from primary VMs experience longer wait time to be scheduled onto physical CPUs when the hypervisor is short of physical cores to run them. Therefore, we can infer performance degradation from long vCPU wait times and trigger the long-term safeguard to protect primary VMs' performance. EVMAgent monitors the 99th percentile value in the wait time per dispatch across all vCPUs belonging to the primary VMs, for every 500ms. Based on our experimental setting, the P99 vCPU wait time is typically below 6 μs when there are abundant idle physical cores to schedule them on. If at least 1% of all vCPU wait times is longer than 50 μs for two consecutive 500ms windows, EVMAgent immediately gives all cores back to the primary VMs and stops harvesting. EVMAgent restarts harvesting 10 seconds after it is stopped to check for possible VM load changes. This second safeguard is implemented in function `SafeAssignCoresAndWait` in Algorithm 1. While harvesting is disabled, the learner continues to learn in the background to take advantage of the full-information feedback.

Hyperparameters. We tuned the SmartHarvest hyperparameters, including the prediction window, cost function and safeguards, for the most latency-sensitive workloads with microsecond-scale service-level objectives (e.g., Memcached). The tuning is done once and the same set of parameter values are conservatively used for all other workloads. We discuss these parameters further in Section 5.

4 Implementation

SmartHarvest can be implemented for any hypervisor that accepts calls (aka hypercalls) from privileged VMs for monitoring core utilization and for reassigning physical cores across VMs. The faster these mechanisms are, the more accurate the assignments are and the fewer cores are needed in the idle buffer to handle spikes in the primary VMs' load. It is important for hypercalls to complete quickly and the core assignment to take effect as soon as possible after the call. For non-preemptive hypervisors, for example, the reassignment may not take effect until a later scheduling event. Next, we describe our SmartHarvest implementation for Hyper-V [59], and the changes needed in the hypervisor. SmartHarvest can be easily implemented for other hypervisors (e.g., Xen, KVM), some of which might not require modifications.

SmartHarvest. Hyper-V is a type-1 hypervisor that runs directly on the bare-metal hardware. It uses a priviledged VM, called "root partition", for executing device drivers and any server-level agents. The root partition is equivalent to the "Dom0" VM in Xen [16]. The hypervisor directs I/O operations from guest VMs to the root partition and returns the results back to those VMs. We build EVMAgent as a user-space program that runs in the root partition and implements Algorithm 1 in 1,900 lines of C++. We used C++ because

the agent must run frequently and efficiently. EVMAgent makes hypercalls using Hyper-V's Host Compute Service (HCS) API, which provides calls for monitoring server-wide physical core utilization and assigning a group of physical cores (cpugroup) to a VM. Core reassignment involves adjusting cpugroups. EVMAgent maintains 2 non-overlapping cpugroups for (1) the ElasticVM, including any cores that it has harvested from primary VMs, and (2) primary VM cores being actively used or in the idle core buffer.

The online learning component of EVMAgent is agnostic to the hypervisor. We implement it using the Vowpal Wabbit (VW) library [9], which contains an efficient online implementation of the cost-sensitive multi-class classification algorithm described in Section 3.4.

Changes to Hyper-V. In the original version of Hyper-V, changing the set of cores assigned to a VM involves two hypercalls: one to detach the VM from the cpugroup and one to attach it to a different cpugroup. Since EVMAgent maintains two non-overlapping cpugroups, core reassignment between two VMs involves 4 hypercalls in total. Hyper-V is also non-preemptive, so it waits until the next time the affected VMs are scheduled to effect the cpugroup changes. This means that the impact of a cpugroup change can be delayed for an entire scheduling period (10ms) in the worst case.

To address these issues, we modify Hyper-V as follows: (1) We create a new cpugroup hypercall, called `merge-call`, that combines detaching a VM from a cpugroup and attaching it to a new one. Besides reducing the number of hypercalls, this change optimizes cpugroup management in that physical cores belonging to both the existing and new cpugroups are not affected by `merge-call`. (2) Instead of waiting until the next scheduling event, we make core reassignments preemptive by sending interprocessor interrupts (IPIs) to the affected cores. The IPIs immediately stop VM execution on any core to which the VM is no longer attached.

Implementing SmartHarvest on Xen would require the IPI change above because Xen is also non-preemptive [10]. KVM on Linux already uses IPIs to effect cpugroup changes [6].

SmartHarvest versions and configurations. We implement two versions of SmartHarvest. Version `cpugroups` uses unmodified cpugroups with the existing delayed effect of non-preemption. To remove cpugroup creation from the critical path, EVMAgent pre-creates all possible cpugroups and simply makes detach/attach hypercalls to the desired cpugroups. We use this version for most of our experiments. We also implement a more efficient version called `IPIs`. This version uses our `merge-call` and our preemptive IPI mechanism for core reassignments. We use this version to assess the importance of online learning as a function of how fast the underlying system is at reassigning cores.

By default, we configure both versions to use a 25ms learning window and poll the primary VMs' usage every 50us

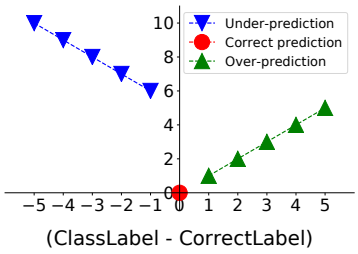


Figure 3. Sample cost function (skewed cost).

during this window (lines 4–10, Algorithm 1). We set the learning rate to be 0.1 (the default value from VW). The learning rate is kept constant so the agent can learn continuously online. In the `cpugroups` version, if a core reassignment is needed, we configure the agent to sleep for 10ms after the change (in `SafeAssignCoresAndWait`). The `IPIs` version does not need this delay because core reassignments take effect faster.

The cost function we use for our cost-sensitive multi-class classification algorithm is shown in Figure 3. The cost grows linearly as our predicted peak usage of the primary VMs deviates from their actual peak usage. It adds a constant additional cost (equal to the initial number of cores allocated to the primary VM) to underpredictions because they trigger the safeguard. We explore alternative cost functions in Section 5.6. For the safeguard, we implement both a default conservative one, which increases the number of cores assigned to the primary VMs based on their peak CPU usage in the last second, and an aggressive version, which reassigns all cores to the primary VMs. To query the vCPU wait times, we use the *Hypervisor Virtual Processor\CPU Wait Time Per Dispatch counter* from Hyper-V.

5 Evaluation

5.1 Methodology

Workloads. We use four real latency-sensitive workloads as primary VMs: IndexServe, Memcached, moses and img-dnn. They all have stringent latency requirements, so SmartHarvest must protect their tail latencies. **IndexServe** is used by Microsoft Bing to serve the web index for user search queries [36]. We use real query traces from Microsoft to generate load for IndexServe in our experiments. **Memcached** is a widely used in-memory key-value store that has sub-millisecond latency [8]. We use *mutilate* [44] to generate load using query distributions from Facebook [14] with all *GET* requests. **moses** and **img-dnn** are latency-critical benchmarks from the TailBench suite [41]. **moses** [43] is a machine translation application written in C++. **img-dnn** [24] is an image recognition application that identifies handwritten characters in the MNIST dataset [30].

Each primary workload is allocated a 10-core VM. Hence, the maximum number of cores we can harvest from each

	IndexServe (500QPS)	Memcached (40KQPS)	moses (400QPS)	img-dnn (2000QPS)
CPU usage	1.3	2.3	1.5	1.7
Peak CPU usage	7	7.7	5.2	6.9

Table 1. avg CPU stats in #cores.

primary VM is 10. For all the primary workloads, we launch the client and server processes in the same VM to put more pressure on SmartHarvest to protect tail latencies: performance losses due to harvesting cannot be hidden behind networking delays or other overheads. We run each workload at low load on the server configuration we use (see below), which present good opportunities for harvesting allocated but idling CPU cores. Table 1 lists the average number of busy cores and the average of the peak number of busy cores for the four primary workloads when each is run alone in a 10-core VM. We poll CPU usage every 50μs and calculate the peak over every 25ms windows. Although the average number of busy cores is low, the peak usage is significantly higher due to short-term query bursts. The peak usage is more important for harvesting, since we must reserve enough cores in the idle buffer to accommodate the bursts of activity in the primary VMs.

We tested three workloads in ElasticVMs: CPUBully, HDInsight, and Terasort. **CPUBully** is a multi-threaded, synthetic workload where each worker thread performs a CPU-bound sum operation on integer values. It has few memory/disk accesses and can utilize as many CPU cores as available. **HDInsight** [1] is a real workload that trains a machine learning (ML) model with TensorFlow used by ML teams in Bing. To keep experiments short, we run one iteration of the training of a logistic regression model with 2GB of data. HDInsight is a CPU-bound workload that completes faster with more CPU cores. **TeraSort** [4] executes Hadoop's TeraSort benchmark to sort a given number of records. We use it to sort 10 million records of 100MB of data. It exhibits high memory and I/O usage due to read and shuffle operations.

Evaluation metrics. SmartHarvest aims to maximize the number of harvested core cycles, while having minimal impact on the QoS of the primary VMs that are co-located with the ElasticVM. We quantify QoS for all four primary workloads using their reported P99 latency, and compare it to the latency in the absence of harvesting. We assume that their P99 latency should not degrade more than 10%. We quantify the amount of harvesting using the average number of cores harvested for CPUBully. For HDInsight and TeraSort, we use the reduction in execution time (speedup), compared to their execution on a single core.

Baseline. We compare SmartHarvest against two baseline approaches. For a fair comparison, we implement both of the baselines using our own agent and check CPU utilization of

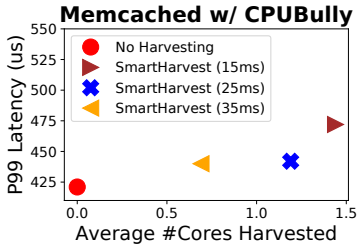


Figure 4. Learning window size exploration.

primary VMs (i.e., their number of busy cores) every 50us to detect bursts. After each core reassignment, we insert a 10ms delay in the `cpugroups` implementation. The `IPIs` implementation of baselines does not need a delay after the core reassignment.

(1) *Prev Peak* is a simple heuristic-based approach that allocates cores to primary VMs based on observed CPU peak from the previous time window. We picked 25ms as the window length to match the learning window size used in SmartHarvest. When the primary VMs use up all their allocated cores, all cores are returned to them from ElasticVM for the next 25ms time window to obtain accurate information on their peak usage again.

(2) *Fixed Buffer* approach applies the design proposed by PerfIso [36] to a virtualized environment. When configured to maintain a fixed-size buffer, the agent simply "slides" the idle buffer reactively every time it reads the primary VMs' CPU utilization, by adding or removing ElasticVM cores as the primary VMs' usage changes, so that the size of the idle buffer stays fixed.

Testbed. We use a two-socket Intel server with Xeon Platinum 8160 processor with 24 cores per socket, running at 2.10GHz, and 255GB DRAM. We report the average of 3 runs. To avoid performance jitter on primary VMs that is independent of harvesting, we disable simultaneous multithreading (SMT), so that a single VM uses each physical core at each point in time. We also disable C-states, P-states, and Turbo-Boost to isolate power management jitter. We run both the primary VMs and ElasticVM on one socket to avoid the variation caused by workloads spanning across NUMA nodes. This maximizes interference between the two workloads. By default, the root partition can run on any logical processor on the server. To isolate management work from the workloads inside guest VMs, we restrict the root partition to 3 cores using Hyper-V's minroot configuration [5]. EVMAgent itself only requires 1 core since it is single-threaded.

5.2 Learning Window Selection

As described in Algorithm 1, EVMAgent makes predictions at regular intervals called learning windows. The window length is key to SmartHarvest's effectiveness. Longer windows allow us to observe the CPU usage for longer and be more conservative, but may fail to adapt quickly enough to

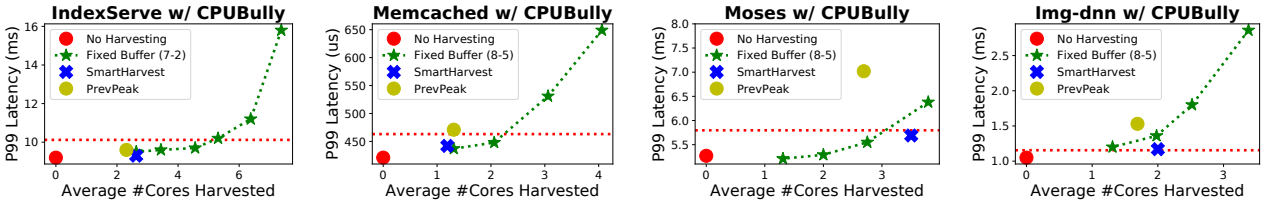


Figure 5. Single Primary VM co-located with CPUBully.

behavior changes in primary VMs. Shorter windows make more predictions, potentially leading to frequent core reassignments that may end up violating the SLO of a primary VM. Given the lack of information about the primary VMs' workloads in public clouds, we conservatively use Memcached to tune the window length because it is a very latency sensitive workload with P99 latency in hundreds of μs. We use the tuned value for the rest of our experiments, and also recommend this value for deployments of SmartHarvest for other workloads.

We vary the window length between 15ms to 35ms in Figure 4 and study the impact on the performance of the primary VM and the amount of harvesting. The y axis is the P99 latency of Memcached (starting at the nominal 421μs), while the x axis shows the average number of cores harvested by the CPUBully over one-minute runs. The 25ms window length strikes a balance between the observation time and reassignment frequency. Hence, achieves a good amount of harvesting while having a low impact on the performance of Memcached. We use 25ms in all subsequent experiments.

5.3 Harvesting from One Primary VM

We first evaluate the `cpugroups` version of SmartHarvest with a single primary VM. We launch the primary VM with 10 cores and the ElasticVM with 1 core in a single NUMA node. The ElasticVM can grow to a maximum of 11 cores. We quantify the impact of harvesting by comparing the tail latency of the primary VM with harvesting enabled and disabled. In the latter case, the ElasticVM is limited to 1 core. We verified that ElasticVM on a single core does not impact the latency of the primary VM. We use the 25ms prediction window for all primary VMs (no per-application tuning).

Figure 5 shows the P99 latency for each primary VM and the corresponding average number of harvested cores for CPUBully. The horizontal lines across the graphs mark the allowable P99 latency (10% increase from the original P99 latency). The green curves show the results for fixed-sized buffers (decreasing in size from left to right). For IndexServe, a fixed buffer of 4 cores is enough to harvest more than 5 cores and stay within the allowed P99 latency. But for Memcached, a minimum of 7 buffered cores are needed to harvest up to just 2 cores due to greater load variation.

Overall, any static buffer size selection is likely to either harvest fewer cores than possible or lead to SLO violations. The heuristic that uses the CPU peak from previous time

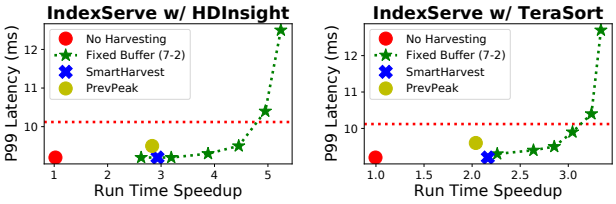


Figure 6. IndexServe co-located with real workloads.

window fails to meet target latencies for the more stringent workloads. For both moses and img-dnn, *PrevPeak* results in more than 50% increase in P99 latency due to sudden increases in CPU utilization that is not captured from the CPU peak usage in the past 25ms.

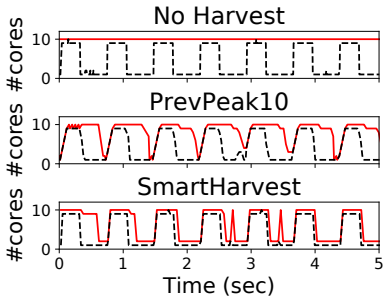
SmartHarvest keeps the P99 latency of the primary VM below the allowed maximum for all workloads while harvesting between 1.5 to 3.5 cores. SmartHarvest does not always harvest as much as the best fixed-size buffer for each load because we use a conservative learning window length that works for sub-millisecond workloads. The fact that SmartHarvest's dynamic tuning approach behaves well across all workloads is an indication of its robustness.

Realistic workloads in ElasticVMs. Figures 6 show results with realistic batch processing jobs, HDInsight and TeraSort, respectively, colocated with IndexServe. The X-axes show speedup on completion time of the batch jobs, so higher is better. The results show realistic batch jobs can get meaningful work completed on harvested cores. SmartHarvest allows the batch workloads to achieve substantial reduction in execution time (2x - 3x) without significant performance slowdown for primary VMs. The results also show that SmartHarvest is robust to widely different batch workloads running in the ElasticVM.

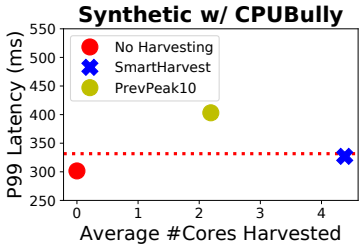
Single primary VM with varying load. We evaluate our system with varying load in a primary VM by decreasing the offered load for Memcached from 80kQPS (medium load) to 20kQPS (vey low load), and then increasing it to 160kQPS (high load) again. Each offered load is run for a minute. Table 2 shows the increase in P99 latency at each load and the average number of cores harvested over the entire 3-minute run. SmartHarvest keeps P99 latency low across all load while harvesting 1.6 cores. SmartHarvest is able to quickly adapt to load changes by continuously learning CPU usage patterns. The fixed buffer policy fails to maintain P99 latency

	P99 Latency (μs)			
	80kQPS	20kQPS	160kQPS	Avg #cores harvested
NoHarvest	477	300	943	0
SmartHarvest	477 + 10	300 + 4	943 + 5	1.6
PrevPeak	477 + 19	300 + 15	943 + 18	1.4
Fixed buffer 5	477 + 53	300 + 400	943 + 158	2.4
Fixed buffer 6	477 + 14	300 + 150	943 + 19	1.6
Fixed buffer 7	477 + 8	300 + 28	943 + 8	1.1

Table 2. Memcached with varying load over time.



(a) Primary VM CPU utilization



(b) Top-level performance metrics

Figure 7. Synthetic workload co-located with CPUBully.

at low levels. For example, a fixed buffer size of 6 harvests as much as SmartHarvest at the cost of doubling the P99 latency at 20kQPS load. Both PrevPeak and a fixed buffer size of 7 lead to low impact on all tail latencies but harvest fewer cores than SmartHarvest.

Conservative heuristic policy. We also evaluated a more conservative version of the heuristic baseline that uses the highest CPU peak observed across multiple windows. For example, *PrevPeak10* uses the peak from the past ten 25ms time windows (total 250ms) to assign cores to the primary VMs. To balance this very conservative allocation, instead of giving all cores back whenever the primary VM runs out of idle cores, *PrevPeak10* returns one core to the primary VM at a time. Figure 7 shows results from a synthetic workload with a square-wave CPU usage pattern run inside primary VM. This workload simulates a multi-threaded server working

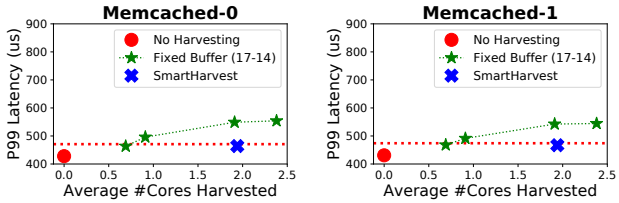


Figure 8. Memcached + Memcached with CPUBully.

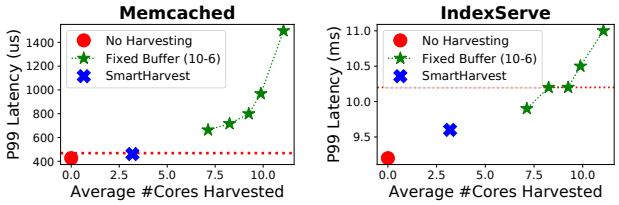


Figure 9. Memcached + IndexServe with CPUBully.

on client requests with fixed processing duration. Figure 7a shows the peak CPU utilization of the primary VM and the allocated cores under no-harvest, *PrevPeak10* and SmartHarvest; the red solid line marks the number of cores allocated to the primary VM and the black dashed line shows the peak number of cores used by primary VM from every 25ms window. *PrevPeak10* fails to react fast to drops in CPU utilization because it relies on old peak usage, which is not representative of the reduced CPU usage. It also takes longer to expand the primary VM when its CPU usage rises. SmartHarvest learns the occurrence of sudden high-to-low transitions after observing it for a few times and starts allocating fewer cores to the primary VM. SmartHarvest also adapts better to increases in CPU usage due to its safeguard design. As a result, SmartHarvest is able to harvest more than *PrevPeak10* while impacting the P99 latency less, as shown in Figure 7b. We believe any heuristic-based approach that relies on peak usage from longer windows in the past will suffer from these shortcomings.

5.4 Harvesting from Multiple Primary VMs

We now evaluate SmartHarvest with more than one primary VM running on the server, again using the `cpugroups` version. Since each primary is allocated 10 cores, the opportunity for harvesting idle cores is higher (20 total).

SmartHarvest treats multiple primary VMs as a single logical entity that shares the same cpugroup. The co-located ElasticVM runs on another (non-overlapping) cpugroup. The maximum number of (physical) cores belonging to the shared cpugroup is the sum of allocated cores for each individual VM. Each VM may have one of its virtual cores scheduled on any physical core in the shared cpugroup, but the VM cannot use more physical cores than its original core allocation. When VMs share a cpugroup, we rely on Hyper-V to make the right scheduling decisions among the VMs to minimize interference. EVMAgent just needs to learn the

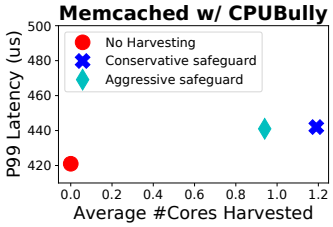


Figure 10. Effectiveness of short-term safeguards.

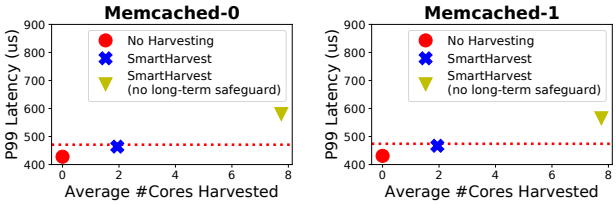
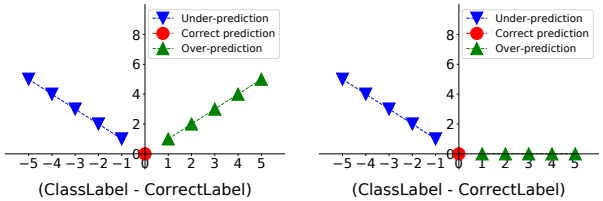


Figure 11. Effectiveness of long-term safeguard.



(a) Symmetric cost **(b)** Hinged cost

Figure 12. Alternative cost functions.

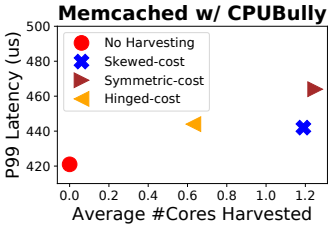


Figure 13. Comparison of different cost functions.

aggregated CPU usage patterns and adjust the cpugroup sizes accordingly. This approach incurs low computational overhead, since SmartHarvest maintains and updates a single ML model.

We experiment with the following combinations of primary VMs: *(A) Memcached + Memcached* and *(B) Memcached + IndexServe* . CPUBully is the batch workload in ElasticVM. Figures 8 - 9 show the impact on P99 latency for each primary VM and the total number of harvested cores.

Combination A (Figure 8) includes two Memcached instances run at the same load. Running two primary VMs in the same cpugroup means sharing of cores and possible loss of data locality due to cold caches. Due to the latency-sensitive nature of Memcached, *as many as 17 buffer cores are required to maintain P99 latencies* bellow their allowable values. SmartHarvest dynamically achieves the harvesting policy without any tuning of its parameters.

Combination B (Figure 9) presents a more interesting scenario as the two primary VMs have different latency requirements and load. Hence, it is *challenging to find any single fixed buffer size that works well for both.* Since SmartHarvest automatically stops harvesting when any of the primary VMs experiences long scheduling delay for its vCPUs, it harvests conservatively and maintains low performance degradation across all instances.

5.5 Effectiveness of Safeguards

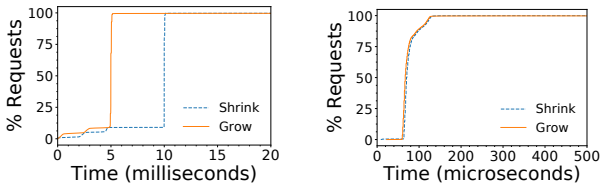
SmartHarvest's two-level safeguard mechanism is triggered whenever the primary VMs exhaust the idle core buffer or their vCPUs experience long wait time before getting scheduled to run. We study the effectiveness of the safeguards described in Section 3.4 for running Memcached.

In Figure 10, we compare two versions of the short-term safeguard: *conservative-safeguard*, which increases the number of primary cores to match peak CPU usage over the past one second plus one, and *aggressive-safeguard*, which gives all cores back to the primary VMs. The aggressive-safeguard allows the model to acquire complete feedback (i.e., the actual peak usage of the primary VMs) when triggered. However, this safeguard tends to be wasteful of resources when the primary VMs' CPU utilization exceeds all assigned cores by small amounts. The conservative safeguard prevails by maintaining similar primary tail latency while allowing a few more cores to be harvested. We use the conservative safeguard in all other experiments.

We next evaluate the effectiveness of the long-term safeguard. SmartHarvest struggles when primary workloads show aperiodic patterns that are hard to predict. For example, when two Memcached VMs are co-located, SmartHarvest without the long-term safeguard fails to predict the aggregate usage pattern of the two VMs, resulting in more than 50% increase in their tail latencies (Figure 11). To avoid performance loss for primary VMs, we rely on the long-term safeguard to disable harvesting in such scenarios. Figure 11 shows that the long-term safeguard effectively maintains workloads' P99 latencies by detecting long primary vCPU wait time and temporarily disabling the harvesting.

5.6 Cost Function Exploration

All results thus far have been collected using the skewed cost function in Figure 12a, which is a natural choice because it penalizes underpredictions more severely than overpredictions. We also considered two alternative cost functions, shown in Figure 12b. Figure 13 compares the three cost functions for Memcached running with CPUBully in the ElasticVM. We observe that by not differentiating the cost of underpredictions and overpredictions, the Symmetric-cost function makes more underpredictions and has a larger impact on the primary VM. On the other hand, since the Hinged-cost function assigns the same cost to all overprediction labels



(a) CpuGroups **(b)** IPI

Figure 14. CDF of time to grow/shrink ElasticVM by one core.

Feature Computation	Model Update	Model Inference
2.6 ± 1.2	10.8 ± 4.6	6.5 ± 4.1

Table 3. Latencies of learning operations (in μs).

(even those that are far from the true label), the model learns to overpredict frequently, thereby reducing the opportunity to harvest cores from the primary VMs. Figure 13 shows our Skewed-cost function harvests more than Hinged-cost while achieving similar tail latency for the primary VMs.

5.7 Overheads Incurred by SmartHarvest

We now discuss the latency overheads added by SmartHarvest's components.

Systems component of EVMAgent. We benchmarked the time taken to expand and shrink a VM using `cpugroups` and `IPIs` and plot the distribution in Figure 14. Resizing a VM using `cpugroups` requires two hypercalls: one to unbind the current cpugroup and another to bind it to a different cpugroup. Each hypercall takes approximately 200μs. The cpugroup assignment is not immediate and may take up to 5ms for growing and 10ms for shrinking as shown in Figure 14a. `IPIs` with our new `merge-call` requires a single hypercall to change the core-affinity of the current cpugroup. The changes are typically visible in less than 100μs from the time the hypercall is initiated (Figure 14b). Section 5.8 quantifies the impact of these overheads on responsiveness and learning accuracy.

Learning component of EVMAgent. The learning component of EVMAgent is lightweight. We benchmarked feature computation, model update, and prediction latencies from EVMAgent and report them in Table 3. On average, it takes less than 22 μs to perform all learning operations, which is negligible compared to the learning window length (milliseconds).

5.8 Responsiveness Vs Learning Accuracy

`IPIs` allow the effect of adding or removing a core to/from a VM to take effect within 130μs at the 99th percentile, substantially faster than the 10ms with unmodified `cpugroups`. Intuitively, if cores can be reassigned very quickly, reacting to primary VMs' needs after observing changes might perform better than predicting their demands in advance. To

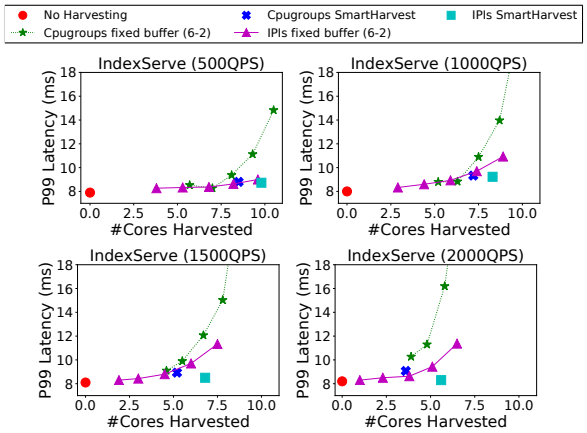


Figure 15. SmartHarvest using `cpugroups` vs. `IPIs` for different IndexServe loads.

help answer this question, we explore if the faster assignment mechanism with `IPIs` eliminates the need of online learning for core harvesting.

Figure 15 compares `cpugroups` and `IPIs` by running IndexServe with CPUBully at various loads on a two-socket Intel server with Xeon E5-2690 v4 processor (14 cores per socket, 256GB DRAM, 2.60GHz). The faster implementation ("IPIs fixed buffer") results in smaller tail latency increase and more harvesting for every fixed-buffer configuration and for every load, compared to using the unmodified `cpugroups` ("Cpugroups fixed buffer"). The frequent CPU usage monitoring and the fast reaction with IPIs allow more fixed buffer configurations to operate safely.

Figure 15 also shows that, though shrinking the ElasticVM in the `cpugroups` implementation of SmartHarvest (labeled "Cpugroups SmartHarvest") may take up to 10ms, the P99 latency of IndexServe is not substantially affected. The reason is that this delay is only detrimental to primary VMs when they need more cores than are available in the buffer of idle cores. Its negative impact is more pronounced at higher IndexServe latency percentiles. Still, the `cpugroups` implementation harvests fewer cores on average than its `IPIs` counterpart (labeled "IPI SmartHarvest"). The faster core re-assignment enabled by `IPIs` allows SmartHarvest to both harvest more cores and have less performance impact on IndexServe.

When SmartHarvest uses `IPIs`, it harvests more cores with the same or lower tail latencies than the fixed-buffer configurations. The relative benefits of online learning are lower than those with the `cpugroups` implementation. Nevertheless, the online nature of SmartHarvest is crucial for efficient harvesting at medium and high loads (QPS) even when IPIs are used: there is no single fixed buffer size that works well across all loads, whereas learning consistently and automatically meets tail latency requirements without additional tuning across loads.

6 Related Work

Several proposed systems use the SLOs of the primary tenants to regulate resource harvesting [40, 44–46, 51, 68, 70], which limits their applicability in public clouds where such information is not available. Other proposals perform extensive profiling to measure or reduce interference among co-located workloads [49, 67, 69]. Profiling the diverse customer workloads is impractical at large scale, especially as the workload characteristics and load change over time. We therefore focus next on prior work that assumes no knowledge of the workloads and SLOs of the primary VMs.

Time series forecasting models such as ARIMA [20] and neural networks [32] can be used for online prediction and achieve good accuracy. However, they require expensive offline training and/or suffer from high overheads in online retraining and prediction. A lightweight online learner has the advantage of adapting to frequent changes in data patterns at a fine time granularity. PRACTISE [62] uses a neural network model to predict future VM resource usage (e.g., CPU, memory, disk, and network bandwidth). The model has an average retraining time of 30 seconds and prediction time of 10 seconds on a 4-core Intel i7 CPU, which is orders of magnitude higher than the microsecond-level update/predict times in SmartHarvest. The quick learning operations in SmartHarvest are necessary to meet sub-millisecond tail latency requirements of latency-critical workloads.

MS Manners [31] targets older machines with single-core CPUs and manages CPU cycles that background tasks can use. We manage CPU cores rather than just CPU cycles; this is critical for hosting latency sensitive workloads on modern multicore servers.

PerfIso [36] maintains a fixed number of cores in the idle buffer, which corresponds to the fixed-buffer policies we compared SmartHarvest against in this work. PerfIso needs offline profiling to determine the fixed number, and does not adapt the number of idle cores to different primary VMs or load levels on the same primary VM. SmartHarvest overcomes these limitations and maintains lower latency for the primary VMs while harvesting more resources.

Kalyvianaki et al. [39] designed a Kalman-filter-based feedback controller that can self-configure its parameters to provision CPU resources for any workload without a priori information. However, the self-configured controller can result in a latency increase for more than 20% of scheduled requests under load variations. Scavenger [37] employs a regulator to harvest resources including cores and main memory using the average and standard deviation of the primary resource consumption. Some of the reported results show large increases in P95 latencies and the impact on P99 latencies is unclear. In contrast to these work, SmartHarvest directly predicts the peak needed cores of primary VMs and protects their P99 tail latencies.

Machine learning has shown promise in various aspects of resource management, including learning-based cluster schedulers [28, 29], learning-assisted schedulers [47, 48, 63–65], and resource allocation mechanisms [11, 60, 66]. These approaches use models that are trained offline, limiting their applicability to our problem of adapting to the resource usage of black-box VMs running in the public cloud. DeepDive [52] applies online unsupervised learning to detect resource interference across black-box VMs, but does not address resource allocation. PRESS [33] and AGILE [50] use online predictors for resource allocation, and we build on such continuous learning methodology. However, we compute additional features from the instantaneous cores states, employ skewed cost functions to penalize underpredictions more than overpredictions, and use safeguards to quickly recover from mispredictions.

7 Conclusion

We proposed SmartHarvest, a system that uses online learning to dynamically predict the number of cores that can be safely harvested from a set of black-box VMs. We also proposed ElasticVM, a new VM type that harvests these cores. Our results showed that SmartHarvest and ElasticVM are capable of harvesting cores aggressively, while protecting the tail latency of co-located latency-sensitive VMs.

References

[1] Azure HDInsight. https://azure.microsoft.com/en-us/services/hdinsight/.

[2] CSOAA multiclass classification. https://github.com/rvw-org/rvw/wiki/CSOAA-multiclass-classification.

[3] Feature importances with forests of trees. https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html.

[4] Hadoop TeraSort. https://hadoop.apache.org/docs/r3.2.0/api/org/apache/hadoop/examples/terasort/package-summary.html.

[5] Hyper-V Minroot. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-minroot-2016.

[6] Kvm/linux kernel scheduler. https://elixir.bootlin.com/linux/v4.14/source/kernel/sched/core.c#L479.

[7] Machine Learning Reductions. http://hunch.net/~jl/projects/reductions/reductions.html.

[8] Memcached: high-performance, distributed memory object caching system. https://memcached.org/.

[9] Vowpal Wabbit. https://github.com/VowpalWabbit/vowpal_wabbit/wiki.

[10] Xen scheduler. https://github.com/xen-project/xen/blob/master/xen/common/schedule.c#L1293.

[11] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.

[12] Amazon Elastic Compute Cloud. Amazon EC2 Spot Instances, 2019. https://aws.amazon.com/ec2/spot/.

[13] 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. Providing slos for resource-harvesting vms in cloud platforms. In *Proceedings of the 14th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), November 2020.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[15] Microsoft Azure. Azure Spot Virtual Machines, 2020. https://azure.microsoft.com/en-us/pricing/spot.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 2003 International Symposium on Operating Systems Principles*, pages 164–177, 2003.

[17] Alina Beygelzimer and John Langford. The offset tree for learning with partial labels. *CoRR*, abs/0812.4044, 2008.

[18] Léon Bottou. On-line learning and stochastic approximations. In *On-line Learning in Neural Networks*, pages 9–42. Cambridge University Press, 1998.

[19] G. E. P. Box and G. M. Jenkins. *Time Series Analysis: Forecasting and Control*. Holden-Day, San Francisco, 1976.

[20] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[21] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, April 2016.

[22] Marcus Carvalho, Walfredo Cirne, Franciso Brasileiro, and John Wilkes. Long-term SLOs for reclaimed cloud computing resources. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 20:1–20:13, Seattle, WA, USA, 2014.

[23] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Chandra Krintz. See spot run: Using spot instances for mapreduce workflows. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 7, USA, 2010. USENIX Association.

[24] Google Cloud. A deep network handwriting classifier. https://github.com/xingdi-ericyuan/multi-layer-convnet.

[25] Google Cloud. Preemptible VM Instances, 2020. https://cloud.google.com/compute/docs/instances/preemptible.

[26] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 153–167, New York, NY, USA, 2017. ACM.

[27] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[28] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, March 2013.

[29] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[30] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[31] John R. Douceur and William J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 247–260. ACM Press, 1999.

[32] Ray J Frank, Neil Davey, and Stephen P Hunt. Time series prediction and neural networks. *Journal of intelligent and robotic systems*, 31(1):91–103, 2001.

[33] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. pages 9 – 16, 11 2010.

[34] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

[35] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[36] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. Perfiso: performance isolation for commercial latency-sensitive services. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC 18)*, pages 519–532, 2018.

[37] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 272–285, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, November 2016. USENIX Association.

[39] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th International Conference on Autonomic Computing*, pages 117–126, 2009.

[40] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 598–610. ACM, 2015.

[41] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[42] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *Proceedings of the 8th ACM International Conference on Web Search and Data Mining*, pages 7–16. ACM, 2015.

[43] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007.

[44] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 9th European Conference on Computer Systems*, page 4. ACM, 2014.

[45] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[46] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving resource efficiency

at scale with Heracles. *ACM Transactions on Computer Systems (TOCS)*, 34(2):6, 2016.

[47] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets 2016, pages 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

[50] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[51] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. Hipster: Hybrid task manager for latency-critical cloud workloads. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 409–420. IEEE, 2017.

[52] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the USENIX Annual Technical Conference*, 2013.

[53] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high *cpu* efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 361–378, 2019.

[54] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[55] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[56] Eric Schurman and Jake Brutlag. Performance related changes and their user impact. In *velocity web performance and operations conference*, 2009.

[57] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the 14th EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the Symposium on Cloud Computing*, 2013.

[59] Anthony Velte and Toby Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., USA, 1 edition, 2009.

[60] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 363–378, Berkeley, CA, USA, 2016. USENIX Association.

[61] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[62] Ji Xue, Feng Yan, Robert Birke, Lydia Y Chen, Thomas Scherer, and Evgenia Smirni. Practise: Robust prediction of data center time series. In *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, pages 126–134. IEEE, 2015.

[63] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, Joseph E. Gonzalez, and Randy H. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 26:1–26:14. ACM, 2014.

[64] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, and Randy H. Katz. *Faster Jobs in Distributed Data Processing using Multi-Task Learning*, pages 532–540. 06 2015.

[65] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, and Randy H. Katz. Multi-task learning for straggler avoiding predictive job scheduling. *Journal of Machine Learning Research*, 17(106):1–37, 2016.

[66] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 452–465, New York, NY, USA, 2017. ACM.

[67] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 607–618. ACM, 2013.

[68] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *Proceedings of the USENIX Annual Technical Conference*, pages 309–322, 2016.

[69] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.

[70] Xiao Zhang, Rongrong Zhong, Sandhya Dwarkadas, and Kai Shen. A flexible framework for throttling-enabled multicore management (TEMM). In *Proceedings of the 2012 International Conference on Parallel Processing (ICPP)*, pages 389–398. IEEE, 2012.

[71] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 755–770, Savannah, GA, November 2016. USENIX Association.