Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments

Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, Anshul Gandhi
PACE Lab, Department of Computer Science, Stony Brook University
{sjavadi,amsuresh,mwajahat,anshul}@cs.stonybrook.edu

Abstract

Resource under-utilization is common in cloud data centers. Prior works have proposed improving utilization by running provider workloads in the background, colocated with tenant workloads. However, an important challenge that has still not been addressed is considering the tenant workloads as a black-box. We present Scavenger, a batch workload manager that opportunistically runs containerized batch jobs next to black-box tenant VMs to improve utilization. Scavenger is designed to work without requiring any offline profiling or prior information about the tenant workload. To meet the tenant VMs' resource demand at all times, Scavenger dynamically regulates the resource usage of batch jobs, including processor usage, memory capacity, and network bandwidth. We experimentally evaluate Scavenger on two different testbeds using latency-sensitive tenant workloads colocated with Spark jobs in the background and show that Scavenger significantly increases resource usage without compromising the resource demands of tenant VMs.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Virtual machines**; *Scheduling*; • **General and reference** → *Evaluation.*

Keywords

Cloud computing, resource utiliization, background workload

ACM Reference Format:
Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, Anshul Gandhi. 2019. Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357223.3362734

1 Introduction

Cloud computing allows tenants to rent economical and virtually unlimited resources, such as Virtual Machines (VMs), to deploy their applications. The cloud, public or private, is often hosted by a provider (e.g., Amazon [2] or Google [18]) on multiple servers in a data center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362734

Servers in cloud data centers often experience low resource utilization [10, 63]. A study focused on Amazon EC2 observed that cloud server usage is often below 10% [30]. A more recent study from Microsoft reported that cloud VMs hosted on Azure have low utilization; the study found that 60% of the VMs have an average CPU usage of less than 20% [6] (see Section 2).

To increase server utilization, prior works have proposed running provider's batch workloads, such as Hadoop or Spark jobs, next to tenant VMs opportunistically to leverage idle resources [19, 32, 68]. While effective, the key challenge with this approach is *interference* – the performance degradation of the colocated tenant VMs due to resource contention with batch workloads at the underlying host server. This interference can be caused by contention for *several resources simultaneously* [24]. Worse, this interference is *dynamic* due to resource demand variations in tenant and batch workloads [16, 67].

In an ideal cloud environment, provider (or *background*) workloads should run next to tenant (or *foreground*) workloads or VMs in such a way that their resource utilization complements that of the tenant VMs. The exact trade-off between performance isolation of tenant workloads and increase in resource utilization depends on the cloud environment and the provider, and should be tunable. In public clouds, performance isolation is key. In private clouds, such as clouds that operate within an organization, a balance is sought between performance isolation for specific high-priority workloads and modest increase in resource utilization. For best-effort clouds, such as community clouds [36], more aggressive resource management can be employed to improve utilization.

While there has been prior work on background workload management (see Section 8), there are specific shortcomings that are yet to be addressed satisfactorily. This is further evidenced by the recent study of production server usage at Alibaba (see Section 2) that found the average CPU and memory utilization to be at most 50% and 60%, respectively, *despite* (i) colocation of online and batch jobs, and (ii) oversubscription of resources [33].

(1) *Need for an application-agnostic, black-box approach.* Existing solutions often either (i) rely on historical usage patterns to predict the resource demand of foreground VMs [6, 67], or (ii) benchmark tenant VM performance to carefully colocate background workloads [9, 10], or (iii) regulate the resource usage of background workloads to avoid SLO violations for the foreground VMs [3, 22, 32]. Such solutions are ineffective and, at times, infeasible in cloud environments since tenants do not expect their VMs to be instrumented [41], and are not required to share their performance SLOs with the provider [15]. Even if foreground VMs can be profiled for a short time, there is often significant variation in tenant workloads that cannot be fully captured by a finite profiling run [24].

(2) *Need for a dynamic and tunable solution.* Another class of solutions focuses on careful VM placement to avoid interference in the first place [59]. However, dynamic changes in tenant loads can lead to interference *after* placement. Further, techniques like VM migration are not agile enough to be frequently employed on tenant VMs to mitigate the dynamic interference [11, 39]. We thus require solutions that are dynamic and can adapt to resource usage variations of the tenant workloads. Further, as discussed above, the solutions should be tunable depending on the performance isolation needs of the environment.

(3) *Need to address multi-resource interference.* While some recent works have proposed dynamic solutions, they often focus on a single resource, such as CPU [22, 62, 68]. Given that, for realistic workloads, several resources may simultaneously be under contention, such resource-specific solutions are inadequate [24].

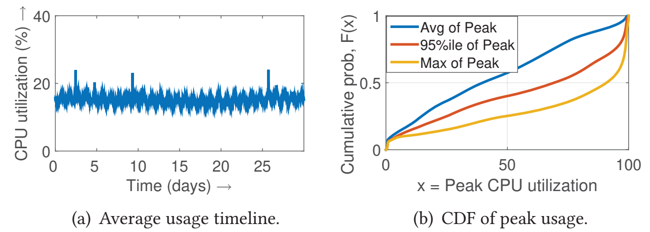
We present Scavenger, a provider-centric resource manager that dynamically regulates the resource usage of background jobs to complement the resource demand of black-box foreground workloads. We consider a cloud environment with tenant VMs as the foreground workload and Spark jobs (within the YARN framework [58]) in the background running on containers. We choose containers as the execution environment for batch jobs for agility in case we need to quickly regulate the background resource usage. Note that Scavenger is a batch workload manager and thus complements schedulers such as Borg [59].

Scavenger does *not* make any assumptions about the foreground workload and does *not* require any prior information about them. We do *not* profile their resource usage offline and do *not* instrument them. Instead, we treat the foreground workload as a *black box* and react to their resource demand in an online manner. This makes Scavenger application-agnostic in practice and easy to deploy.

The core idea of Scavenger's resource regulation algorithm is to use the mean and standard deviation of the foreground workloads' resource usage, over a window of observations, to obtain a statistically significant estimate of the opportunity for background usage. This approach is easy to implement, is analytically sound, and helps to immediately react to abrupt changes in the foreground workload's resource demand, including phase changes.

Scavenger regulates processor resources (including CPU and last-level cache (LLC)), memory capacity, and network bandwidth. Scavenger leverages cgroups for processor resource regulation and uses the Instructions-Per-Cycle (IPC) counter to track the impact on foreground VMs in a black-box manner. For memory capacity and network bandwidth regulation, we monitor the resource usage of foreground workloads and reactively scale (up or down) the resource consumption of batch job containers. In the worst case, if the foreground demand increases abruptly, we terminate the tasks running within the background containers to immediately release resources. We implement Scavenger as a daemon running on the server with less than 1% overhead.

Our experimental results on two different testbeds using latency sensitive foreground workloads from CloudSuite [14] and TailBench [25], colocated with Spark batch jobs, show that Scavenger can satisfactorily balance the trade-off between foreground performance isolation and increasing the server resource usage. Without Scavenger, foreground performance degradation is often higher than 50%, and can be as high as 10–20×. With Scavenger, the average performance degradation is less than 10%.



(a) Average usage timeline. (b) CDF of peak usage.

Figure 1: *VM-level CPU utilization for the Azure trace.*

We find that, under the black box requirements, while CPU regulation may not suffice by itself to address contention, when combined with LLC, network, and memory regulation, Scavenger significantly improves the utilization of multiple resources while mitigating contention; using Spark jobs in the background, Scavenger consistently increases server memory and CPU usage by more than 100%. We also conduct limit studies with resource-intensive microbenchmarks running in the background to highlight the performance isolation efficacy of Scavenger.

2 Background and Motivation

To motivate the need for Scavenger and identify its requirements, we analyze cloud data center resource usage traces from Azure (2016) and Alibaba (2018).

2.1 Azure VM-level resource usage traces

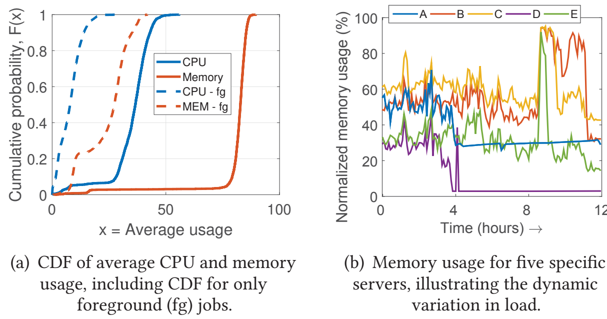
The Azure trace contains first-party VM CPU utilization data from one region [6]. The trace spans over 30 days and reports (only) CPU utilization (min, average, and max) of over 2 million VMs, every 5 mins, over their lifetime.

Figure 1(a) shows the timeline plot for average CPU utilization for every 5-min interval, averaged over all VMs that exist during that interval. We see that the average CPU utilization is quite low, typically less than 20%. Figure 1(b) shows the CDF of peak CPU utilization; the CDF is obtained by considering the average, 95%ile, and max of per-interval peak usages reported for each VM over their respective lifetime. We find that the median of the average, 95%ile, and max of peak usage is about 40%, 70%, and 90%, respectively. This shows that peak usage (over the lifetime) can be high when considering individual VMs. This observation also suggests that VMs (in the Azure trace) were likely provisioned for peak CPU usage.

Summary: *The VM usage pattern is variable enough to provide opportunities for colocation. However, since VMs may require full CPU capacity at some point, the colocated workloads need to be agile enough to relinquish resources. Also, since tenant load is hard to predict, and some VMs do require full capacity at some point, oversubscription of resources may not be feasible for all servers.*

2.2 Alibaba colocated job usage traces

The Alibaba production cluster traces [1] contain server-level CPU and memory usage sampled every 10s for about 4,000 servers over



(a) CDF of average CPU and memory usage, including CDF for only foreground (fg) jobs.

(b) Memory usage for five specific servers, illustrating the dynamic variation in load.

Figure 2: ***Analysis results for the Alibaba cluster trace.***

8 days. The servers had colocated online (or foreground) containerized jobs and background non-containerized batch jobs to increase resource usage; normalized usage of both jobs is also provided. However, performance/latency information for jobs is not provided.

The solid lines in Figure 2(a) show the CDF of average total utilization (foreground+background) for CPU and memory across all servers; the average is taken per server over the length of the trace. We also plot the average usage for only the foreground online jobs. We see that the average CPU usage is almost always less than 50%. If we consider only foreground, then average CPU usage is almost always less than 20%. Thus, while colocation helps, there is still room for improvement in CPU usage.

In terms of average memory usage, colocation helps significantly, with the average server-level usage typically exceeding 70%. The per-server peak memory usage numbers are also quite high, suggesting that most servers do require their provisioned memory capacity at some point during the 8 days of the trace duration. However, we do find instances where there is significant temporal variation in memory usage, representing an opportunity for improvement. To highlight the scope for improvement, we show specific examples of normalized per-server memory usage snippets in Figure 2(b). Server A has high memory usage in the first 4 hours, peaking at about 70% usage; however, thereafter, its memory usage is low, around 30%; we see a similar behavior for server D. Server E, on the other hand, has memory usage in the 20–40% range, except for the distinct peak of about 90% at the 9 hour mark; similarly for servers B and C.

Summary: *The above findings show that there is potential for improving resource utilization in data centers despite the current practices of colocation and oversubscription. The memory usage results show that it is critical for batch workload managers to be dynamic to fully realize the potential of improving resource usage via colocation.*

3 Novelty of Scavenger in the Context of Prior Work

There has been much prior work that focuses on improving cloud resource utilization by launching background jobs colocated with foreground (or tenant) workloads. Given the complexity of the problem, and the inherent trade-off between performance isolation and resource usage, this continues to be an active research topic; we are aware of at least 5 papers on this topic in 2018 [22, 31, 53, 57, 62] and at least 1 in 2019 [3]. While we discuss related work in detail in Section 8, we now highlight some of the prior works, classified according to the premise of the approach, to put our work in context.

- The first category of prior work considers a cluster where foreground workloads are also operated by the provider, e.g., Heracles [32], Borg [59], and Bistro [19], or where the foreground workload's performance can be monitored by the provider, e.g., PARTIES [3]. In such cases, the *performance requirements of the foreground workload are known a priori*, which allows the solution to accordingly regulate background usage.
- Another category of prior work assumes that foreground workloads' resource usage *can be predicted*, e.g., ResourceCentral [6], Zhang et al. [67], and TR-Spark [63], or *can be accurately profiled*, e.g., Paragon [9] and Cuanta [20]. The profiled or predicted resource usage pattern of the foreground workload is then used to tailor the resource consumption of the background workload(s).
- The third category focuses on regulating the usage of a single resource, such as CPU (e.g., MIMP [68]), LLC (e.g., dCat [62]), or network (e.g., QJUMP [21]).

We argue that there is considerable potential for research on improving the usage of ***multiple resources*** simultaneously by colocating batch jobs with ***black-box tenant VMs***; this defines the scope and novelty of Scavenger. The black-box requirement is realistic in public clouds as tenant VMs cannot (or should not) be instrumented, and also because their workload may change dynamically over time [24, 41]. Even if foreground VMs can be profiled for a short time, there is often significant variation in tenant workloads that cannot be fully captured by a finite profiling run [24]. The black-box assumption is also beneficial in private clouds as it avoids the overhead of profiling the workloads and tracking their performance. In contrast to existing approaches (Section 8) that either assume the tenant is a white box or require a one-time profiling of the tenant (e.g., PerfIso [22]), Scavenger is truly black-box, or application-agnostic, in nature.

4 Design of Scavenger

We consider a cloud data center with several physical machines (PMs), or servers, that host tenant VMs, which are referred to as ***foreground*** workloads or VMs or jobs; each PM may host several tenant VMs. We regard these VMs as *black-box* workloads with unpredictable resource consumption and unknown application SLO requirements. The only information the provider has is the resources requested by the tenant VMs and any metrics available at the host/hypervisor, such as resource usage and hardware performance counters. While the design of Scavenger is generic, in this paper we assume that the PMs run Linux.

To improve resource usage, providers can launch batch jobs colocated with the foreground VMs; we refer to such provider-owned batch jobs as ***background*** workloads or jobs. These could be complex data analytics workloads, such as Hadoop [54] or Spark [64] jobs, or simple computational jobs. Given their agility, we consider background jobs to be running on containers. Background jobs are controlled by the provider, and are not black box.

To address the resource contention between foreground VMs and the background containers, Scavenger monitors the resource

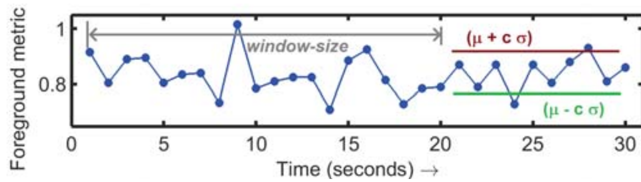


Figure 3: ***Illustration of Scavenger's generic algorithm.***

demand and performance counters of foreground VMs, and dynamically regulates the resource usage of the background jobs to satisfy the demands of the foreground. In the worst case, tasks within a container can be killed to immediately release resources for foreground VMs. We rely on the fault tolerance model of the batch job framework to prevent the entire batch job from being terminated if a subset of its tasks are killed. Commonly employed batch job frameworks, such as Spark [64] or Hadoop [54], already provide this fault tolerance feature by default; the failed task can be relaunched on a different container to continue job progress. The monitoring and resource regulation is managed via Scavenger daemons that run on each cloud PM, thus making Scavenger distributed in nature. In this paper, we consider contention for processor (including CPU and last-level cache (LLC)), memory capacity, and network bandwidth resources.

4.1 High-level overview of Scavenger's resource regulation algorithm

While the exact resource regulation algorithm is different for different resources, as we explain in the following subsections, the core idea is similar. At runtime, Scavenger periodically monitors specific metrics from the foreground VM, such as network usage or number of instructions executed, to estimate the range of resource requirements for the foreground VM(s). In our implementation, we use a monitoring interval of one second to balance responsiveness and low overhead, similar to prior work [20, 59, 62].

Initially, when the foreground VM starts executing, we do not allocate any resources to the background and instead monitor the foreground metrics for w seconds, where w is the tunable *window-size* parameter. Based on the observed metrics, say $\{x_1, x_2, \ldots, x_w\}$, Scavenger computes the sample mean, $\mu = (\sum_{i=1}^{w} x_i)/w$, and the sample standard deviation, $\sigma = \sqrt{(\sum_{i=1}^{w}(x_i - \mu)^2)/(w-1)}$. Since these empirical measures are known to be consistent estimators of the true underlying distribution [61], we obtain a statistically significant estimate of the foreground VM's resource demand as $[\mu - c \cdot \sigma,\ \mu + c \cdot \sigma]$, where c is a tunable parameter, referred to as *std-factor*. The probability that the resource demand lies in the $(\mu \pm c \cdot \sigma)$ range is higher when considering the sum of metrics of multiple foreground VMs [23], as suggested by the Central Limit Theorem.

Based on the obtained $(\mu \pm c \cdot \sigma)$ range, the generic Scavenger algorithm proceeds as follows:

(1) If the metric observed in the next interval is within the $(\mu \pm c \cdot \sigma)$ range, we consider the foreground VM's resource demands as being satisfied.
(2) If there is a significant deviation of the observed metric beyond this range, we consider this a phase change in the foreground workload and/or a violation, and react accordingly (as detailed in the following subsections).
(3) The difference between total resource capacity and $(\mu + c \cdot \sigma)$ can then be used by the background jobs; see details in the following subsections. Note that we reserve the $c \cdot \sigma$ capacity for the foreground as "headroom". If the foreground uses this headroom, Scavenger can infer the increased resource demand of the foreground and regulate the background usage accordingly.

Figure 3 illustrates an example scenario for our generic algorithm. The Scavenger algorithm is intentionally designed with tunable parameters, such as *std-factor* and *window-size*, to control the extent of colocation. This is helpful when applying Scavenger to specific environments; for example, Scavenger can be more aggressive in private clouds that tolerate some performance degradation.

4.2 Mitigating memory capacity contention

We closely follow the generic algorithm from Section 4.1 for regulating the memory allocation of the background jobs and use the per-second memory usage of the foreground VM as the monitored metric. Based on the initial *window-size* seconds of observation, we compute the sample mean and sample standard deviation and reserve $(\mu + c \cdot \sigma)$ for the foreground VMs; the remaining memory is allocated to the background containers.

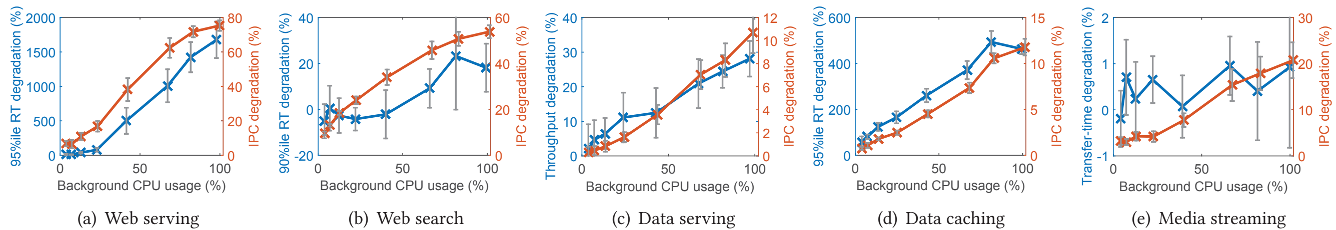
Any time the foreground memory usage, say m, goes above the $(\mu + c \cdot \sigma)$ upper limit, we treat it as a violation. When this happens, Scavenger immediately pauses or kills (depending on the implementation) a subset of tasks within the background containers to release the required memory. Additionally, Scavenger resets μ to be the current foreground memory value, m (that caused the violation), instead of using the moving-average μ computed over the last w (*window-size*) seconds. This is done for two reasons: (i) to quickly update (increase) the foreground reserved memory to $(m + c \cdot \sigma)$, and (ii) to essentially remodel the foreground memory usage behavior by resetting the μ estimation. Note that the σ computation (the sample standard deviation over the last w seconds) remains unchanged.

On the other hand, if the foreground memory usage goes below $(\mu - c \cdot \sigma)$ for w consecutive seconds, we treat it as a phase change for the foreground workload. When this happens, we recompute the new μ and σ over the last w seconds. Note that μ and/or σ are only reset when there is a violation or a phase change. Also note that when the memory usage is in the $(\mu \pm c \cdot \sigma)$ range, there is no change in the foreground or background memory allocation.

At all times, the difference between total memory and foreground reserved memory $(\mu + c \cdot \sigma)$ is allocated to background jobs. We discuss the black box sensitivity analysis for the tunable parameters c and w in Section 6.1.

4.3 Mitigating network contention

The network bandwidth regulation algorithm is similar to the memory regulation discussed above. We monitor the foreground traffic through the virsh interface every second. To regulate the background network traffic, we use Linux's traffic control mechanism [52]. In particular, we use the token bucket filter to enforce



(a) Web serving (b) Web search (c) Data serving (d) Data caching (e) Media streaming

Figure 4: *Impact of background LLC workload on CloudSuite performance (left y-axis) and its IPC (right y-axis).*

bandwidth limits on the background jobs' egress traffic; we do not impose any limits on the foreground workload traffic.

4.4 Mitigating processor cache contention

There are several processor resources that must be regulated, including cache and CPU cores. We first discuss the more challenging problem of regulating cache contention here, and then discuss CPU core contention.

Regulating the last-level cache (LLC) usage is complicated by the fact that we cannot easily regulate the cache access or capacity of the applications on a server. Newer processors, such as the Intel Xeon E5 v4 family, allow for fine-grained LLC capacity management via Cache Allocation Technology (CAT) [40]. In order to target generic processors, we do not assume access to CAT. We discuss how Scavenger can be integrated with CAT in Section 7.

Need for a metric to track cache contention. The difficulty in addressing cache interference is that there is no effective way to estimate the cache pressure created by a workload, as opposed to the easily available memory capacity and network bandwidth usage metrics. Prior work suggests that using the number of cache references or cache miss rate (CMR, monitored via performance counters) can help predict the cache requirements of a workload [62]. We find that this is not always the case.

We experimented with the SPEC CPU benchmark suite colocated with dCopy [7] (LLC microbenchmark) and found benchmarks, such as gcc and zeusmp, that have high cache references and CMR, but are not significantly impacted by dCopy. We also found examples, such as sphinx3 and tonto, where the CMR and cache reference rate is low, but the impact of dCopy is significant. This is because even a few cache references can lead to eviction of part of the working set of the foreground VM, resulting in significant latency impact. On the other hand, due to pipelining of instructions, some workloads can better tolerate cache interference.

Making the case for Instructions-Per-Cycle as a proxy metric. The Instructions-Per-Cycle (IPC) metric has often been used in computer architecture studies as a proxy for performance [13, 38, 44, 46, 65]. Some recent works have also used IPC and related metrics as a proxy for cloud workload performance [35, 66]. For Scavenger, the intuition behind using IPC as a proxy is that if IPC drops, we can consider this as an indication of processor cache contention, and thus an indication of cache pressure.

To make the case for using IPC as a proxy for foreground VM performance, we examine how IPC reacts to a drop in performance due to cache contention. We use a 4-core server and launch a 1-core foreground VM running one of five latency-critical CloudSuite workloads (see Section 5.3) and run the dCopy LLC microbenchmark [7] on a container using the other three cores; see Section 5.2 for details about our experimental setup. Note that there is no sharing of cores. To control the induced cache load, we add a sleep timer to the dCopy microbenchmark.

Figure 4 shows our experimental results for degradation in foreground IPC (right axis) and performance (left axis) when compared to the baseline (no background jobs), as a function of the background CPU usage. For each workload, we use the performance metric reported by the benchmark. We show the average and standard deviation bars in each case based on 10 runs of each experiment. As the background load increases (on the 3 cores allocated to it), we see that IPC and performance clearly degrade in a correlated manner for all workloads, except Media streaming. For Media streaming, the reported performance metric (transfer time) does not change much, despite a noticeable degradation in IPC. This is likely because Media streaming is network intensive, and does not use much CPU. Since a proxy is a must for the black box scenario, in the absence of a perfect proxy, we argue that, based on the above results, IPC is a viable (albeit far-from-perfect) alternative cache pressure proxy.

Processor cache regulation. The above results also show that simply partitioning the CPU cores, as in PerfIso [22], is not enough to avoid contention due to shared caches. However, the above results do suggest that we can mitigate the impact of background cache pressure on foreground performance (IPC) by limiting the amount of time the background runs on the processor. We thus cap the load induced by background containers by regulating their CPU quota (maximum CPU cycles given to a process under the Completely Fair Scheduler).

Our algorithm for regulating the CPU quota is based off of our generic algorithm framework in Section 4.1, with some subtle differences. To preserve the black box nature of Scavenger, we use IPC as the monitoring metric, measured every second (configurable), and compute the $(\mu \pm c \cdot \sigma)$ range based on IPC measurements. Note that for the memory and network regulation algorithms, the upper limit of the range, $\mu + c \cdot \sigma$, was used as an estimate of the amount of resources to be reserved for foreground. However, when using IPC as the metric, the upper limit does not directly correspond to the required CPU quota, thus providing no estimate of how much quota can be allocated to the background. Instead, when the foreground IPC is in the $(\mu \pm c \cdot \sigma)$ range, we consider this as an indication that the foreground has negligible cache contention and thus increase the background container's CPU quota by some fixed amount, *quota-increase*.

If the IPC drops below $\mu - c\sigma$, we decrease the background container quota by a fixed factor, *quota-decrease*, to reduce the cache contention. Finally, if IPC is beyond $\mu \pm 2 \cdot c\sigma$, we consider it as a phase change for the foreground and immediately drop the CPU quota of the background to a minimum value. We then wait for *window-size* seconds to reestablish the μ and σ for the foreground workload in its new phase. We discuss sensitivity analysis for the tunable parameters of the algorithm in Section 6.1.

Note that there is a potential weakness to employing IPC as a proxy for cache pressure. When the processor is not under contention, then IPC may not be a good proxy for cache pressure as it may degrade due to other resource contentions, such as network bandwidth. While Scavenger will detect and mitigate the other contentions in a timely manner, it will respond to the potentially degraded IPC metric by (unnecessarily) lowering the background container's CPU quota. As a result, the background job progress will be negatively impacted; however, the foreground workload performance will *not* be impacted.

4.5 Mitigating CPU core contention

As noted in prior work, sharing of CPU cores between foreground and background jobs can result in unpredictable contention [26, 27]. We tried setting the cpu.shares value under Linux's cgroups to prioritize foreground VMs over background containers, but this did not provide sufficient isolation (see Section 7). Instead, we consider the cores of the foreground VMs to be pinned and use cpuset to allocate only those cores to the background containers that are not being used by the foreground. This prevents any contention, including for per-core caches, that arises by sharing of cores.

5 Evaluation Methodology

This section describes the evaluation methodology we employ for the performance evaluation results presented in Section 6. We start by detailing our Scavenger prototype implementation, followed by our experimental setup and the workloads we employ for evaluating Scavenger.

5.1 Scavenger prototype implementation

Our prototype implementation for the Scavenger daemon is largely written in C++. The main Scavenger background daemon combines the resource regulation algorithms from Sections 4.2 – 4.5 into a single process. Given its design, the core Scavenger algorithm is easy to implement, requiring about 750 lines of code. For CPU management, our Scavenger daemon interacts with the Linux cgroups subsystem; we use a simple shell script to achieve this result. The daemon constantly monitors the respective resources (via virsh [29]) and IPC (via hardware performance counters) of the foreground VMs. Based on the algorithms, the daemon changes the resource allocation of the background containers dynamically using resource-specific mechanisms: TC [52] for network, cpuset for core allocation, CPU quota for processor, and YARN APIs for memory. Our Scavenger daemon implementation results in about 1% cpu overhead, on average. Note that Scavenger does not require changes to the kernel or to YARN.

Deployed architecture: Figure 5 illustrates our Scavenger deployment on a cluster of cloud physical machines (PMs), which are assumed to be under the control of the provider. The orange boxes

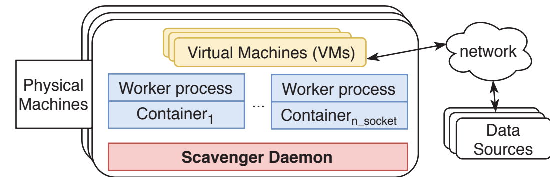


Figure 5: *Illustration of our Scavenger deployment.*

on each PM in Figure 5 represent foreground tenant VMs whose workload is considered to be an unknown (black-box). The blue boxes represent background job containers; these could be running worker processes of distributed data processing frameworks such as Hadoop and Spark (see Section 5.4). The worker processes read from/write to the data sources via the network. Each PM runs our Scavenger daemon (red box) that interacts with the foreground VMs and background containers. We next explain the specific experimental setups we employ for evaluating Scavenger.

5.2 Experimental setup

We use two different sets of servers for our experiments.

Lab testbed: Each server has 1 socket with 4 cores (Intel Xeon E3 v3, 3.4GHz), sharing an 8MB L3 cache; and 16 GB memory. Servers are connected via 1Gb/s links.

Cloud testbed: In this CloudLab testbed [4] (Clemson site), each server has 2 sockets with 10 cores each (Xeon E5 v2, 2.2GHz), and a 25MB L3 cache per socket; and 250 GB memory. Servers are connected via 10 Gb/s links.

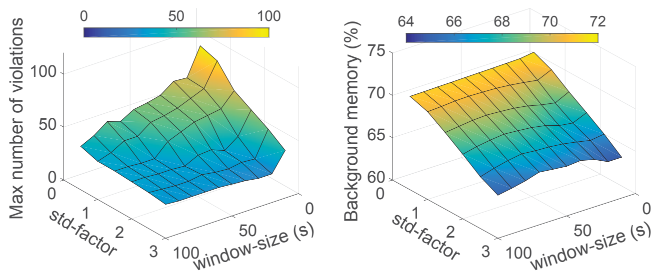
We use KVM (on top of Ubuntu 16.04) to deploy VMs on these PMs; the size of the VM is dictated by the foreground workload. For background jobs, we use Docker (v18.03) to launch containers.

5.3 Foreground workloads

We employ the following latency-critical workloads, representative of realistic online services, as the foreground application to evaluate the efficacy of Scavenger:

CloudSuite [14]. We use the latest version, CloudSuite 3.0 [43], which provides eight workloads, of which five can be categorized as latency-sensitive workloads.

(1) *Web serving* is a PHP-MySQL-Memcached based multi-request class social networking benchmark application. 95%ile response time is used as the performance metric.
(2) *Web search* deploys Apache Solar search engine to respond to simulated clients' web search requests. 90%ile response time is used as performance metric.
(3) *Data serving* uses Yahoo Cloud Serving Benchmark (YCSB) [5] to generate loads for a Cassandra data store. Throughput is used as the performance metric.
(4) *Data caching* employs a Memcached caching server and Twitter dataset to simulate the behavior of Twitter. 95%ile response time is used as performance metric.
(5) *Media streaming* uses Nginx as a streaming server for hosted videos and httperf as the client that requests videos. Average transfer time across all requests is used as the performance metric.



(a) Maximum number of violations (lower is better).

(b) Background memory afforded (higher is better).

Figure 6: *Sensitivity analysis for std-factor and window-size parameters of the memory regulation algorithm.*

TailBench [25]. TailBench is a recent benchmark suite specifically designed for analyzing latency-critical applications. There are eight workloads in the suite, all of which use 95%ile response time as the reported performance metric (further details can be found in the TailBench paper [25]): (i) *xapian*, an online search benchmark using the Wikipedia dataset as search index; (ii) *moses*, a statistical machine translation application using the opensubtitles.org English-Spanish corpus; (iii) *silo*, an in-memory database application driven using TPC-C [56]; (iv) *specjbb*, an industry-standard Java middleware benchmark [50]; (v) *masstree*, a key-value store application driven using YCSB [5]; (vi) *shore*, an on-disk database driven using TPC-C [56]; (vii) *sphinx*, a speech recognition system driven using the AN4 audio dataset [49]; and (viii) *img-dnn*, a handwriting recognition application driven using the MNIST images database [12].

All of the above workloads employ their own custom load generators, resulting in dynamic load variations (in the range of 10–60% CPU load in our experiments).

5.4 Background workloads

We employ microbenchmarks and Spark jobs as our background workloads; microbenchmarks are used as adversaries to stress test the performance of Scavenger.

Microbenchmarks. We employ the following for our adversary studies: (i) *dCopy* [7] copies vectors repeatedly to stress the cache; (ii) *stress-ng* [51] is a cpu stress benchmark; and (iii) *iperf* [55] is a network bandwidth measurement tool that we employ to stress the network.

Spark jobs. Spark [64] is a scalable and resilient distributed data processing framework that is popularly employed for iterative machine learning jobs. Spark jobs rely on distributed storage platforms to store their job data. In our deployment of Spark (v2.3), we use the distributed HDFS [47] as the storage core. We also employ Yarn [58] (v3.1), a resource management framework that manages the cluster resources and schedules user applications, to manage background jobs. For the Spark workload, we employ analytics jobs from BigDataBench [17] and Spark-Bench [48], such as FFT, KMeans, Sorting, etc.

6 Evaluation Results

We now present our evaluation results for Scavenger. We start with sensitivity analysis results to configure Scavenger, and then present our main evaluation results on both testbeds using Spark jobs in the background. Finally, we discuss our adversarial (limit) study using microbenchmarks in the background to evaluate the performance isolation of Scavenger under stress. Where possible, we evaluate the impact of the foreground and background workload's resource demand on Scavenger's ability to improve utilization.

6.1 Sensitivity analysis

We use sensitivity analysis to determine the parameter values to be used for the resource regulation algorithms from Section 4.2 – 4.4; note that the CPU cores regulation algorithm from Section 4.5 has no tunable parameters. Our analysis must be black-box and should not involve workloads that will serve as foreground in the evaluation.

Memory regulation algorithm sensitivity analysis. To determine the right values for the *window-size* and *std-factor* parameters of our memory regulation algorithm from Section 4.2, we require a black-box approach that does not involve the foreground workload. We resort to simulations for sensitivity analysis and use the recent Alibaba traces [1] containing foreground memory usage, sampled every 10s, for about 4,000 servers for 8 days.

Figure 6 shows the impact of different *window-size* and *std-factor* parameter settings on the maximum number of violations (across all traces) and the average background memory afforded. In general, a lower *std-factor* (c) favors available background memory but results in high violations (i.e., not being able to meet the memory demand of foreground). This is because lower the c value, lower is the amount of memory reserved for foreground ($\mu + c \cdot \sigma$), see Section 4.2. Likewise, a lower *window-size* results in higher violations as there is insufficient data for accurately (re)estimating μ and σ. While the parameter values can be set by the provider per their needs, we choose values that maximize the afforded background memory while resulting in fewer than 30 violations: *std-factor* = 2 and *window-size* = 60s. We use these values for memory regulation in subsequent evaluations.

Network regulation algorithm sensitivity analysis. We use a similar black-box approach to choose the parameters for network regulation. Since the Alibaba traces do not have enough information to obtain network utilization values, we use network traffic traces from WITS [60] for our sensitivity analysis. Our analysis suggests that *std-factor* = 2 and *window-size* = 30s work well.

Processor cache regulation algorithm sensitivity analysis. Employing the same trace-driven approach as above for cache regulation algorithm is infeasible as we require information on how the foreground IPC will degrade under different algorithm parameters. Instead, we conduct actual experiments using the CloudSuite workloads in foreground and dCopy in background; we do not use Media streaming workload as it will later be employed as foreground for evaluating network contention. To preserve the black-box nature of Scavenger, we will not use the CloudSuite workloads employed here when evaluating cache regulation in the subsequent evaluation

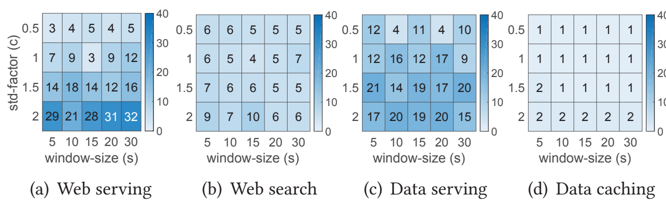


Figure 7: ***Degradation of foreground IPC (lower is better) colocated with dCopy under processor regulation.***

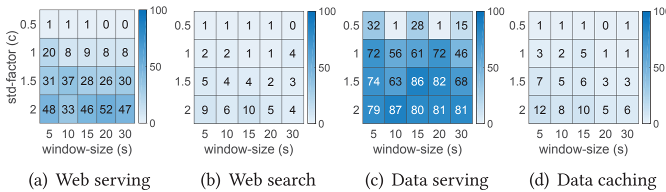


Figure 8: ***Background CPU usage afforded (higher is better) under the processor regulation algorithm.***

subsections; instead, we will use TailBench, which is not employed for sensitivity analysis.

There are four parameters for cache regulation algorithm (see Section 4.4): *quota-increase*, *quota-decrease*, *std-factor*, and *window-size*. For *quota* parameters, we use the AIMD (additional increase multiplicative decrease) approach, inspired by TCP congestion control [42], for exploring the parameter range. We vary *quota-increase* from 1% to 30% of a CPU core, and vary *quota-decrease* by various multiplicative factors. For each pair of *quota* parameters, we vary *std-factor* from 0.5 to 2, and *window-size* from 5s to 30s. We use the Lab testbed and employ the CloudSuite workloads in the foreground on a 1-vCPU VM and run dCopy on a container in the background on the remaining 3 cores. While we perform several experiments across all parameter ranges, we briefly highlight our results below.

We find that *quota-increase* of 10% CPU core and *quota-decrease* of 2 (halving the quota) works well. For this pair of parameter settings, our sensitivity analysis for *std-factor* (also referred to as *c*) and *window-size* is shown in Figure 7 and 8, which evaluate the foreground IPC degradation (lower is better) and background CPU usage afforded for dCopy (higher is better), respectively; we report the average numbers based on 3 runs. We see that some workloads, such as Data caching and Web search, are less sensitive to parameter variations, whereas others, such as Web serving and Data serving, are highly sensitive. Recall, from Section 4.4, that we increase background quota when the foreground IPC is in the $(\mu \pm c \cdot \sigma)$ range; thus, a larger value of c affords larger background usage, but at the expense of foreground IPC degradation (due to increased colocation). For *window-size*, the impact is less pronounced and not monotonic. While tunable per provider's needs, we set *std-factor* = 1 and *window-size* = 15s to limit the IPC degradation, which is our black-box proxy for performance degradation.

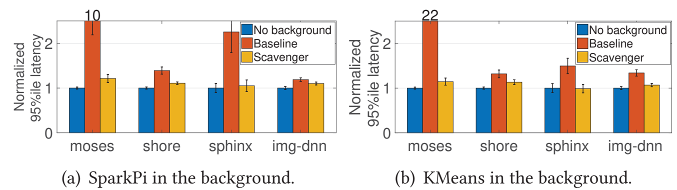


Figure 9: ***Performance degradation of individual TailBench workloads in Lab testbed colocated with Spark.***

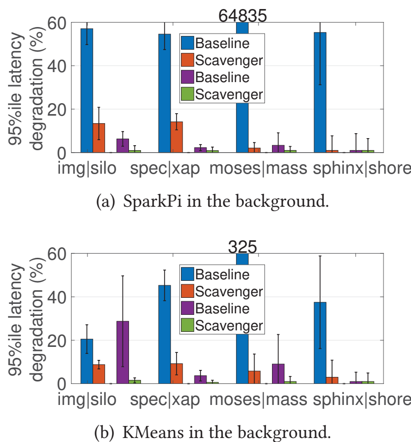
6.2 Evaluation with Spark jobs as the background batch workload

We now present our evaluation results with Spark jobs running in the background and the Scavenger algorithms tuned per the above sensitivity analysis results. Each experiment is typically run multiple times, with each run lasting for 360s, including a 60s warm-up period. We compare Scavenger with the case of no background and the black-box baseline case of cpu core isolation via cpuset. We do not compare with white-box approaches such as Dirigent [69] or Bistro [19] since they require SLO and latency monitoring of the foreground workload.

TailBench workloads as foreground. We start with the case of TailBench workloads in the foreground. We perform experiments on both testbeds. For the Lab testbed, we run a TailBench workload on a 1-vCPU VM and use the remaining 3 cores (via cpuset) to launch Spark containers; this 1:3 core allocation represents the case of heavy background usage. Figure 9 shows the average performance degradation compared to the case of no background, for baseline (no Scavenger but with cpuset) and Scavenger, based on 10 runs for each workload. We show results for four workloads that exhibit sensitivity to colocation; the performance of the other TailBench workloads was not much impacted by background Spark jobs.

For all cases, we see that, compared to the baseline, Scavenger significantly reduces the performance degradation of TailBench due to background Spark jobs, often to less than 10%. The average foreground degradation under baseline is 283% and 572%, respectively, when colocated with SparkPi and KMeans. If we omit the highly sensitive *moses* workload, the average degradation is still 61% and 39%. By contrast, the average degradation under Scavenger is 12% and 8%, respectively, when using SparkPi and KMeans in the background. Compared to baseline, Scavenger reduces the performance degradation by 78% and 85%, respectively, when using SparkPi and KMeans in the background. Note that the baseline here represents the case of only regulating CPU cores; clearly, such an approach does not suffice to mitigate cache contention.

In terms of utilization, Scavenger increases average CPU usage across all workloads, compared to no background, by about 170% and 198%, respectively, when using SparkPi and KMeans in the background. Likewise, the memory usage increases by 142% and 230%, respectively. The highest gains in CPU usage, of about 350%, are for *specjbb* (in the foreground) while the lowest gains, about 37%, are for the highly sensitive *moses*. We further analyze the impact of the workload's resource pressure on Scavenger's ability to improve utilization in Section 6.3.



(a) SparkPi in the background.

(b) KMeans in the background.

Figure 10: ***Performance degradation of multiple TailBench workloads in Cloud testbed colocated with Spark.***

Multiple foreground VMs. We now use the Cloud testbed and run two foreground TailBench workloads simultaneously on 2-vCPU and 8-vCPU VMs, one on each socket, illustrating the case of multiple foreground VMs hosted on the same physical machine. The remaining 8 cores of socket 0 and 2 cores of socket 1 are used to host Spark job containers. Of the 8 TailBench workloads, we pick 4 random unique pairs and report our results for these settings, averaged over 5 runs.

Figure 10 shows the latency degradation results over no background for baseline and Scavenger. For each set of 4 bars, the first 2 bars refer to the 2-vCPU TailBench VM on socket 0 and the last 2 bars refer to the 8-vCPU TailBench VM on socket 1; the TailBench workloads are denoted in the x-axis labels (abbreviated in some cases). Clearly, the foreground latency degradation under baseline can be quite high, often exceeding 50%. The average degradation when colocated with KMeans is 56%, and that when colocated with SparkPi is greater than 100% (due to the very high degradation for *moses*). By contrast, the degradation under Scavenger is almost always less than 15%, with average degradation of 4.8% when colocated with SparkPi and 5.6% when colocated with KMeans. Compared to baseline, Scavenger reduces the foreground latency degradation by 61.7% and 67.2%, respectively, when the foreground is colocated with SparkPi and KMeans.

In general, the degradation is much higher for the first TailBench workload that is hosted on 2 vCPUs and is colocated with an 8-core Spark job; this is because of the increased resource demand created by the larger-sized background job. We confirmed this by reversing the configurations of the TailBench workload pairs in Figure 10; Scavenger continued to significantly outperform baseline, with the improvement over baseline ranging from 20.1% to 97.5%. Note that the results for TailBench degradation are largely consistent with those from Figure 9; *moses* continues to be most sensitive to contention.

In terms of utilization, Scavenger increases average CPU usage across all cases, compared to no background, by 43% and 34%, respectively, when using SparkPi and KMeans in the background. The memory usage increases more significantly, by 201% and 321%, respectively.

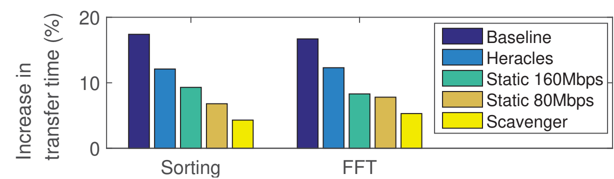


Figure 11: ***Performance degradation of Media streaming (in Lab testbed) when colocated with Spark jobs.***

Media streaming as foreground. For evaluating the network regulation of Scavenger, we consider the Media streaming workload from CloudSuite. All other foreground workloads we consider have low network bandwidth usage. For background, we consider the Sorting and FFT Spark workloads from BigDataBench since they have high network usage. We use the Lab testbed with foreground running on a 2-vCPU VM and background container running on the remaining 2 cores of the same socket. When there is no background, Media streaming consumes network bandwidth in a dynamic manner, with an average usage of about 268Mbps (out of the 1Gbps available capacity); in isolation, the average transfer time (performance metric) for foreground is 530ms.

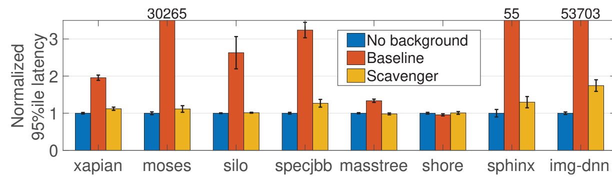
Figure 11 shows our results, averaged over 3 runs, for different background jobs under network regulation. We show results for baseline (no regulation), Heracles network regulation, static background limits (via TC [52]), and Scavenger network regulation; for Heracles, we implement the regulation algorithm from the paper [32], running at the same frequency ($1s^{-1}$) as Scavenger.

We see that the performance degradation for Media streaming under no regulation exceeds 15%. Heracles only reduces this degradation to about 12%; this is because Heracles assumes a stable network usage and thus reserves only a small buffer bandwidth. However, Media streaming has dynamic network usage, which is not well handled by Heracles. The static limits approach works moderately well, but requires (white box) trial-and-error to find the right limits. By contrast, the dynamic Scavenger algorithm reduces the degradation to 4.3% in case of Sorting as background and to 5.3% in case of FFT; this represents a more than 3× improvement over baseline.

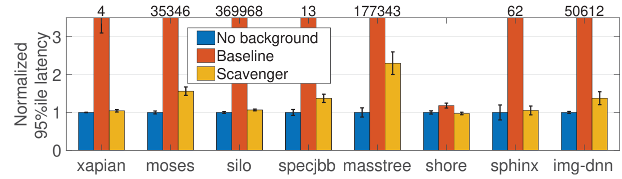
In terms of background network usage, baseline and Heracles afford about 320Mbps and 290Mbps, respectively, for Spark. Under the static approaches, Spark uses almost the entire set limit (80Mbps and 160Mbps). Under Scavenger, we afford about 180Mbps (and 32–43% additional CPU usage) for Spark. Given its dynamic nature, Scavenger outperforms static approaches while affording higher background usage.

6.3 Limit study with stress microbenchmarks

The impact of colocation on foreground performance depends on the resource demand created by background jobs. We now conduct a limit study to evaluate the performance isolation of Scavenger by colocating stress-test microbenchmarks in the background that serve as adversarial or "worst-case" workloads as they consume all available resources and create substantial contention. The objective here is to assess whether Scavenger can sufficiently regulate the background workload, even completely throttling the background, if needed, to minimize the performance impact on the foreground.



(a) Lab testbed: 1-vCPU foreground VM, 3-core background container.



(b) Cloud testbed: 4-vCPU foreground VM, 6-core background container.

Figure 12: *Performance degradation of foreground TailBench, colocated with dCopy, under processor regulation.*

Processor regulation with dCopy as background. For this limit study, we only employ the processor cache regulation algorithm to focus on cache contention.

Figure 12(a) shows the results of our Lab testbed experiments with TailBench in the foreground on a 1-vCPU VM and dCopy container in the background on the remaining 3 cores; the last-level cache is shared and under contention. We report average values and show standard deviation bars based on 10 runs. The performance (95%ile latency) is normalized to that of the foreground when run in isolation (no background). Note that the y-axis is capped to allow for a meaningful comparison.

Clearly, the baseline (no Scavenger but with cpuset) results in very high latency for almost all workloads; the numbers are especially high for *moses*, *sphinx*, and *img-dnn*. The high latencies under baseline highlight the severe performance impact of our background adversary workload on the foreground. The median increase in latency for baseline compared to no background is 193%. This reaffirms the fact that simply isolating CPU cores will not suffice to mitigate contention. By contrast, the latency is much lower with Scavenger; the median increase in latency compared to no background is about 11%. For *img-dnn*, Scavenger significantly improves upon the baseline, but the latency increase is about 60% compared to no background. This is likely because the IPC for cache-intensive *img-dnn* is not as sensitive to cache contention as its performance, thus the black-box Scavenger is not fully aware of the degradation. Nonetheless, given that this is a limit study, the performance degradation numbers are encouraging; without Scavenger, the baseline numbers are 158% higher, on average.

In terms of utilization, when colocated with the cache-intensive dCopy, Scavenger increases average CPU usage across all workloads, compared to no background, by about 127%. We find that the CPU usage improvement is lower when the foreground workload is more sensitive to LLC contention (e.g., *moses*), highlighting the difficulty in maintaining acceptable latencies and affording higher background resource utilization for such workloads. We also repeated the above set of experiments by replacing dCopy with the CPU-intensive stress-ng microbenchmark [51] in the background. We observed negligible degradation for the foreground workloads, but a more impressive CPU usage improvement of 285%. In summary, for the Lab testbed, Scavenger improves the CPU utilization on average by 127%, 184%, and 285%, when the background workload is dCopy (very cache intensive), Spark jobs (moderately cache intensive), and stress-ng (mildly cache intensive), respectively. This suggests that Scavenger's ability to improve utilization is inversely proportional to the background workload's resource (cache, in this case) pressure.

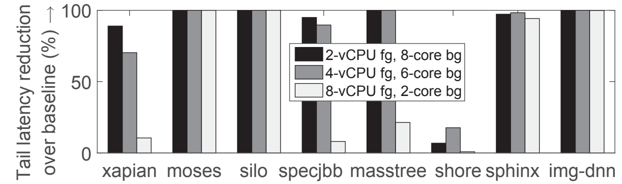


Figure 13: *Scavenger's latency reduction over baseline for different foreground (fg) and background (bg) sizes.*

Figure 12(b) shows the results of our Cloud testbed experiments with TailBench in the foreground on a 4-vCPU VM and dCopy in the background on 6 cores. As before, the y-axis is capped to ease the comparison of results. At a high-level, the results are consistent with those for our Lab testbed, illustrating the versatility of Scavenger. However, under baseline, we see very high degradation for *silo* and *masstree* (both of which are memory intensive). We believe there are two reasons for this high degradation: (i) the load generators employed in TailBench are open-loop [45], so the backlog created at the foreground due to resource contention continues to grow indefinitely; and (ii) the "no background" latencies are quite small for these workloads (less than a millisecond) and so the relative latency for baseline is amplified. Omitting these two workloads, compared to no background, the median increase in latency is about 3674% for baseline, but only about 21% for Scavenger, representing almost 99% improvement over baseline. However, these latency improvements come at the expense of negligible background resource usage (only about 3–5%), highlighting the fact that Scavenger can successfully and aggressively regulate background workload to mitigate its impact on foreground performance.

To evaluate the efficacy of Scavenger for different foreground and background load, we repeat the above Cloud testbed experiments with different configurations of the TailBench VM and dCopy container sizes. Figure 13 shows the percentage tail latency reduction afforded by Scavenger over baseline for 2-vCPU, 4-vCPU, and 8-vCPU foreground TailBench VMs, colocated respectively with 8-core, 6-core, and 2-core dCopy containers. In general, Scavenger's benefits are more pronounced when the background load is higher, since there is greater need for performance isolation in this case. Nonetheless, in almost all cases, the improvement over baseline is significant. For *moses*, *silo*, *sphinx*, and *img-dnn*, the latency reduction over baseline is very high under all configurations; this is because the baseline resulted in severe performance degradation for these workloads (see Figure 12(b)).

Network bandwidth regulation with iperf as background. For this limit study, we only employ the network bandwidth regulation algorithm. We use the Lab testbed with Media streaming foreground

running on a 2-vCPU VM and a 2-core background container running iperf. We report average results based on 3 runs. When using the default *std-factor* setting of 2, Media streaming's transfer-time increases by about 4.8% as a result of 2 violations (meaning the foreground required more bandwidth than reserved for it by Scavenger). In terms of background bandwidth usage, of the remaining nearly 700Mbps (Media streaming uses 268Mbps on average), iperf consumes 115Mbps under Scavenger's network regulation.

Figure 14 shows the results for *std-factor* settings of 0.5, 1, 1.5, and 2, to illustrate the trade-off between foreground performance and background resource usage afforded by the tunable parameters of Scavenger. If we are willing to allow more violations, iperf can use 421Mbps, representing a combined network usage of 68%, as opposed to just 27% when there is no background.

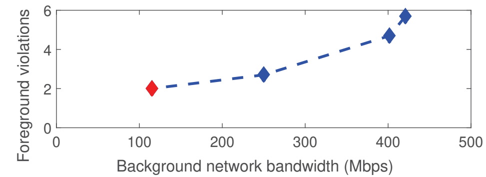
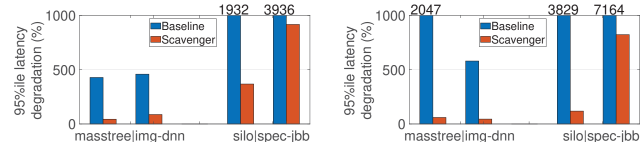


Figure 14: ***Trade-off between foreground violations and background bandwidth for different* std-factor *settings. The default* std-factor *setting of 2 is shown in red.***



(a) SparkPi in the background. (b) KMeans in the background.

Figure 15: ***Performance degradation of multiple TailBench workloads colocated with Spark, when employing core sharing.***

7 Discussion

Core sharing. One of the design decisions in Scavenger is to prevent core sharing between foreground and background workloads to provide performance isolation. As noted in Section 4.5, Linux's cgroups provides cpu.shares setting to prioritize different processes. To evaluate the performance impact of using cpu.shares to share CPU cores between foreground and background workloads, we consider a testbed with multiple servers each of which has 2 sockets with 4 cores (Intel Xeon L5520, 2.27GHz) and 8MB L3 cache each, 32 GB memory, and 1Gb/s network links. We launch one 4-vCPU background container and one 2-vCPU foreground VM on each processor socket. Thus, the foreground VMs share their assigned physical cores with the containers. We set the cpu.shares value for the containers as 2 (minimum allowed value) and that for foreground VMs as 262,144 (maximum allowed value), thus completely prioritizing the foreground over the background.

Figure 15 shows the latency degradation results over no background for baseline (no Scavenger) and Scavenger, both with core sharing, averaged over 4 runs. We illustrate the results for four different scenarios using TailBench workloads for foreground and Spark in the background: (i) *masstree* and *img-dnn* as the two foreground workloads, and SparkPi as the background (on both containers), (ii) *silo* and *spec-jbb* as the two foreground workloads, and SparkPi as the background; scenarios (iii) and (iv) are same as above, expect that we run the more cache-intensive KMeans as the background. We see that, with core sharing, Scavenger does not provide good performance isolation for the foreground, resulting in as much as 900% degradation compared to the latency under no background. By contrast, when we disable core sharing, the latency degradation is much lower, as shown in Figure 10. In terms of background resource usage, we find that there is less than 10% CPU usage increase. These results suggest that core sharing severely impacts foreground performance, thus validating our design decision to disable core sharing when employing Scavenger.

Tunable parameters. Our experimental results show that Scavenger affords different trade-offs between performance isolation and resource usage improvement depending on the sensitivity of the foreground and background workloads to resource contention. The exact trade-offs can be tuned via the algorithm parameters, such as *std-factor*, that were intentionally included in the design of Scavenger.

Tolerance for performance degradation. Our results also show that there are some workloads, such as *img-dnn*, that are very sensitive to contention. In such cases, if no foreground performance degradation can be tolerated, provider workloads should not be run in the background or a more accurate black-box proxy for foreground performance should be sought. As discussed in Section 4.4, finding such black-box proxy metrics is challenging.

Extension to Cache Allocation Technology (CAT). While we did not have access to CAT-equipped servers in our testbed, we believe that the processor regulation algorithm of Scavenger can benefit such servers as well. Instead of regulating LLC contention using CPU quota, we can directly employ CAT to dynamically resize the cache allocation between foreground and background, via our IPC-based regulation algorithm.

8 Related Work

Improving resource utilization in private clusters: The problem of resource underutilization has been around since before shared public clouds. Heracles [32] combines software and hardware isolation mechanisms to run batch jobs next to latency sensitive jobs. Heracles considers dedicated clusters where the provider is aware of the foreground application and its SLOs, and can benchmark the performance of foreground jobs with different levels of colocation. Since Heracles requires knowledge of foreground SLOs, it is not applicable in cloud environments where (black box) tenant SLOs are application-specific and not known to the provider. However, the Heracles' network sub-controller by itself does not require SLO information and was thus employed for comparison in Figure 11.

PARTIES [3] is a recently proposed resource controller that mitigates SLO violations between colocated latency-sensitive applications using software and hardware mechanisms. PARTIES requires knowledge of the applications' SLO and current performance information, making it suitable for cloud environments where such information is available to the provider; by contrast, Scavenger does not require such information, making it more broadly applicable.

Borg [59] is Google's cluster manager that runs Google's jobs on their clusters. All job tasks are run in cgroup-based containers and are assigned priorities based on their functionality. Since

all tasks are known to Borg, it is aware of their resource requirements and priorities. Scavenger has a similar goal of performance isolation as Borg, but we consider foreground jobs as black-box tenant VMs. Further, all tenant VMs have to be treated as having the same (high) priority in our case. Bistro [19] is a job scheduler that runs data-intensive batch jobs next to online tenant workloads in Facebook's production systems. To avoid disrupting foreground jobs, Bistro manually constrains the resource capacity allocated to batch jobs based on the known characteristics of the foreground jobs. However, in cloud environments, the foreground job workload (and its characteristics) may change unpredictably, limiting the applicability of Bistro.

PerfIso [22] is a black-box approach for isolating the CPU interference between foreground and background jobs by reserving some buffer CPU cores to accommodate the load variations in foreground jobs. However, as acknowledged by the authors, PerfIso does require a critical *one-time performance profiling* of the foreground to determine the extent of load variations that the foreground workload will experience, allowing PerfIso to reserve the number of buffer cores accordingly. Thus, if the foreground workload changes dynamically, the profiling step may have to be repeated frequently. Further, as we show throughout our results, isolating CPU cores alone does not mitigate processor cache contention.

Improving resource utilization in public cloud servers: In public cloud environments, the foreground (tenant) VMs cannot be controlled and their resource demands should be met at all times based on their VM sizes. Zhang et al. [67] rely on historical usage patterns of CPU and disk usage to predict the required resources for tenant VMs; the remaining spare compute cycles and storage space are then leveraged by the provider's batch workloads. Resource Central [6] uses a similar approach to colocate production and non-production VMs on Azure cloud servers to increase CPU utilization.

TR-Spark [63] runs Spark on transient VMs, such as spot instances, which can be used by the provider in the background. The key idea is to introduce checkpointing to allow job progress despite worker failures by modifying Spark's Task Scheduler. MOON [28] provides a similar solution, but for Hadoop jobs. However, TR-Spark relies on prediction of worker failures, suggesting that changes in the foreground workload can be predicted. In general, tenant workloads need not follow specific patterns and may not be predictable [16]; the performance and revenue loss due to mispredictions can be substantial [8].

Regulating the usage of specific resources: dCat [62] presents a cache performance isolation approach by exploiting the CAT technology (cache allocation technology [40]) on Intel's newer x86 machines to dynamically resize the *cache allocation* based on the needs of the workloads. However, dCat can only be used on servers equipped with CAT. QJUMP [21] addresses *in-network interference* by defining priority levels for packets, allowing foreground job packets to jump-the-queue over background job packets. MIMP [68] proposes a similar *CPU* scheduling policy that allows background Hadoop jobs to run only when foreground VMs are not actively utilizing the CPU. CPI^2 [66] employs statistical approaches to analyze an application's Cycles-Per-Instruction (CPI) metric to detect and mitigate *processor* interference between threads of different jobs. The generic idea in CPI^2 of analyzing CPI to detect resource contention is similar to Scavenger's use of IPC to detect foreground cache pressure; however, unlike CPI^2, Scavenger also leverages IPC, via a moving window approach, to detect phase changes in the workload. Tableau [57] is a scheduler for Xen that mitigates CPU interference among VMs (all foreground) by scheduling them according to their complementary resource demands. Dirigent [69] is a white-box solution that profiles the execution of foreground jobs and uses this profile to yield processor resources when the foreground is making good progress. PerfGreen [53] uses a similar idea to leverage idle cores for running batch jobs.

The above works target a specific resource contention. In general, several resources may simultaneously be under interference [24, 34]. Further, as we show in our results, e.g., Figure 4, managing the contention for a single resource, such as CPU cores, may not suffice.

Mitigating interference among colocated workloads/VMs: Bubble-Up [37] presents a resource usage characterization methodology that uses a "Bubble" program capable of applying variable pressure to the memory subsystem of a server. Based on profiling of the Bubble program when colocated with a foreground application, the authors learn how much memory pressure the application generates. Cuanta [20] presents a similar approach to estimate the cache usage behavior of applications. Paragon [9] uses collaborative filtering techniques for quickly classifying an incoming application based on the performance interference it causes and can tolerate. In an offline step, a few applications are run across different server configurations and against multiple microbenchmarks for training. DIAL [24] also uses profiling to estimate the intensity of interference before deciding on load balancing strategies.

The above works require profiling of foreground applications to infer their resource requirements; however, this may not be feasible in cloud environments where workloads can change unpredictably. Consequently, tenant VMs may exhibit dynamic resource usage patterns which cannot be captured by limited profiling.

9 Conclusion

This paper presents Scavenger, a dynamic, black-box multi-resource manager that improves resource utilization in cloud servers. Scavenger works by colocating batch job containers with black-box tenant VMs on host servers and dynamically regulating the resource usage of batch jobs to meet the resource demands of the VMs. Importantly, Scavenger does so without instrumenting or offline profiling the tenant VMs. Experimental results on different testbeds show that Scavenger increases server usage without compromising the resource demands of tenant VMs. In general, Scavenger's ability to improve server usage is inversely proportional to the tenant and batch workload's resource demand. By regulating the batch workload's resource consumption, Scavenger mitigates the latency degradation of tenant workloads in all cases.

Acknowledgment

This work was supported by NSF CNS grants 1617046, 1717588, and 1750109. We thank the anonymous reviewers and our shepherd, Sailesh Krishnamurthy, for their constructive feedback and suggested improvements.

References

[1] Cluster data collected from production clusters in Alibaba for cluster management research, 2018. https://github.com/alibaba/clusterdata.

[2] Amazon Elastic Compute Cloud (Amazon EC2), 2018. http://aws.amazon.com/ec2.

[3] Chen, S., Delimitrou, C., and Martínez, J. F. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA, 2019), ASPLOS '19, pp. 107–120.

[4] CloudLab. https://www.cloudlab.us. The University of Utah.

[5] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, IN, USA, 2010), SoCC '10, pp. 143–154.

[6] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China, 2017), SOSP '17, pp. 153–167.

[7] DCOPY (part of BLAS). http://www.netlib.org/blas.

[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, WA, USA, 2007), pp. 205–220.

[9] Delimitrou, C., and Kozyrakis, C. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, TX, USA, 2013), pp. 77–88.

[10] Delimitrou, C., and Kozyrakis, C. 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, UT, USA, 2014), ASPLOS '14, pp. 127–144.

[11] Delimitrou, C., Sanchez, D., and Kozyrakis, C. Tarcil: High Quality and Low Latency Scheduling in Large, Shared Clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing* (Kohala Coast, HI, USA, 2015), SOCC '15, pp. 97–110.

[12] Deng, L. The MNIST Database of Handwritten Digit Images for Machine Learning Research. *IEEE Signal Processing Magazine 29*, 6 (2012), 141–142.

[13] Ebrahimi, E., Lee, C. J., Mutlu, O., and Patt, Y. N. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, USA, 2010), pp. 335–346.

[14] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. Clearing the Clouds: a Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (London, UK, 2012), ASPLOS '12, pp. 37–48.

[15] Gandhi, A., Dube, P., Karve, A., Kochut, A., and Ellanti, H. The Unobservability Problem in Clouds. In *Proceedings of the 2015 IEEE International Conference on Cloud and Autonomic Computing* (Cambridge, MA, USA, 2015).

[16] Gandhi, A., Harchol-Balter, M., Raghunathan, R., and Kozuch, M. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *Transactions on Computer Systems 30* (2012).

[17] Gao, W., Zhan, J., Wang, L., Luo, C., Zheng, D., Ren, R., Zheng, C., Lu, G., Li, J., Cao, Z., Zhang, S., and Tang, H. BigDataBench: A Dwarf-based Big Data and AI Benchmark Suite. *CoRR abs/1802.08254* (2018).

[18] Google Cloud. https://cloud.google.com.

[19] Goder, A., Spiridonov, A., and Wang, Y. Bistro: Scheduling Data-parallel Jobs Against Live Production Systems. In *Proceedings of the 2015 Usenix Annual Technical Conference* (Santa Clara, CA, USA, 2015), USENIX ATC '15, pp. 459–471.

[20] Govindan, S., Liu, J., Kansal, A., and Sivasubramaniam, A. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal, 2011), pp. 1–14.

[21] Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N. M., Moore, A. W., Hand, S., and Crowcroft, J. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, USA, 2015), NSDI'15, pp. 1–14.

[22] Iorgulescu, C., Azimi, R., Kwon, Y., Elnikety, S., Syamala, M., Narasayya, V., Herodotou, H., Tomita, P., Chen, A., Zhang, J., and Wang, J. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *Proceedings of the 2018 USENIX Annual Technical Conference* (Boston, MA, USA, 2018), pp. 519–532.

[23] Janus, P., and Rzadca, K. SLO-aware colocation of data center tasks based on instantaneous processor requirements. In *Proceedings of the 8th ACM Symposium on Cloud Computing* (Santa Clara, CA, USA, 2017), pp. 256–268.

[24] Javadi, S. A., and Gandhi, A. DIAL: Reducing Tail Latencies for Cloud Applications via Dynamic Interference-aware Load Balancing. In *Proceedings of the 2017 IEEE International Conference on Autonomic Computing* (Columbus, OH, USA, 2017), pp. 135–144.

[25] Kasture, H., and Sanchez, D. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-critical Applications. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on* (2016), IEEE, pp. 1–10.

[26] Leverich, J., and Kozyrakis, C. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands, 2014).

[27] Li, J., Sharma, N. K., Ports, D. R. K., and Gribble, S. D. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA, 2014).

[28] Lin, H., Ma, X., Archuleta, J., Feng, W.-c., Gardner, M., and Zhang, Z. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (Chicago, IL, USA, 2010), pp. 95–106.

[29] Linux man page. virsh(1). https://linux.die.net/man/1/virsh.

[30] Liu, H. A Measurement Study of Server Utilization in Public Clouds. In *Proceedings of the 9th IEEE International Conference on Dependable, Autonomic and Secure Computing* (Sydney, Australia, 2011), pp. 435–442.

[31] Liu, Q., and Yu, Z. The Elasticity and Plasticity in Semi-Containerized Colocating Cloud Workload: A View from Alibaba Trace. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA, 2018), pp. 347–360.

[32] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, OR, USA, 2015), ISCA '15, pp. 450–462.

[33] Lu, C., Ye, K., Xu, G., Xu, C. Z., and Bai, T. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Proceedings of the 2017 IEEE International Conference on Big Data* (Boston, MA, USA, 2017), pp. 2884–2892.

[34] Maji, A., Mitra, S., and Bagchi, S. ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services. In *Proceedings of the 2015 IEEE International Conference on Autonomic Computing* (Grenoble, France, 2015), ICAC '15, pp. 91–100.

[35] Maji, A. K., Mitra, S., Zhou, B., Bagchi, S., and Verma, A. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference* (Bordeaux, France, 2014), Middleware '14, pp. 277–288.

[36] Marinos, A., and Briscoe, G. Community cloud computing. In *Proceedings of the 1st International Conference on Cloud Computing* (Beijing, China, 2009), CloudCom '09, pp. 472–484.

[37] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L. 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* (2011), ACM, pp. 248–259.

[38] Moreto, M., Cazorla, F. J., Ramirez, A., Sakellariou, R., and Valero, M. FlexDCP: A QoS Framework for CMP Architectures. *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 86–96.

[39] Nathan, S., Bellur, U., and Kulkarni, P. Towards a Comprehensive Performance Model of Virtual Machine Live Migration. In *Proceedings of the 6th ACM Symposium on Cloud Computing* (Kohala Coast, HI, USA, 2015), SOCC '15, pp. 288–301.

[40] Nguyen, K. T. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, 2016.

[41] Novaković, D., Vasić, N., Novaković, S., Kostić, D., and Bianchini, R. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (2013), pp. 219–230.

[42] Padhye, J., Firoiu, V., Towsley, D., and Kurose, J. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review 28*, 4 (1998), 303–314.

[43] Palit, T., Shen, Y., and Ferdman, M. Demystifying Cloud Benchmarking. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on* (2016), IEEE, pp. 122–132.

[44] Qureshi, M. K., and Patt, Y. N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Orlando, FL, USA, 2006), pp. 423–432.

[45] Schroeder, B., Wierman, A., and Harchol-Balter, M. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation* (San Jose, CA, USA, 2006), NSDI'06.

[46] Sharifi, A., Srikantaiah, S., Mishra, A. K., Kandemir, M., and Das, C. R. METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management. *SIGMETRICS Perform. Eval. Rev. 39*, 1 (2011), 13–24.

[47] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage*

Systems and Technologies (MSST) (Incline Village, NV, USA, 2010), MSST '10, pp. 1–10.

[48] Benchmark Suite for Apache Spark. https://github.com/CODAIT/spark-bench.

[49] Sphinx Speech Group, Carnegie Mellon University. CMU Robust Speech Recognition Group: Census Database. http://www.speech.cs.cmu.edu/databases/an4.

[50] Standard Performance Evaluation Corporation. SPECjbb2015. https://www.spec.org/jbb2015.

[51] Stress-ng. http://kernel.ubuntu.com/~cking/stress-ng.

[52] Traffic control howto. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html, 2006.

[53] Tesfatsion, S. K., Wadbro, E., and Tordsson, J. PerfGreen: Performance and Energy Aware Resource Provisioning for Heterogeneous Clouds. In *Proceedings of the 2018 IEEE International Conference on Autonomic Computing* (Trento, Italy, 2018), pp. 81–90.

[54] The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org.

[55] Tirumala, A., Qin, F., Dugan, J., Ferguson, J., and Gibbs, K. Iperf, 2006.

[56] TPC Council. TPC-C benchmark, revision 5.11. www.tpc.org/tpcc, 2010.

[57] Vanga, M., Gujarati, A., and Brandenburg, B. B. Tableau: A High-throughput and Predictable VM Scheduler for High-density Workloads. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal, 2018).

[58] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, CA, USA, 2013).

[59] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems* (Bordeaux, France, 2015).

[60] WAND Network Research Group. WITS: Waikato Internet Traffic Storage. http://www.wand.net.nz/wits/index.php.

[61] Wasserman, L. Models, statistical inference and learning. In *All of Statistics: A Concise Course in Statistical Inference*. Springer Publishing Company, 2010, ch. 6.

[62] Xu, C., Rajamani, K., Ferreira, A., Felter, W., Rubio, J., and Li, Y. dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service. In *Proceedings of the 13th EuroSys Conference* (Porto, Portugal, 2018), EuroSys '18, pp. 14:1–14:13.

[63] Yan, Y., Gao, Y., Chen, Y., Guo, Z., Chen, B., and Moscibroda, T. TR-Spark: Transient Computing for Big Data Analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA, 2016), pp. 484–496.

[64] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA, USA, 2010).

[65] Zhang, X., Dwarkadas, S., and Shen, K. Hardware Execution Throttling for Multi-core Resource Management. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (San Diego, CA, USA, 2009).

[66] Zhang, X., Tune, E., Hagmann, R., Jnagal, R., Gokhale, V., and Wilkes, J. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), pp. 379–391.

[67] Zhang, Y., Prekas, G., Fumarola, G. M., Fontoura, M., Goiri, Í., and Bianchini, R. 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* (2016), pp. 755–770.

[68] Zhang, Wei and Rajasekaran, Sundaresan and Duan, Shaohua and Wood, Timothy and Zhuy, Mingfa. Minimizing Interference and Maximizing Progress for Hadoop Virtual Machines. *SIGMETRICS Performance Evaluation Review 42*, 4 (2015), 62–71.

[69] Zhu, H., and Erez, M. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, GA, USA, 2016), pp. 33–47.