Analytical Approaches for Dynamic Scheduling in Cloud Environments

A Dissertation presented

by

Seyyedahmad Javadi

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2019

Abstract of the Dissertation

Analytical Approaches for Dynamic Scheduling in Cloud Environments

by

Seyyedahmad Javadi

Doctor of Philosophy

in

Computer Science

Stony Brook University

2019

Scheduling is a critical component for applications running on a cluster of nodes. Cloud resource capacity may vary dynamically due to resource contention that complicate scheduling decisions, in addition to the traditional challenge of varying workload demand and data popularity. To address these issues, scheduler design decisions must be augmented with analytical techniques to automatically adapt to varying workload and system conditions. In this dissertation, we consider generic scheduling problems that arise in today's cloud data center applications. In particular, we identify three popular scheduling scenarios and propose dynamic solutions for them.

First, many online application services are now provided by cloud-deployed Virtual Machine (VM) clusters. Given that cloud resource capacity can vary with time, request scheduling in cloud clusters from the tenant's perspective is a challenging and open problem. Second, resource under-utilization is common in cloud data centers. While running batch workloads in the background has been established as a common approach to improving resource utilization, an important challenge is considering the tenants' workloads as a black-box. Third, customer facing online services rely on scalable and low-latency data stores to maintain acceptable query tail latencies. An open scheduling challenge is the assignment of newly created data segments to worker nodes to prevent load imbalance among them.

It is our thesis that using analytical approaches to perform dynamic scheduling is critical to address the emerging performance challenges of cloud-deployed applications. In support of our thesis, we address the above outlined challenges. First, we present DIAL, an interference-aware load balancing framework that can directly be employed by cloud tenants without requiring any assistance from the provider. DIAL leverages ideas from queuing theory and machine learning to infer the colocated load and adjust the scheduling on individual VMs accordingly. Second, we present Scavenger, a batch workload scheduler that opportunistically runs containerized batch jobs next to tenants' workloads to improve utilization without requiring information about the tenant workload. Third, we present an efficient segment assignment strategy, EASY, that leverages analytical modeling to predict the future load induced by data segments, thus allowing for long-term balancing of load across worker nodes.

To my lovely parents, family and friends who encouraged me wholeheartedly.

Contents

List of Figures

Acknowledgements

Many people have helped and supported me throughout my PhD to accomplish the research goals and to complete this dissertation. Firstly, I would like to thank my advisor, Anshul Gandhi, who has genuinely supported me, provided the required resources, and contributed to this work. He taught me several lessons including and not limited to realizing research goals systematically, writing technical papers, and giving technical presentations. I am also thankful for my dissertation committee members, Erez Zadok, Michalis Polychronakis, and Sameh Elnikety for their insightful comments. A big thank you to I.V. Ramakrishnan, C.R. Ramakrishnan, Cynthia SantaCruz-Scalzo, Betty Knittweis, and Kathy Germana for their invaluable administrative support during my stay at Stony Brook University.

I would also like to thank the current and past students of PACELab who contributed to this work — Amoghavarsha Suresh, Muhammad Wajahat, Shubham Jindal, Shireen Nagdive, Shalini Bhaskara, Rahul Doshi, Prashanth Soundarapandian, Harsh Gupta, Robin Manhas, Shweta Sahu, Piyush Shyam Banginwar, Vaishali Chanana, Rashmi Narvekar, Mitesh Kumar Savita, Himanshu Rajput, Sagar Mehra, and Bharath Reddy.

In the last five years, I have been very fortunate to have had great friends, Reza Baseda, Moussa Ehsan, Javad Nejati, Andrew Guthrie, Ibrahim Hammoud, Mahdi Javanmard, Ali Mirmoghtadaei, Masoud Rohizedeh, Mohsez Zakeri, Hamid Reza Asadi, Amirmasoud Almotahari, Hamed Ghavamnia, and Abouzar Samiei, just to name a few. Thanks for all your support and help, and for making my PhD so much fun.

Finally, I would like to thank my parents, my sister and brothers, and my relatives for their endless love and support, and for trusting my decision to pursue a PhD. It was your support and prayers that helped me to stay strong and work hard all these years far from home.

Chapter 1

Introduction

With the emergence of the cloud computing paradigm, many online services and applications are now provided by cloud deployed resources. The cloud offers virtually unlimited and economical compute and storage resources to end-users. These resources can be leased on an hourly basis, and are provided by several cloud service providers, such as Amazon, Google, Microsoft, and IBM.

Scheduling is a critical component for applications running on a cluster of nodes. This is because scheduling decisions play a crucial role in determining the end-to-end application tail latency. In emerging cloud environments, scheduling decisions are complicated by the fact that underlying resource capacity may vary dynamically due to resource contention, in addition to the traditional challenge of varying workload demand and data popularity.

The primary cause of varying resource capacity in cloud environments is multi-tenancy, which is a fundamental design principle of cloud computing. Under multi-tenancy, a Physical Machine (PM) is shared among multiple cloud tenant Virtual Machines (VMs)/containers and potential provider workloads. When the aggregate resource demand on the PM is high, we see resource contention and performance interference between the hosted applications.

Interference is a common occurrence in data centers [168, 27], and is prevalent in both public and private cloud deployments. Several studies [85, 42, 145, 161, 99, 94] have shown that tail response times for applications under interference are significantly higher, to the tune of 3-10$\times$, on AWS cloud instances. Further, the studies reveal several instances of interference that last for 30s or longer. Similar observations have also been made for other

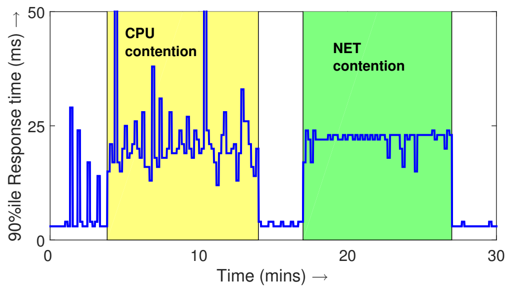


Figure 1.1: Effect of interference on application performance. During the CPU contention (yellow shaded region) and network contention (green shaded region) intervals, 90%ile response time increases by about 4×.

public cloud providers, including Azure [154], Google and Rackspace [74]. Likewise, interference is known to impact application performance on private clouds [137, 109, 97, 70]. Interference can be caused by the contention of any physical resource among co-located VMs, including CPU, network, disk, memory, and last-level-cache (LLC). Furthermore, interference can be caused by contention for *several resources simultaneously* [63] and is *dynamic* due to resource demand variations in tenant and potential provider workloads [45, 169]. Our own experiments on OpenStack for a popular multi-tier web benchmark, CloudSuite [39], highlight the impact of CPU and network interference on tail response times, as shown in Figure 1.1. We discuss interference in detail later in Section 2.3.

In the context of resource contention and performance interference, we specifically consider three significant scheduling challenges:

1. Cloud tenant VM's variable resource capacity: Cloud tenant VM resource capacity can be impacted by the colocated VMs because of interference. Prior work on interference mitigation has typically focused on provider-centric solutions. A popular approach is to profile applications and co-schedule VMs that do not contend on the same resource(s) [10, 15, 138, 97, 30]. However, since interference is dynamic and can emerge unpredictably, statically co-scheduling VMs will not suffice. VM migration can help in this case, but interference is volatile and short-lived, often lasting for only a couple minutes [85]; by contrast, migration can take several minutes [96] and can incur overheads [32], especially for stateful applications [31].

A critical challenge that has not been addressed with regards to interference is the *lack of visibility and control between the provider and the tenant*, especially in public clouds [42]. Specifically, tenant VMs in the public cloud can not, or should not, be profiled a priori by the provider due to privacy concerns [99]. Further, providers are not always aware of the cloud user's Service Level Objective (SLO) requirements or the user application's bottleneck resources.

2. Background workload impact on cloud tenant VM performance: Servers in cloud data centers often experience low resource utilization [31, 158]. A study focused on Amazon EC2 observed that cloud server usage is often below 10% [81]. To increase server utilization, prior works have proposed running provider's batch workloads, such as Hadoop or Spark jobs, next to users' VMs opportunistically to leverage idle resources [170, 82, 50]. While effective, the key challenge with this approach is *interference*, due to which the colocated tenant VMs' performance can degrade significantly.

   Existing solutions often either (i) rely on historical usage patterns to predict the resource demand of foreground VMs [169, 22], or (ii) benchmark tenants VM performance to carefully colocate background workloads [30, 31], or (iii) regulate the resource usage of background workloads to avoid SLO violations for the foreground VMs [82, 60]. Such solutions are ineffective and, at times, infeasible in cloud environments as tenants do not expect their VMs to be instrumented [99], and are not required to share their performance SLO requirements with the provider [42]. Even if foreground VMs can be profiled for a short time, there is often significant variation in tenants workloads to be accurately captured by a finite profiling run [63]. There is thus a need for black-box background workload scheduling to simultaneously improve resource utilization and avoid SLO violations for the tenant workloads.

3. Impact of hotspots and load imbalance on application tail latency: cloud-deployed applications' performance challenges are not limited to the unpredictable performance of acquired cloud resources by the tenants. Scheduling over a cluster of worker nodes to balance load while respecting data locality is a classic scheduling challenge. To put this challenge in context, we study Online Analytical Processing (OLAP) systems and show how critical this challenge is.

In the big-data paradigm, OLAP systems typically split a big table into several data segments and distribute these data segments among a cluster of worker nodes. To serve a query, every worker node runs the query on its' assigned data segments, and then these local results are integrated to compute the final response. In such systems, Segment Assignment Strategy (SAS), which dictates which worker nodes host a newly generated segment, plays a crucial role in preventing hotspots that severely impact query tail latencies. The difference in popularity of different data segment and the specific mix of queries makes SAS a challenging problem. Note that hotspot refers to having some highly-loaded worker nodes while the other worker nodes are under-utilized. In this case, intra-application resource contention is the underlying reason for high tail latencies [65].

In this dissertation, we look at the above three challenges from the scheduling perspective. Regarding the first challenge, we have a request scheduling problem with the goal of having minimum tail latency for applications running on top a cluster of VMs facing unpredictable performance. Regarding the second challenge, batch workloads need to be scheduled next to the black-box foreground workloads (tenants' VMs) with two competing goals: (1) foreground workloads' SLO are not violated, and (2) background workloads' progress rate is maximized; the progress rate is directly correlated with resource utilization. Regarding the third challenge, we have a data segment scheduling problem with the goal of having a balanced load among the worker nodes to prevent hotspots and high tail latency.

It is our thesis that using analytical approaches to perform dynamic scheduling is critical to address the above three challenges. Accordingly, this dissertation makes three contributions:

1. Dynamic Interference-Aware Load balancing (DIAL) [63]: DIAL is a dynamic user-centric scheduler for mitigating interference in load-balanced cloud deployments. We consider a generic cloud-deployed application that has a tier of worker nodes hosted on multiple VMs and experiencing unpredictable interference from colocated VMs (owned by other cloud tenants). The incoming load is distributed among the worker nodes via one or more load balancers. This generic model is widely applicable, for example, for web applications (where workers are web or application servers), online analytical processing (OLAP)

systems like Pinot [66], etc. The key idea behind DIAL is to *infer* contention in colocated VMs and *adapt* the load scheduling of incoming requests among user VMs. We introduce a model for interference, based on queueing theory [56, 72], to understand the impact on the performance of contention at shared physical resources. DIAL then optimizes the time-varying load distribution among worker VMs to reduce tail latency.

We implement and experimentally evaluate DIAL for two specific application classes:

(a) Web applications: We implement DIAL on HAProxy [55], and evaluate DIAL's benefits using two popular web applications with varying workload under CPU, network, disk, and cache interference on OpenStack and AWS clouds. Our experimental results show that DIAL reduces 90%ile response times by as much as 70% compared to interference-oblivious load balancers. Further, compared to existing interference-aware solutions, DIAL reduces tail response times by as much as 48%.

(b) OLAP Systems: We implement DIAL for a popular and open-source OLAP system called Pinot that has been used in production clusters at LinkedIn and Uber. Our experimental results on a KVM cluster show that DIAL can reduce 95%ile query completion times by 16-40% under CPU and LLC contention.

2. Provider-centric resource manager for background workloads (Scavenger) [64]: Scavenger dynamically regulates the resource usage of background jobs to complement the resource demand of black-box foreground workloads. We consider a cloud environment with customer VMs as the foreground workload and Spark (within the YARN framework [139]) as the background workload running on containers. Scavenger is a reactive workload scheduler, and complements proactive schedulers such as Borg [141]. 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's scheduling application-agnostic in practice and easy to deploy.

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 [39] and TailBench [67], 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%. Scavenger consistently increases server memory and CPU usage by more than 100%.

3. Efficient segment Assignment StrategY (EASY) [65]: EASY is our proposed load-aware data segment assignment solution for Online Analytical Processing (OLAP) systems. EASY addresses the traditional challenge of varying workload demand and data popularity for Pinot which is a popular distributed near-realtime OLAP solution extensively used at LinkedIn and Uber for serving user queries and for internal analysis.

   We implement EASY on top of Pinot and experimentally evaluate EASY using a realistic Pinot benchmark. Our results show that EASY significantly improves the load balance among worker nodes, reducing query tail latencies by up to 6–21% when compared to the default SAS of Pinot and Druid. Importantly, EASY requires few changes and creates negligible overhead.

The rest of this dissertation is organized as follows. Chapter 2 discusses the cloud computing related concepts that we use repeatedly in this dissertation. Chapter 3 discusses related work and motivates our work further by explaining the design challenges that have not yet been fully addressed. Our three contributions, DIAL, Scavenger, and EASY are illustrated by Chapter 4, 5, and 6, respectively. Finally we conclude with a summary of this dissertation in Chapter 7.

Chapter 2

Background

This dissertation deals with cloud computing resources and workloads, and related concepts. We thus start with an overview of these concepts in this chapter. The chapter also motivates our work by analyzing production resource usage traces.

We first briefly discuss cloud computing in Section 2.1. In particular, we discuss virtualization technology in Section 2.1.1 and private versus public clouds in Section 2.1.2. We then describe cloud workloads used in our experiments in Section 2.2. The workloads are categorized into latency-sensitive (Section 2.2.1) and batch applications (Section 2.2.2). The main challenge we encounter in cloud computing performance management is performance interference; thus we discuss performance interference in Section 2.3. Accordingly, Section 2.3.1 illustrates interference with more details, Section 2.3.2 discusses reasons for interference, and Section 2.3.3 illustrates prevalence of interference in public and private clouds. To motivate the need for better scheduling approaches, we provide an analysis of resource usage in Section 2.4 illustrating the current state of cloud data centers' resource utilization. The analysis covers three traces, namely Google traces (Section 2.4.1), Azure traces (Section 2.4.2) and Alibaba traces (Section 2.4.3).

2.1 Cloud Computing

Cloud computing is often referred to as the practice of delivering computing services (e.g., servers and storage) over the Internet. Cloud computing has shifted the traditional ways of computing drastically and gained tens of billions of dollars of market by offering services from bare metal servers to serverless computing. Many online services and applications, such as Netflix [16] and Expedia [37], are now provided by cloud-deployed Virtual Machines (VMs). The main benefits of cloud computing include, but not limited to, low cost, elasticity, and the ability to pay-as-you-go.

Virtualization is often referred to as the underlying technology that empowers cloud computing to share physical resources among multiple tenants; thus Section 2.1.1 briefly overviews the core concepts of virtualization. Besides, cloud services can be deployed in four different ways, namely private cloud, community cloud, public cloud, and hybrid cloud [89] from which we focus on private and public clouds, discussed briefly in Section 2.1.2.

2.1.1 Virtualization Technology

Virtualization technology allows cloud providers to run users' applications in isolated environments, with high flexibility from computing and storage perspectives, by running multiple VMs or containers on the same Physical Machines (PMs). The software layer which is responsible for managing resources and sharing them among VMs is called the hypervisor. There are two types of hypervisors, as shown in Figure 2.1: (1) type-1, bare metal hypervisors, that run directly on top of hardware, and (2) type-2, hosted hypervisors, that operate as complex applications on top of the existing Operating Systems (OSs). In both types, hypervisors act as the Virtual Machine Manager (VMM) and allocate resources to VMs as well as monitor and control their usage.

Figure 2.1 also shows a simplified picture of current commodity servers. Current processors consist of several CPU cores, each of which has its L1 and L2 caches, while they share Last Level Cache (LLC) or L3 cache [8]. Having many cores enables a high degree of parallelism where different processes/threads can be scheduled to be run at the same time. Hardware vendors also provide special support for virtualization technology such as hyperthreading which increases the degree of parallelism by launching two

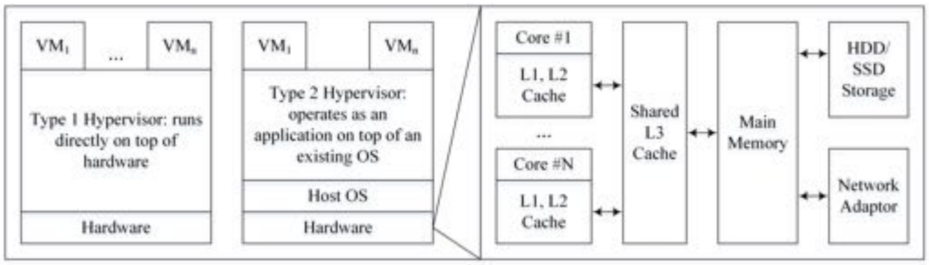


Figure 2.1: Types of hypervisors and a simplified view of a multicore system.

hardware/hyper-threads per CPU core.

Due to the enormous benefits of virtualization and the significant attention gained by it, several platform virtualization software (hypervisors) have been developed such as KVM, Xen, VMWare ESX and Hyper-V, to name a few. To conduct our experiments, we use the Kernel-based Virtual Machine (KVM), which is a virtualization infrastructure for the Linux kernel. KVM is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V), and thus users can run multiple VMs running unmodified Linux or Windows images. KVM converts Linux into a type-1 (bare-metal) hypervisor and since it is is part of the Linux kernel, it has all the required operating system-level components to run VMs such as a memory manager, process scheduler, input/output (I/O) stack, device drivers, security manager, a network stack, just to name a few. Every VM is implemented as a regular Linux process, scheduled by the standard Linux scheduler, with dedicated virtual hardware like a network card, graphics adapter, CPU(s), memory, and disks.

Cloud deployment is a challenging task that needs software technologies beyond hypervisors. Accordingly, to provide all the required technologies from networking management to a dashboard, several enterprise and open-source cloud frameworks have been developed among which OpenStack [101] is very popular and has been used in production systems. OpenStack is an open-source cloud framework that embraces a modular architecture to provide a set of core services that facilitates scalability and elasticity as core design principles. The main components of OpenStack are as follows:

- Compute: This service manages virtual machine instances using different hypervisors the default of which is KVM for Linux.

- Networking: This service provides various networking services such as IP address management, DNS, DHCP, load balancing and security groups.

- Image service: This service provides disk-image management services, including image discovery, registration, and delivery services to the Compute service, as needed.

- Dashboard: This is a web-based interface for both cloud administrators and cloud tenants using which they can provision, manage and monitor cloud resources.

In this dissertation, we use KVM and OpenStack to perform our experiments.

2.1.2 Private versus Public Clouds

Private cloud is defined by NIS [89] as "*cloud infrastructure provisioned for exclusive use by a single organization comprising multiple consumers, such as business units. The cloud may be owned, managed, and operated by the organization, a third-party, or some combination of them, and it may exist on or off premises*". For instance, companies can use their infrastructure for cloud services by deploying virtualization frameworks such as OpenStack [101]. These infrastructure are typically behind an internal firewall, thus they offer an increased level of security. However, the companies will still be responsible for the management, maintenance, and updating of the infrastructure.

Public cloud is defined by NIS [89] as "*cloud infrastructure provisioned for open use by the general public. The cloud may be owned, managed, and operated by a business, academic, or government origination, or some combination of them and the cloud exists on the premises of the cloud provider*". Cloud users (e.g., companies) can then use several types of services (e.g., launching VMs) offered by the providers. In other words, cloud users share underlying physical infrastructure and get benefits such as lower cost, elasticity and geo-distributed computing and storage resources without being concerned about underlying data centers' management and maintenance.

2.2 Cloud Workloads

Different types of workloads can be deployed on the cloud. Several efforts have been made to create representative benchmarks to be used by academia and industry. In this dissertation, we use multiple benchmarks categorized into latency-sensitive workloads and batch workloads, discussed briefly in the following two sections.

2.2.1 Latency-Sensitive Workloads

Latency-critical workloads generally refer to applications or their components that have strict end-to-end response time or execution time restrictions. These applications are widespread in data centers [67] and form a fabric of interactive, large-scale (scale-out) online services. We use following latency sensitive workloads in our experiments:

- CloudSuite [39]: The first version of this benchmark suite was released in 2012 and has constantly been updated over time [106]. CloudSuite 3.0 provides eight workloads, five of which can be categorized as latency-sensitive workloads:

    1. Web Serving: It is clear that web serving, such as social networking services, is a very popular workload. Accordingly, this benchmark is a multi-tier, multi-request class, PHP-MySQL based social networking application. The benchmark has four tiers: (1) the web server, (2) the database server, (3) the Memcached server, and (4) the clients.
    2. Web Search: Considering the massive amount of daily generated web contents, search engines play a crucial role to find related information. Web Search benchmark deploys Apache Solar search engine framework to respond to simulated real-world clients' request to the index nodes containing an index of the text and fields found in a set of crawled websites.
    3. Data Serving: Many online services often deal with massive amount of data that can be queried by deploying backend data stores. Data Serving benchmark relies on the Yahoo! Cloud Serving Benchmark (YCSB), which is a framework to benchmark data

store systems. Data serving benchmark uses YCSB to generate loads for Cassandra data store.

4. Data Caching: Many caching systems have been developed to speed up database-driven applications by caching data and objects (e.g., queries' responses) in main memory to reduce the number of times an external data source must be read (e.g., queries being re-executed). Data Caching benchmark uses the Memcached data caching server and twitter dataset to simulate the behavior of Twitter.
5. Media Streaming: Nowadays streaming services such as video streaming are very popular. Media Streaming benchmark uses Nginx web server as a streaming server for hosted videos of various lengths and qualities. The client, based on httperf's wsesslog session generator [92], generates a request mix for different videos, to stress the server.

- TailBench [67]: 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 [67]):

    1. Xapian: an online search benchmark using the Wikipedia dataset as search index.
    2. Moses: a statistical machine translation application using the opensubtitles.org English-Spanish corpus.
    3. Silo: an in-memory database application driven using TPC-C [134].
    4. Specjbb: an industry-standard Java middleware benchmark [122].
    5. Masstree: a key-value store application driven using YCSB [20].
    6. Shore: an on-disk database driven using TPC-C [134].
    7. Sphinx: a speech recognition system driven using the AN4 audio dataset [121].
    8. Img-dnn: a handwriting recognition application driven using the MNIST images database [33].

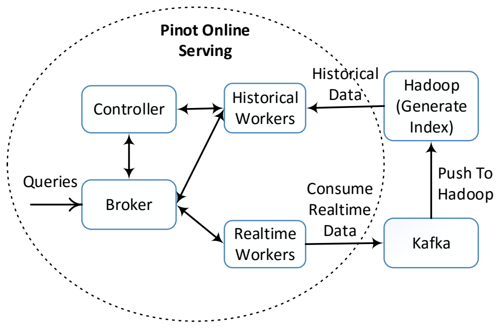


Figure 2.2: Overview of Pinot's architecture.

- WikiBench [135]: WikiBench is a distributed web application benchmarking tool based on Wikipedia. The benchmark uses real traces to generate realistic workloads to target systems hosting Wikipedia data and software.

- Pinot [66]: Pinot is a distributed near-realtime OLAP (On-Line Analytical Processing) data store that is used at LinkedIn for various user-facing functions and internal analysis. Pinot has been open-source since 2015 and is currently being used by other companies such as Uber. Pinot is designed to be able to process 100 Million SQL-like queries for 100 Billions of records in 10s of ms latency.

  Figure 2.2 illustrates the Pinot architecture including the three main components: (1) controller, (2) broker, and (3) worker nodes. The controller is responsible for cluster-wide coordination and segment assignment to worker nodes (SAS). The broker (or brokers) receives queries from clients, distributes them among workers, and integrates the results from the workers and sends the final result back to clients; the end-to-end query response time can be obtained at the broker. The worker nodes host data segments and respond to query sub-requests that originate from the broker. We have implemented an improved Pinot benchmark that is described in Section 6.5.1.

In this dissertation, we have three contributions, namely, DIAL, Scavenger, and EASY. We use CloudSuite, WikiBench, and Pinot for evaluating DIAL (Sections 4.5 and 4.6), CloudSuite and TailBench to evaluate Scavenger (Section 5.5), and Pinot to evaluate EASY (Section 6.5).

2.2.2 Batch Workloads

Batch processing refers to non-interactive computation such as one-off jobs (e.g., compilation) and processing of multiple items in batches. With emergence of the big-data paradigm, batch workloads also refer to processing of massive amount of data in a distributed fashion. Several big-data frameworks can run batch workloads among which the following ones are prevalent and often used in research and production systems:

- Hadoop [128]: Apache Hadoop is a collection of open-source software libraries that enable the distributed processing of massive amount of data across clusters of nodes using simple programming models. The well-known programming model used by Hadoop is MapReduce in which a traditional single-node computation is instead done by several nodes in parallel on distributed splits of input data during map and reduce phases, thus speeds up the computation significantly. Hadoop Distributed File System (HDFS) is storage core of Hadoop and enables splitting large files into configurable size blocks and distributing them across nodes in a cluster. Being able to scale up from single servers to thousands of machines and detecting and handling failures at the application layer are two main features of Hadoop architecture. In addition to Hadoop Common, HDFS and Hadoop MapReduce modules, Hadoop Yarn [139] is a very popular and highly deployed resource management framework that is mainly responsible for managing cluster resources and scheduling user's applications.

- Spark [129]: Apache Spark is an open-source distributed engine for large-scale data processing. Spark is known to be significantly faster than Hadoop MapReduce paradigm as it employs in-memory processing [162]. For in-memory processing, Spark introduces Resilient Distributed Dataset (RDD) which is a read-only multiset of data items distributed over the main memory of a cluster of nodes and maintained in a fault-tolerant manner [162]. A full Spark deployment requires a cluster manager and a Distributed File System (DFS). Accordingly, Spark and Hadoop can have main components in common, such as using Hadoop Yarn and HDFS as Spark cluster manager and Spark DFS, respectively.

- TensorFlow [126]: This is a high-performance numerical computation framework released in 2017 by Google Brain. TensorFlow has a flexible architecture that enables easy deployment of computation across a variety of platforms (e.g., CPUs and GPUs) and range of computing resources from desktops to clusters of servers. In addition to computer research and industry, many other scientific domains use TensorFlow due to its strong support for machine learning and deep learning.

In this dissertation, we use spark-based benchmarks, including Spark-Bench [120] and BigDataBench [7], in our experiments for Scavenger.

2.3 Performance Interference in Cloud

Performance Interference is one of the main challenges of cloud computing; thus this section provides an overview of performance interference, its underlying reasons and prevalence in cloud environments.

2.3.1 Performance Interference

Performance interference occurs when co-located VMs/containers on a Physical Machine (PM) contend for resources. To properly define performance interference, let us consider a high-level comparison between a non-virtualized environment and a virtualized environment. Suppose two applications (A_1 and A_2) run on two separate non-virtualized servers. Both the servers have distinct resources such as L3 cache, CPU cores, disk, and network bandwidth. Therefore, both applications often have predictable performance since they have stable resource allocation over time. Now suppose two VMs (VM_1 and VM_2) are co-located on the same PM and share the PM resources. Virtual machines VM_1 and VM_2 run applications A_1 and A_2, respectively. In this case, we define performance interference as follows:

Performance interference is the situation where co-located VMs' requests contend for the shared resources; thus the real capacities of resources assigned to the VMs are less than what they should be. In other words, when VMs do not get their requested resource capacities (based on their specifications) because of sharing of underlying PM resources, we call it performance interference

The immediate impact of performance interference is unpredictable performance which is very important and challenging. Performance variation has been observed across resources (CPU capacity, network latency, I/O bandwidth,etc.) and across CSPs and data center locations [49, 38, 147, 41], and is among the top concerns that dissuade customers from cloud adoption [133, 132, 107]. For example, application developers at Facebook, Google, and Bing focus on maintaining low tail latencies [3, 118] to avoid significant revenue losses due to user abandonment [142]. The severe performance variation in cloud deployments (ranging from 2-27×), highlighted by existing studies [156, 90, 146], coupled with the (unsurprising) fact that CSPs do not offer performance guarantees [43], prevents these businesses from moving their latency-sensitive applications to the cloud.

Back to our example, considering the above definition, VM_1 and VM_2 might not deliver stable performance over time which is in contrast to the predictable performance of non-virtualized environments. Let's assume that both applications A_1 and A_2 are CPU-intensive. If the underlying physical CPU cores shared by VM_1 and VM_2 are not enough to handle the sum of peak load of A_1 and A_2, then VM_1 and VM_2 will contend for the underlying CPU cores and face interference, resulting in performance loss. In other words, VM_1 and VM_2 only get a fraction of the computing resources they are entitled to get. These situations illustrate performance interference.

Figure 2.1 also emphasizes that co-located VMs shares PM resources (e.g., LLC, CPU cores, disk, and network bandwidth). Sharing resources between the co-located VMs without providing performance isolation can lead to unpredictable performance for the VMs [147, 49]. This co-location and lack of performance isolation, which are non-trivial challenges, especially in the case of LLC, is the critical reason for interference. The next section will explain this fact with more details.

2.3.2 Reasons for Interference

The previous section presented the problem of performance interference caused by resource contention in virtualized environments. In general, any shared resource under contention can result in interference. The following is a list of the primary resources in a typical commodity server that can be a source of interference:

- **Last Level Cache (LLC)**: LLC or L3 cache is low latency and expensive memory that is used to save the data fetched from main memory to do subsequent read/write operations faster. Co-located VMs share LLC of their host because of which their memory access pattern will affect each others' performance. For instance, if some of the co-located VMs access random sections of their memory space (such that the accessed space is bigger than LLC size) very frequently, it will pollute LLC lines and lead to higher LLC miss rate for the co-located VMs. Higher LLC miss rate leads to increased memory operations. Since latency of a memory operation (LLC miss) is very high compared to LLC hit, performance of co-located VMs drops considerably [148].

- **Main memory bandwidth**: VMs contend on other components of main memory including channels, ranks, and banks. Often these sources of contention are categorized as memory bandwidth interference [148].
- **CPU**: Virtual CPU (VCPU) is the unit of assigning CPU cores to VMs. In general, the number of VCPUs that are assigned to co-located VMs can be more than the number of physical cores or hardware threads in the host. This approach helps to increase host CPU utilization since previous studies show that in cloud environments considerable percentage of VMs' VCPUs are idle [44]. However, if more than the existing CPU cores or hardware threads in a Physical Machine (PM) is assigned to its hosted VMs, and the VMs execute computationally intensive operations, they will contend for CPU cycles.
- **Network I/O bandwidth**: Because network communications are a fundamental part of today's distributed systems and applications, network stack of a PM is a critical shared resource among co-located VMs in the PM. If multiple VMs execute several network I/O operations (close to the peak capacity) at the same time, they will face high queuing delay or even packet drop because of high pressure on the shared network stack. For instance, if there is a 1Gb link between a PM and rest of the infrastructure, and two different tenants' VMs are co-located in the PM, interference will happen if the VMs push network traffic more than the peak bandwidth.
- **Main memory capacity**: In general, total memory capacity assigned to the co-located VMs in a PM can be more than available physical memory in the PM [57]. Memory overprovisioning is a well-known approach to launch more VMs and have higher physical resource utilization. However, if VMs request total memory capacity that is more than available capacity, the host OS will have to deal with insufficient

physical memory capacity. In fact, in this situation and considering virtual memory capacity, there will be a high rate of disk I/O in host OS, and this can lead to huge performance loss.

- **Disk I/O bandwidth**: Attached persistent storage (e.g., HDD and SSD) is an important shared resource used by many applications. The co-located VMs' disk IO requests contend on disk IO stack for service. In other words, multiple VM requests will have to be multiplexed on one disk, which will result in I/O bandwidth contention. For example, suppose an HDD disk with a known sequential write speed. If only one VM (VM_1) is writing to the disk sequentially, it can write up to that speed. However, if a co-located VM (VM_2) starts to issue random reads/writes to the same disk since the disk head will move randomly, the sequential write speed for VM_1 will drop considerably.

Cloud vendors are aware of the performance interference that is caused by resource contention. While different cloud vendors have deployed their approaches to ensure predictable performance, the solutions are far from perfect. For instance, reservation-based methods in which some shares of a PM resource are reserved for each of the hosted VMs can lead to low utilization and inefficient resource usage. The reason is that there can be a situation where some of the VMs do not use their share while others need more resources. Furthermore, it is hard to share some of the resources such as LLC while guaranteeing that VM requests do not interfere with each other.

Popular public cloud offerings, such as Amazon AWS [4], have taken steps to limit performance interference. In particular, AWS limits the amount of resources, such as network or CPU cores, that are assigned to customer VMs. Further, AWS does not typically oversubscribe resources on their hosts [11]. Thus we do not see interference for resources such as CPU and memory. However, interference does exist even in AWS when LLC and disk I/O bandwidth is under contention, as we observed in our experiments on AWS. Thus, in the absence of perfect isolation mechanism, if multiple VMs start using a shared resource at the same time such that the available resource capacity is not enough to meet all VM demands, the performance of the VMs will drop.

2.3.3 Prevalence of Interference

Several studies have highlighted the prevalence of interference and its negative impact on performance in cloud computing environments. The studies typically consider two main categories of testbeds: (1) a cluster of VMs launched on popular public clouds where there is limited visibility of the underlying physical infrastructure since they are controlled by cloud providers and, (2) a cluster of physical servers that runs popular cloud management software (e.g., OpenStack) and is under the full control of the researchers because of which we call them private cloud environments. Accordingly, we summarize and explain these research results based on their deployed experimental setups in the following two subsections.

Public Cloud Environments

In this section, we summarize the results of the main studies that consider the impact of sharing resources in public cloud on VMs' performance. Typically, several VMs are launched and their performance is evaluated by running different benchmarks under varying circumstances.

Wang *et al.* [147] show that virtualization and sharing of resources in Amazon EC2 affects network performance of applications. In particular, they show that small EC2 instances often share processors and they get 40% to 50% of the physical CPU sharing. Accordingly, they conclude that observed periodic low TCP throughout for the small instances is because of the processor sharing. Besides, they show that round-trip-times variations are much more higher for EC2 instances (especially small ones) in comparison to non-virtualized machines.

Ghoshal *et al.* [49] also observed an occasional drop in performance of EC2 instances when they run a benchmark for a long time (same observation in [100]). They guess that this can be because of sharing underlying resources. Furthermore, they show that if we run MPI (message passing interface) tasks on multiple EC2 instances while they are sharing an EBS volume to perform I/O, there will be high resource contention over a limited network. This can negatively affect the overall performance. Although in this scenario, VMs of the same tenant share the EBS volume, it still illustrates the importance of performance isolation. The authors also observed considerable improvement in I/O performance after the peak processing hours since then there is limited contention for the shared resources.

Maji *et al.* [85] run an identical workload on both private and Amazon EC2 clusters for 100 hours and compare the response time distribution to show that interference is a real problem. According to their results, EC2 cluster response time distribution has much longer tail latencies which indicated the periods of unpredictable performance. Several of these interference periods were 30 seconds long and the longest one was 140 seconds. Furthermore, using the method described in [115] and after some trial and error, they could co-locate two EC2 instances. Then, in one of the VMs, they run a workload that accesses memory very frequently to create LLC contention. They observed the order of 4$\times$ increase in response times of the application running in the second VM [115]. This observation shows that LLC contention can happen for Amazon EC2 instances launched by several independent tenants. Gandhi *et al.* [42] also show that Amazon EC2 instances exhibit significant variation in performance, probably due to resource contention from (unobservable) co-located VMs.

Our Experiments: Amazon AWS recently started providing a new option called EC2 dedicated host [6]. Users can leverage dedicated hosts to address compliance requirements and reduce costs by using existing server-bound software licenses. Interestingly, using dedicated hosts, users can launch their EC2 instances on them. There is an upper limit on how many instances a user can launch into a dedicated host. Placement of EC2 instances on the dedicated hosts can be done either by the users themselves or automatically by AWS. One interesting feature of this new option is that researchers can co-locate EC2 instances easily (without using the complex method described in [115]) to study the impact of interference. We use this feature to create LLC and disk I/O interference easily, but could not create CPU, memory, and network contention [63]. Therefore, LLC and disk I/O interferences are still real problems in Amazon AWS.

Private Cloud Environments

We now discuss prevalence of interference in private cloud environments. Koh *et al.* [71] did one of the earliest analysis of performance interference in vitualized environments in 2007. They select several realistic workloads and run different pairs of them in two VMs on top of the Xen hypervisor in order to study how different applications affect each others' performance. Their results confirm that there can be a high degree of interference when different workloads are running in co-hosted VMs. For instance, they show that performance of a cache-intensive application will drop significantly when another cache-intensive workload is running in a co-located VM. They obtain similar results for CPU and I/O intensive applications. Bu *et al.* in [10] show that running CPU-intensive (or I/O intensive) workloads in the co-located VMs increases the runtime of the target CPU (or I/O) bound benchmark by up to 7× of the non-interference runtime. On the other hand, they show that applications that are sensitive to *different* resources face less performance degradation when they run on two co-located VMs. For example, a CPU-intensive application experiences less severe performance drop (less than 1.5×) when an I/O intensive workload is running in the co-located VM. Similar observations where made by Pu *et al.* [111, 110] where the authors suggest running I/O intensive workloads with co-located CPU-intensive workloads to limit performance interference.

Nathuji *et al.* [97] run a micro-benchmark in a VM co-located with a second VM to show the impact of LLC interference. The micro-benchmark consists of several iterations over a specified working set size until a constant amount of data (which is much more larger than LLC size) is accessed. The micro-benchmark is run for several increasing working set sizes and execution times are reported. When they run the micro-benchmark alone in the VM while the second VM is idle, the execution times increase when the working set size becomes larger than LLC size. The reason is that because of the higher LLC miss rate for larger working sets; more memory operations are needed. Then, if the second VM runs few synthetic memory intensive threads, the execution times of the micro-benchmark increase even when the working set sizes are considerably less than LLC size. This clearly illustrates that the second VM uses some of the LLC capacity and the VMs requests interfere. They show that running the workload in the second VM can increase the execution tomes of the micro-benchmark by 380% because of LLC sharing.

While popular hypervisors such as Xen have been progressing in scheduling mechanisms to provide a fair share of the resources (or flexible resource sharing approaches) to the VMs, Zhang *et al.* [165] show that it is still possible to have considerable interference. They run TPC-W benchmark as the target (foreground) application and Hadoop as the background workload in a co-located VM to create contention. Even if they tune Xen scheduler's parameters in order to give Hadoop the lowest possible weight, there is a significant difference in the distribution of TPC-W response times when it is run alone (e.g., 90th percentile of TPC-W response times is 100 msec) compared to when it is run with the co-located VM running Hadoop and sharing the same CPU core (e.g., 90th percentile of TPC-W response times is 400 msec). Mukherjee *et al.* [93] also confirms the same observation when Apache instances share a non-virtualized server versus the scenario that they are run in different VMs sharing the underlying server. The results show almost 20× increase in the mean response time of Apache instances running on the VMs when they are highly loaded.

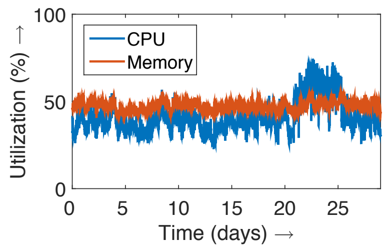
Our Experiment: We also conducted several experiments in our private OpenStack cloud testbed and observe as much as 4× increase in response times of Apache web server during different types of interference. We study network I/O, CPU, LLC, and main memory capacity interference in this dissertation. Under network I/O interference, the 90% percentile of Apache response time increased by 5×. Under CPU interference, the 90% percentile of Apache response time increased by 4×. Under LLC interference, the 90% percentile of Apache response time increased by 15×. Under memory capacity interference, the Apache response time was very noisy and unstable, that indicated an abnormal situation for the co-located VMs and the underlying PM.

2.4 Server Utilization Analysis

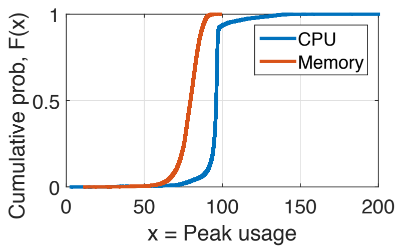
We analyze real-world server resource usage to motivate further research on dynamic scheduling and improving resource utilization in cloud data centers. Several service operators have recently released resource usage traces for their data center servers. We analyze the resource usage of the following three traces:

1. Server-level resource utilization traces (CPU, memory, disk I/O) from Google (2011).
2. VM-level resource utilization traces (CPU) from Microsoft Azure (2016-2017).
3. Server-level resource usage traces (CPU and memory) for colocated workloads from Alibaba (2018).

Of these, the Alibaba trace is most relevant to our motivation, so we briefly discuss our analysis of the Google and Azure traces first, and then explain the analysis of the Alibaba trace in detail.



(a) Average usage timeline.



(b) CDF of peak usage.

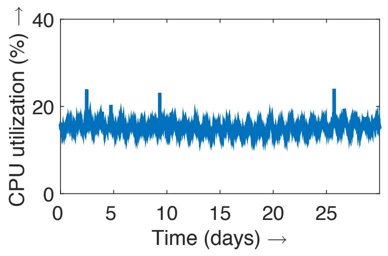
Figure 2.3: *Average and peak utilization of CPU and memory for the Google cluster trace.*

2.4.1 Google Server Resource Usage Traces

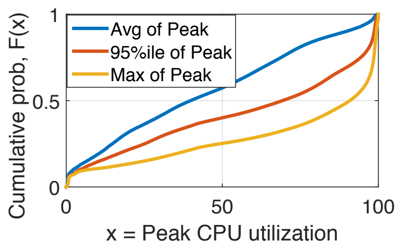
The Google trace contains resource usage information for a cluster of about 12,500 servers over a period of 29 days from May 2011 [150, 114]. Of relevance to us is the CPU and memory usages for the tasks scheduled on the PMs and the normalized resource capacity of the PMs. Disk I/O times are also provided, but only for a portion of the time range, and the normalized per-server disk capacity details are unavailable. The task-level resource usages are reported in 5-minute intervals and can be used to find per-machine CPU and memory utilization.

Figure 2.3(a) shows the timeline plot for average CPU and memory utilization averaged over all PMs in the trace. We see that the average CPU and memory utilization is around 40.9% and 47.4%, respectively. Figure 2.3(b) shows the CDF of per-server peak (overall intervals) CPU and memory utilization across all servers. We find that the median of peak usage is 79.6% and 96.1% for CPU and memory, respectively. This shows that peak usage for servers is high; however, note that the peak is computed over the 29 days length of the trace. Also, note that the peak usage is sometimes greater than 100% for CPU; this happened for nearly 6.6% of the servers. This is either an anomaly or a result of hyper-threading which was not accounted for by the normalization used in the trace.

Summary: *We note that the utilization for this cluster is moderate, but there is room for improvement, as suggested by the unused resource utilization in Figure 2.3(a).*



(a) Average usage timeline.



(b) CDF of peak usage.

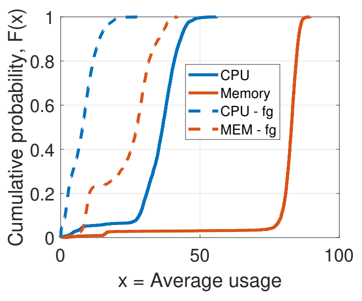
Figure 2.4: *VM-level CPU utilization for the Azure trace.*

2.4.2 Azure VM-Level Resource Usage Traces

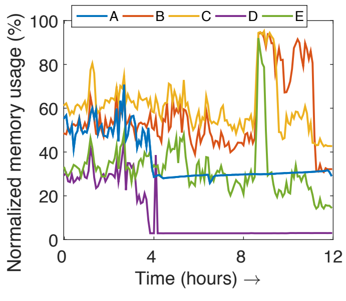
The Azure trace contains first-party VM CPU utilization data from one region [22]. The trace spans over 30 days and reports (only) CPU utilization (min, average, and max) of over 2 million VMs, every 5 mins, over their lifetime.

Figure 2.4(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 2.4(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.*



(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 2.5: *Analysis results for the Alibaba cluster trace.*

2.4.3 Alibaba Server Resource Usage Traces

The Alibaba production cluster traces [1] contain server-level CPU and memory usage sampled every 10s for about 4,000 servers over 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 2.5(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 2.5(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.*

Chapter 3

Related Work

Scheduling is a critical component for applications running on a cluster of nodes; thus several prior works have tried to address various challenges of scheduling. In this dissertation, we focus on scheduling in cloud environments, which is more challenging because of performance interference, as discussed in Section 2.3. This chapter reviews prior work in the broad area of scheduling and interference management to make a case for our contributions.

We first discuss the important and recent prior work on workload scheduling and cluster management in cloud environments in Section 3.1. Scheduling background workloads in a best-effort manner is a common practice to improve resource utilization in cloud environments while guaranteeing that foreground/primary workloads' performance is not affected. Section 3.2 thus reviews the related work on background workload scheduling. Accordingly, improving resource utilization in private clusters and public cloud servers are discussed in Section 3.2.1 and Section 3.2.2, respectively. Despite the several prior works on this context, Section 3.2.3 illustrates that many of these related work only regulate the usage of specific resources; this motivates our work which regulates multiple resources. Performance interference is a critical challenge for cloud computing, as we discussed in detail in Section 2.3. Therefore, Section 3.3 discusses interference management in cloud environments. In particular, Section 3.3.1 reviews related work on interference detection, Section 3.3.2 illustrates prior work on interference-aware performance management, Section 3.3.3 discusses the limitation of provider-centric interference management approaches, and Section 3.3.4 motivates the need for a user-centric interference management solution.

3.1 Scheduling in Cloud Environments

Scheduling is a broad concept and can be used in many contexts. In this section, we review the important prior work on request scheduling in cloud environments. The high-level problem is to schedule requests (of workloads) on a cluster of VMs or servers while satisfying the workloads' performance requirements. Note that we use term request in a broad context ranging from short-lived jobs (e.g., short background job) to long-lived jobs (e.g., cloud tenant VM). Having jobs with different requirements, such as different resource demand (potentially dynamic) and priorities, makes scheduling a challenging problem.

Borg [141] 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 priority based on their functionality (such as high-priority latency-sensitive jobs and low-priority batch jobs). To accommodate higher priority jobs on a machine, Borg starves, and can even kill lower priority jobs. Since all tasks of jobs in the cluster are known to Borg, it is aware of their resource requirements and priorities. Scavenger has a similar goal as Borg when it comes to performance isolation, but we consider a public cloud environment (as opposed to a private cluster) where foreground jobs are black-box customer VMs; thus, we do not have the same cgroup resource regulation tools at our disposal when managing foreground jobs. Further, all customer VMs have to be treated as having the same (high) priority in our case.

Mesos [58] is a cluster sharing platform that has been developed by researchers in UC Berkeley (presented in 2011), and Mesos open-sourced platform was announced in 2016 by Apache Software Foundation. Mesos allows various frameworks to share clusters efficiently. A framework is a software system that manages and executes one or more jobs on a cluster. Mesos aims to be a generic platform with minimal size; thus Mesos leaves the interval scheduling to the frameworks and focuses on how many resources every framework can get using common policies (e.g., fair sharing). To do so, Mesos introduces the concept of *resource offers* to enable fine-grained sharing across frameworks. Each framework running on Mesos consists of two components: a scheduler that registers with the master to be offered resources, and an executor process that is launched on slave nodes to run the framework's tasks. While the Mesos master determines how many resources to offer to each framework, the frameworks' schedulers select which of the offered resources to use. OS container technologies are used to isolate resources between framework executers running on the same slave. While Mesos offers specific resource vectors (e.g., CPU cores and memory, to name a few) to the frameworks, it does not handle scenarios where the frameworks running in the background use just the idle resources and do not affect the foreground workloads' performance. Our background workload scheduler, Scavenger, can extend Mesos to allow frameworks to run their jobs in the best-effort manner. We implement Scavenger on top of Apache Yarn [139], which is another popular resource management framework.

Bistro [50] is a job scheduler that runs data-intensive batch jobs next to online customer workloads in Facebook's production systems. Bistro uses a hierarchical resource models to address possible contention at multiple levels, including data volumes, host, and rack. To avoid disrupting foreground jobs, Bistro constraints the resource capacity allocated to batch jobs by manually configuring this available capacity based on the characteristics of the foreground jobs. While Facebook knows the resource demand patterns of their foreground jobs, this is not always true. In a public cloud, the provider only has details of the requested VM size, which, as we saw from the VM-level Azure traces in Section 2.4.2, can be very conservative.

3.2 Background Workload Scheduling in Cloud Environments

In order to put our work on scheduling background workloads in context, we now discuss related works that address the underutilization problem by colocating background jobs with foreground jobs.

3.2.1 Improving Resource Utilization in Private Clusters

The problem of resource underutilization has been around since before shared public clouds. In private clusters, where the provider has full knowledge of all running applications, the underutilization problem can be solved by colocating batch jobs that complement the known resource usage patterns of foreground jobs in the cluster. In some cases, the performance requirements of the foreground jobs are also known, so their resource demand can be regulated as well. Recall the moderate utilization numbers for the Google cluster traces analyzed in Section 2.4.1; this represents a private cluster.

Heracles [82] combines software and hardware isolation mechanisms to run batch jobs next to latency sensitive jobs. The Heracles controller measures foreground job latency every 15 seconds to decide on colocation. Heracles focuses specifically on dedicated cluster environments where the provider is aware of the foreground application and its SLOs, and the provider can benchmark the performance of foreground jobs with different levels of colocation. This profiling approach is not feasible for public cloud environments where the cloud provider can not (or should not) benchmark or profile black box customer VMs.

PARTIES [13] is a recently proposed resource controller that mitigates SLO violations between colocated latency-sensitive applications using software and hardware mechanisms. PARTIES assumes that the applications' SLO requirements and current performance information is known to the provider, making it suitable for private clouds, but not public clouds.

PerfIso [60] 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. This approach may be infeasible in public clouds where the provider cannot profile black box tenant VMs. Further, as we show throughout our results, isolating CPU cores alone does not mitigate processor cache contention.

3.2.2 Improving Resource Utilization in Public Cloud Servers

In public cloud environments, the foreground (customer) VMs cannot be controlled and their resource demands should be met at all times based on their VM sizes. However, their resource usage pattern can be studied to make predictions, if possible.

Zhang et al. [169] rely on historical usage patterns of CPU and disk usage to predict the required resources for customer VMs; the remaining spare compute cycles and storage space are then leveraged by the provider's batch workloads. Resource Central [22] used a similar approach to colocate production and non-production VMs on Azure cloud servers to increase CPU utilization. TR-Spark [158] aims to run Spark on transient VMs, such as spot instances, which can be run by the provider in the background. The main idea is to introduce checkpointing to allow job progress in spite of worker failures by modifying Spark's Task Scheduler and Shuffle Manager. MOON [76] provides a similar solution, but for Hadoop jobs, by using transient resources to host data replicas and intermediate data. However, TR-Spark relies on prediction of worker failures, suggesting that changes in the foreground workload can be predicted.

In general, customer workloads need not follow specific patterns and may not be predictable [45]; the performance loss due to misprediction and subsequent oversubscription can be expensive and may result in loss of customer revenue [28].

3.2.3 Regulating the Usage of Specific Resources

There have been prior works on regulating the resource utilization of specific resources, such as CPU and network. These works typically focus on the scheduler or the device driver to prioritize foreground jobs.

dCat [153] presents a cache performance isolation approach by exploiting the CAT technology (cache allocation technology [98]) 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. Further, dCat only considers LLC interference. QJUMP [53] addresses in-network interference by defining priority levels for packets, allowing foreground job packets to jump-the-queue over background job packets. This mitigates the increase in switch queueing of foreground jobs caused by batch jobs under colocation. PerfIso [60] uses a foreground application performance profiling approach to determine the number of CPU cores that can be safely allocated to background jobs. MIMP [170] proposes a similar CPU scheduling policy that allows background Hadoop jobs to run only when foreground VMs are not actively utilizing the CPU. The authors also modify the Hadoop scheduler to prioritize jobs that can best utilize the variable resource capacity that is unused by the foreground. CPI^2 [167] employs statistical approaches to analyze an application's Cycles-Per-Instruction (CPI) metric to detect and mitigate processor interference between threads of different jobs. However, CPI^2 only handles processor interference. Tableau [136] is a scheduler for Xen that mitigates CPU interference among VMs (all foreground) by scheduling them according to their complementary resource demands. Dirigent [171] 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 [127] uses a similar idea to leverage idle cores for running batch jobs.

The above works target a specific resource contention under colocation. In general, several resources may simultaneously be under interference [63, 84]. Further, as shown in Section 5.5, managing the contention at a single resource, such as CPU, will not suffice. Thus, there is a need for solutions, such as Scavenger, that address multiple, concurrent resource contentions.

3.3 Interference Management in Cloud Environments

Interference among cloud applications has received significant attention from researchers because of the severe performance degradation caused by interference and because of its complex and uncertain nature [42, 137]. Interference management is a key component of this dissertation, thus we now discuss related work in this area in detail.

3.3.1 Interference Detection

Recent work has emphasized the need for user-centric interference detection [85, 84, 12, 61]. IC^2 [85] employs decision trees using VM-level statistics to detect interference at the cache; this information is then used to tune the configuration of web servers in co-located environments. Casale et al. [12] focus on CPU interference and present a user-centric technique to detect contention by analyzing the CPU steal metric. CRE [2] makes use of collaborative filtering to detect interference in web services by monitoring response times. While we also monitor response time, we go beyond detection and also estimate the amount of interference. CPI^2 [167] employs statistical approaches to analyze an application's CPI metric to detect and mitigate processor interference between threads of different jobs. While CPI^2 can be used in virtual environments, public cloud VMs (e.g., AWS) do not always expose performance counters.

There have also been studies on interference detection from the perspective of the hypervisor (e.g., ILA [10], TRACON [15], and DejaVu [138]). While useful, such techniques require hypervisor access for monitoring host-level metrics (e.g., global CPU usage from Dom0 or hardware counters), which are not always feasible for cloud users.

There are also techniques [26, 19, 68] that automatically diagnose performance issues; however, such methods are not tailored for timely interference mitigation.

3.3.2 Interference-Aware Performance Management

ICE [84] proposes interference-aware load balancing by limiting the CPU utilization of the affected VM below a certain threshold. While effective, we find, via experiments (see Section 4.5.3), that this strategy is not adaptive to different levels of interference. Mukherjee et al. [61] propose a tenant-centric interference estimation technique that employs a software probe periodically on all tenant VMs, and compares the performance of the probe at runtime versus that in isolation to quantify interference. The authors later extended this work to PRIMA [94], which is an interference-aware load balancing and auto-scaling technique that leverages the above-described probing technique to make load balancing decisions. However, PRIMA only focuses on mean response time (as opposed to the more practical tail response time metric) and limits itself to network interference. Bubble-Up [88], Tarcil [32] and Quasar [31], and ESP [95] profile workload classes and carefully colocate workloads that do not significantly impact each others' performance due to their specific resource requirements. By contrast, DIAL does not control colocation (since VM placement is typically not in the user's control); instead, DIAL globally adjusts the LB policy of the fg application to reroute some of the requests directed at affected VMs. Interference-aware server reconfiguration [85] attempts to tune the parameters of the server or VM, such as Timeout and MaxClients, to minimize the impact of interference; this is complementary to our work that focuses on load balancing.

There is also prior work on interference-aware scheduling (e.g., Paragon [30]), migration (e.g., DeepDive [99]), resource provisioning (e.g., Q-Clouds [97]), and placement (e.g., CloudScope [14]) of VMs. However, such techniques require hypervisor-level visibility and control, and are thus orthogonal to our user-centric focus in this dissertation.

Prior work on load balancing typically focuses on adaptive strategies to deal with imbalance in load caused by server faults or congestion. In addition to the heuristic policies considered in Section 4.5.3, Tiwari et al. [131] propose an extension to Round Robin for heterogeneous servers that considers queue lengths. HALO [46] proposes extensions to Round Robin and other policies for heterogeneous servers focusing on mean response time. Such policies are not designed for interference, but could be extended. Another option is to simultaneously send requests to multiple servers to reduce latency [27]. Again, while not designed for interference, such policies can help, though at the cost of additional resources. Recently, Sparrow [103] proposed a probabilistic scheduler that samples the queue length at a few servers, and picks the one with the lowest load. While we do not focus on scalability in this dissertation, such sampling approaches can be integrated with DIAL for operating in large clusters.

3.3.3 Limitations of Provider-Centric Solutions

Cloud providers have taken steps to address performance interference. To prevent interference, providers try to isolate and partition hardware resources, such as CPU, that a user VM is allocated. However, low-level hardware resources, such as processor caches and memory buses, are notoriously hard to isolate without incurring significant overhead [160, 85, 70]. Our experiments on AWS EC2 using microbenchmarks confirm the presence of LLC contention and disk I/O contention (see Section 4.5.3). Regardless of the ease and feasibility of isolation, limiting resource sharing among colocated VMs can lower utilization, defeating the purpose of cloud computing [84, 97]. In fact, cloud platforms, such as OpenStack, overcommit CPU and memory by default to increase resource utilization.

To further mitigate interference, prior work suggests carefully co-scheduling VMs that do not interfere [10, 15, 97, 30]. Unfortunately, it is not always possible to predict the resource usage of every colocated VM [85, 84], especially since providers are unaware of software and applications installed by users on their VMs [99, 42]. As a result, interference can unpredictably emerge or cease on any colocated VM; statically scheduling VMs will thus not suffice. While VM migration can help address dynamic interference, there are several issues that make this approach infeasible: (i) the migration process can itself take on the order of minutes to complete [79, 96], (ii) migration can lead to new interference among colocated VMs on the target host [155], and (iii) migration can incur substantial compute and communication overheads [152, 143].

In practice, provider-centric solutions also suffer from the fact that they are not aware of the user deployment. For example, providers might not be aware of the SLO requirements of the user application, or the deployment details, such as the different tiers of the application and resource bottlenecks at each tier. In fact, providers might not be able to access application-level performance metrics, such as per-tier latencies, workload request mix, session lengths, etc. Given these limitations, we explore a different class of solutions.

3.3.4 The Case for User-Centric Solutions

User-centric solutions aim to address interference from within the user's application deployment without requiring any assistance from the cloud provider or other cloud users. The user deploys such solutions, and thus have the advantage of being aware of, and having visibility into, the application deployment and SLO requirements.

While provider-centric solutions can be useful in specific scenarios, for example, where the user and provider belong to the same organization and thus share visibility and control of the application [168, 30], this is not always the case. In several private cloud deployments, the provider only maintains the infrastructure and is not responsible for users' SLOs. In general, it is unlikely that cloud providers will engage with every cloud user to mitigate their performance interference problems. User-centric solutions can address this gap by directly empowering the users to mitigate interference in their deployments. Further, such solutions can complement provider-centric solutions, especially when provider efforts to mitigate interference are not enough to avoid specific SLO violations for user applications.

Chapter 4

Interference-Aware Load Balancing

Request scheduling for applications in cloud environments faces a new challenge–the cloud resources, such as Virtual Machines (VMs) and containers have unpredictable performance due to the underlying physical servers' resource contention. Despite several provider-centric interference mitigation efforts, performance interference is still a major concern that can degrade the cloud-deployed applications' performance significantly. In this chapter, we propose a *user-centric* interference mitigation approach which can be employed by the tenant without requiring any assistance from the provider or hypervisor. Such user-centric solutions empower the tenant to have greater control over their application performance, which is often the most important criteria for users.

There are several types of applications in cloud environments; we focus on load-balanced applications that are common in the cloud. In a load-balanced application, there is a load balancing tier with one or more load balancers that are responsible for dispatching incoming requests among a cluster of nodes (e.g., VMs).

We present DIAL, a dynamic solution for mitigating interference in load-balanced cloud deployments. The key idea behind DIAL is to *infer* contention in colocated VMs and *adapt* the load distribution of incoming requests among user VMs accordingly. In other words, DIAL addresses the request scheduling challenge by taking the potential change in the cloud resources' capacity into account to decide which node should run an incoming request.

Part of the results from this chapter was published in ICAC 2017 [63]. We introduce the problem and discuss the scope of this chapter in Section 4.1. We then illustrate our proposed user-centric approach for estimating the severity of resource contention in Section 4.2. Using the estimation results, we discuss how DIAL calculates load balancing weights in Section 4.3. We put all the components together and discuss the DIAL control flow in Section 4.3.3. We implement DIAL for two different types of applications, namely web applications and OLAP applications; thus we first discuss our generic evaluation methodology in Section 4.4, and then provide the evaluation results for web and OLAP applications in Section 4.5 and Section 4.6, respectively.

4.1 Overview and Scope

Applications deployed in the cloud can experience undesirable performance effects, the most severe of which is *interference*. Performance interference is caused by contention for physical resources, such as CPU or LLC, among colocated VM users/tenants. Prior work has shown that interference in public and private cloud environments can degrade application performance by as much as 27× [145, 157, 10].

Interference is an undesirable side-effect of a fundamental design principle of the cloud, namely, *multi-tenancy* (sharing of a physical server among users). While specific resources, such as CPU, can be partitioned among colocated VMs by cloud providers, other resources, such as processor caches, are notoriously hard to partition [84]. Nonetheless, the partitioning of resources among tenants can adversely impact cloud resource utilization. Further, resource contention depends on the workload of all colocated tenant VMs, and is thus *dynamic and unpredictable* [85]; as a result, static partitioning is not a useful solution.

Prior work on interference mitigation has typically focused on provider-centric solutions. A popular approach is to profile applications and co-schedule VMs that do not contend on the same resource(s) [10, 15, 138, 97, 30]. However, since interference is dynamic and can emerge unpredictably, statically co-scheduling VMs will not suffice. VM migration can help in this case, but interference is volatile and short-lived, often lasting for only a couple minutes [85]; by contrast, migration can take several minutes [96] and can incur overheads [32], especially for stateful applications [31].

A critical challenge that has not been addressed with regards to interfer-

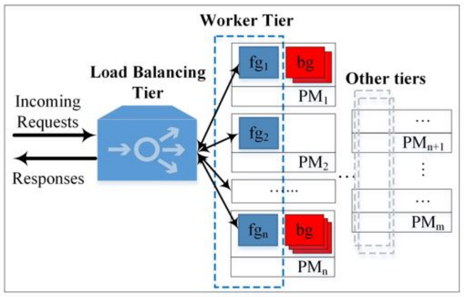


Figure 4.1: Illustration of a generic cloud application containing LBT and worker tier deployed on multiple foreground (fg) VMs experiencing interference from background (bg) VMs.

ence is the *lack of visibility and control between the provider and the tenant*, especially in public clouds [42]. Specifically, tenant VMs in public can not, or should not, be profiled a priori by the provider due to privacy concerns [99]. Further, providers are not always aware of the cloud user's Service Level Objective (SLO) requirements or the user application's bottleneck resources.

We present DIAL, a dynamic solution for mitigating interference in load-balanced cloud deployments. We consider a generic cloud-deployed application that has a tier of worker nodes hosted on multiple VMs and experiencing unpredictable interference from colocated VMs (owned by other cloud tenants), as shown in Figure 4.1. The incoming load is distributed among the worker nodes via one or more load balancers. This generic model is widely applicable, for example, for web applications (where workers are web or application servers), Online Analytical Processing (OLAP) systems like Pinot [66], etc.

Figure 4.1 illustrates a typical multi-tier cloud deployed application. The worker nodes process the incoming requests, and are illustrated as a tier of VMs. Incoming requests are distributed among the worker nodes using an application-specific scheduler or dispatcher or load balancer; we abstract this entity as a Load Balancing Tier (LBT). Our focus in this dissertation is on the worker nodes and the LBT; specifically, we propose a new technique to dynamically infer the interference on worker nodes and adjust the load balancing weights for the worker VMs in the LBT. We assert that the LBT is ideally suited to mitigate the effects of volatile interference on worker VMs as the LBT acts at the front-end for the worker tier.

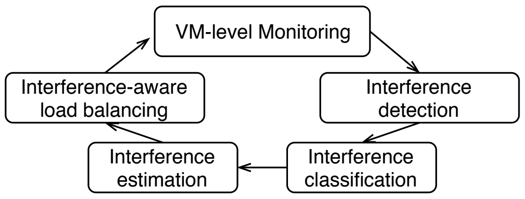


Figure 4.2: Illustration of DIAL's control flow.

The worker tier is hosted on multiple foreground (fg) VMs, each of which is hosted on a physical machine (PM); we highlight the worker tier fg VMs in Figure 4.1. Each PM may also host background (bg) VMs that do not belong to the fg user, as shown in Figure 4.1. The fg and bg VMs on a PM can contend for shared physical resources, such as CPU, network bandwidth (NET), disk I/O bandwidth (DISK), and last-level-cache (LLC), resulting in interference. Note that the fg user does not have visibility into the bg VMs; in fact, the fg user is unaware of bg VMs.

4.1.1 DIAL Overview

Our solution, DIAL, is a user-centric dynamic Interference-Aware Load Balancing framework. The design of DIAL addresses two key questions:

(i) How can users estimate the interference that their VMs are experiencing without any assistance from the provider, hypervisor, or colocated users? (Section 4.2)

(ii) Given this information, how should users dynamically distribute load among their VMs to minimize tail latencies in the presence of interference? (Section 4.3)

The key idea in DIAL is to estimate, from within a user VM, the amount of interference being induced by colocated VMs, and then adapt the incoming load intensity for each user VM accordingly. Figure 4.2 shows a high-level overview of DIAL's control flow. DIAL monitors performance metrics from within the VMs and signals interference if tail latency goes above a certain threshold (detection, see Section 4.2.1). DIAL then determines if the detected event is a load change for the application or a resource contention event (classification, see Section 4.2.2). Depending on the source of contention, DIAL quantifies, or infers, the severity of resource contention (estimation, see Section 4.2.3). Based on this quantification, DIAL determines the theoretically-optimal load balancing weights that the user application should employ to mitigate the impact of contention (see Section 4.3). The above steps are continually employed at runtime, enabling DIAL to respond dynamically to contention.

4.2 User-Centric Estimation of Interference

To effectively mitigate interference, DIAL must first estimate the amount of interference that each user VM is experiencing. To this end, we define *amount of interference: the fraction of available physical resources that are in use by colocated background VMs.* In the context of Figure 4.1, the amount of interference is the fraction of physical resources on a PM that are in use by the colocated bg VMs, and are thus unavailable to the fg VM on that PM. As we show below, estimating the amount of interference is non-trivial as it requires classification and modeling of interference.

4.2.1 Impact of Interference on Tail Latencies

Interference is known to impact application response times [145, 157, 10]. DIAL leverages this fact to estimate the amount of interference that an fg VM is experiencing because of resource contention created by colocated bg VMs. Specifically, DIAL aims to infer the amount of interference, or resource contention, that the bg VMs must be creating to effect the observed rise in fg response times.

Figure 4.3 shows the impact of different types of resource contention on the 90%ile response time of an OpenStack cloud-deployed Apache web server VM hosting files and driven by the httperf load generator. We create contention for this fg VM by running various microbenchmarks in colocated bg VMs. The x-axis denotes the percentage of total resource usage, which is the sum of resource usage by the fg VM and all colocated bg VMs, normalized by peak resource capacity or bandwidth. For example, if the total network bandwidth usage is 80MB/s, and the peak network bandwidth is about 115MB/s, then the resource utilization is $80/115 \approx 0.7$. We make two observations from this figure: (i) *response time increases considerably under interference*, (ii) *the relationship between total resource usage and response time depends on the exact resource under contention.*

Detecting interference: DIAL uses the first observation to *detect* when

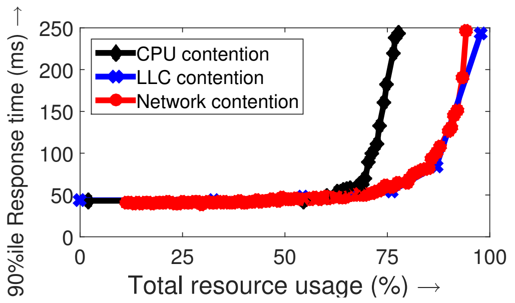


Figure 4.3: Performance of an OpenStack-deployed Apache web server under interference from colocated VMs running microbenchmarks.

the fg application VM is under interference. Specifically, from Figure 4.3, we see that application response times, or T_x, are initially *low and stable* (left of the graph). However, once the total resource usage increases (right of the graph), because of the increased resource demand from bg VMs, the fg response times *rise sharply*. Thus, DIAL signals interference when T_x goes beyond the 95% *confidence intervals* (around the mean of periodically monitored tail latencies) observed during no or low interference.
Need for identifying the source of interference: The second observation suggests that using tail response times to estimate interference will require knowledge of the specific resource that is under contention.

4.2.2 Classifying Interference Using Decision Trees

Our next task is to classify the source of interference, which is defined as the *dominant resource under contention*. Note that it is possible for several resources to be simultaneously under contention; however, we only consider dominant resource contention. We defer the analysis of multiple resources under contention to future work. Our key idea in classification is to observe the impact of interference on easily observable user metrics such as CPU utilization, I/O wait time, etc., which can be obtained from within the VM via the `/proc` subsystem. In general, one can monitor all available metrics and use feature selection algorithms, such as LASSO or ridge regression, to derive the list of useful features, which can then be used for classification.

DIAL uses decision trees to classify contention. The decision tree classifier is trained by running controlled interference experiments using microbenchmarks and monitoring the metrics in each case. After training, the decision tree can classify the source of interference, even for unseen workloads, based on the observed metric values (Section 4.5.3).

Distinguishing interference from workload variations: An application's response time can degrade for various reasons, such as workload surges, in addition to interference. While our detection methodology detailed in Section 4.2.1 does not distinguish between interference and workload variations, DIAL makes this distinction at the classification stage by again leveraging the decision tree classifier. Specifically, To distinguish interference from workload variations, DIAL normalizes the observed metric values with *predicted* values based on monitored workload intensity. Prior work has shown that linear models can accurately predict CPU usage based on workload intensities [164]. We thus use linear regression to predict the metric values as a linear function of the number of requests seen in the past monitoring interval.

The intuition behind this approach is that, in the absence of interference, the normalized values will be close to 1 under workload variations. The decision tree can thus use the deviation of the observed metrics from the normalized metrics to distinguish workload changes from interference.

4.2.3 Queueing-Based Model for Interference

The final step is to use the classification information to estimate the amount of interference, which is the *fraction of resources that are in use by colocated bg VMs.* Once we have these estimates, DIAL can redistribute incoming load accordingly to mitigate the impact of interference (Section 4.3).

From Figure 4.3, we see that tail response times increase non-linearly with the total usage of the resource under contention. Recall that the total resource usage is the sum of resource usage of the fg VM (can be monitored by the fg user) and all colocated bg VMs (cannot be monitored by the fg user). Our key idea is to model this non-linear relationship for each resource; this allows inferring the resource usage of the colocated bg VMs based on observed fg tail latencies, which in turn gives us the amount of interference.

Modeling interference: We employ queueing theory to model the non-linear relationship between resource usage and tail latencies. Queueing models suggest that the tail response time for an application is inversely proportional to the unused capacity of the VM [56]. Mathematically, $T_x \sim 1/(1-\rho_{fg})^{\alpha}$, for some parameter α, where ρ_{fg} is the resource load of the fg application (such as CPU utilization or I/O bandwidth utilization), normalized to peak resource usage; that is, $0 \leq \rho_{fg} \leq 1$. Prior work [59] has shown that $\alpha = 2$ works well for practical settings given the high variability in real workloads. Prior theoretical work has also shown that a quadratic term in the denominator can result in better predictability under high loads [113]. However, such models do *not* account for interference.

Under interference, the fg application experiences congested resources due to colocated bg VMs. As a result, the application experiences higher load than it would in the absence of interference. We model this effect by adding the resource usage of colocated bg VMs to that of the fg VM, resulting in fg response times being inversely proportional to $(1 - (\rho_{fg} + \rho_{bg}))$. The sum of loads exerted by the fg and bg VMs, $(\rho_{fg} + \rho_{bg})$, represents the normalized total resource utilization. We thus approximate x%ile response time as:

$$T_x = c_0 + c_1/(1 - \rho_{fg} - \rho_{bg}) + c_2/(1 - \rho_{fg} - \rho_{bg})^2, \tag{4.1}$$

where $\vec{c}$ is the coefficient vector that depends on the *specific resource under contention*. The polynomial function in Eq. (4.1) is inspired by prior work on queueing systems [113, 9] to interpolate between low load and high load regimes.

To determine the coefficients, we train the model in Eq. (4.1) by creating different levels of resource usage and monitoring the T_x of fg VMs (see Section 4.3.3). We then use multiple linear regression over this training data to derive the resource-specific coefficients. While Eq. (4.1) is inspired by queueing models, it can accurately track the relationship between tail response times and resource usage for realistic web applications, as we show in Section 4.5.

Applying the model to estimate interference: Eq. (4.1) can be easily employed to estimate the amount of interference. After detection and classification, the fg user can estimate ρ_{bg} by monitoring T_x and ρ_{fg}, and solving Eq. (4.1) for ρ_{bg}.

4.3 Interference-Aware Load Balancing with DIAL

Interference-aware load balancing is the key component of DIAL. When there is no interference, balancing the load among VMs works well to provide low response times. However, if one of the VMs is facing interference (can be estimated via the above-described interference modeling), then its share of the load must be adjusted accordingly. One might think that reducing the share of load in proportion to the available capacity at the compromised VM, $(1 - \rho_{bg})$, should work well. Unfortunately, this approach can be far from optimal, as we show via experiments in Section 4.5.3.

4.3.1 Minimizing Tail Response Times for Web Applications

To minimize application tail response times under interference, we again employ queueing theory. We first consider the case where any VM can serve an incoming request, as in the case of a web application tier. Consider a cluster of n VMs, with VM i facing interference of $\rho_{bg,i}$. Let the fraction of total incoming load that is directed to VM i be p_i; we refer to p_i as the weight assigned by the load balancer (LB) to VM i. If the total arrival rate for the application is a, the arrival rate for VM i is $a \cdot p_i$. Our goal is to determine the p_is that minimize the x%ile response time, T_x.

To obtain a simple closed-form expression for the theoretically optimal p_is, we model each VM as an M/M/1 system. By focusing on the dominant resource that is causing interference, as classified using the decision tree, we employ the M/M/1 model to represent the contention at the dominant resource. While this is an oversimplification, the resulting closed-form tail latency expression enables the optimization and determination of theoretically-optimal load balancer weights. We note that the resulting p_is are only optimal under the M/M/1 model; we refer to these as the "theoretically optimal" weights in the rest of the chapter.

For the M/M/1 model, the response time is known to follow an Exponential distribution [56]. We can thus obtain any tail probability of response time by using the CDF of the Exponential distribution. Under the M/M/1 assumption, T_x for a cluster of n VMs is approximated as:

$$T_x \approx \sum_{i=1}^{n} p_i \cdot \frac{-\ln(1 - x/100)}{r_i - a \cdot p_i}, \tag{4.2}$$

where r_i represents the throughput of VM i (with contention). Since interference reduces the throughput of the compromised VM, we set $r_i = r \cdot (1 - \rho_{bg,i})$, where r is the peak throughput of an application VM. For example, if the peak throughput of our Apache server is $r = 1000$ req/sec, and it is experiencing an estimated interference of $\rho_{bg} = 0.6$, then we set $r = 1000 \times 0.4 = 400$ req/sec.

Eq. (4.2) above works for all percentiles of response time. For example, if $x = 90$, meaning we focus on the 90%ile response time, then the term in the numerator becomes $-\ln(1 - 0.9) = \ln 10$. For 95%ile response times, the numerator becomes $\ln 20$. Interestingly, the optimization for p_is discussed below does *not* depend on the numerator value (since it is independent of p_i), and thus *our results apply, as-is, for any percentile of response times*, including the median.

Given a (which can be monitored) and r_i (derived as discussed above using interference estimation from Section 4.2), T_x can be expressed as a function of p_i via Eq. (4.2). We can now derive the theoretically optimal weights, p_is, that minimize T_x in Eq. (4.2) via calculus, as presented below (proof omitted for ease of presentation).

Lemma 1. *The theoretically optimal load split for minimizing T_x for a cluster of n VMs with total arrival rate a and individual VM throughputs r_i is given by:*

$$p_i^* = \left(r_i \sum_{j=1}^{n} \sqrt{r_j} - \sqrt{r_i} \sum_{j=1}^{n} r_j + a\sqrt{r_i} \right) / \left(a \sum_{j=1}^{n} \sqrt{r_j} \right) \tag{4.3}$$

Proof. The proof of optimality proceeds via mathematical induction on n. We first prove the base case for $n = 2$. Let the probability of sending a request to VM$_1$ (with throughput r_1) be p; thus, arrival rate into VM$_1$ is $a \cdot p$. Then, under the M/M/1 queueing model [56], the response time for VM$_1$ is distributed as $Exp(r_1 - a \cdot p)$. Based on this, the x%ile response time is $\frac{-\ln(1-x/100)}{r_1 - a \cdot p}$. Likewise, the x%ile response time for VM$_2$ (with arrival rate $a \cdot (1-p)$) is $\frac{-\ln(1-x/100)}{r_2 - a\cdot(1-p)}$. We now approximate T_x for the 2-VM system as:

$$T_x \approx p \cdot \frac{-ln(1 - x/100)}{r_1 - a \cdot p} + (1 - p) \cdot \frac{-ln(1 - x/100)}{r_2 - a \cdot (1 - p)}.$$

We now derive the optimal value of $0 \le p \le 1$ that minimizes T_x. After some algebra (taking derivatives of T_x w.r.t. p), we get

$$p_1^* = p^* = \frac{r_1\sqrt{r_2} - r_2\sqrt{r_1} + a\sqrt{r_1}}{a(\sqrt{r_1} + \sqrt{r_2})}.$$

Now assume that the above expression for p^* is true for $n = k$. Then, for $n = (k+1)$, we partition the $(k+1)$ VM system into a single VM with request probability p_n and a k-VM system with request probability $(1-p_n)$. For the k-VM system (with primed variables) with request rate $a' = a \cdot (1 - p_n)$, by the inductive hypothesis, we have:

$$p_i'^* = \frac{r_i \sum_{j=1}^{k} \sqrt{r_j} - \sqrt{r_i} \sum_{j=1}^{k} r_j + a' \sqrt{r_i}}{a' \sum_{j=1}^{k} \sqrt{r_j}}.$$

The approximate x%ile response time for the $(k+1)$-VM system can then be written as:

$$T_x \approx p_n \cdot \frac{-ln(1 - x/100)}{r_n - a \cdot p_n} + (1 - p_n) \cdot \sum_{j=1}^{k} p_i'^* \cdot \frac{-ln(1 - x/100)}{r_i - a' \cdot p_i'^*} \tag{4.4}$$

Note that T_x is itself a function of p_n since request rate for the k-VM system is $a' = a(1-p_n))$. We now derive the theoretically optimal p_n^* by differentiating Eq. (4.4) to get:

$$p_n^* = \frac{r_n \sum_{j=1}^{n} \sqrt{r_j} - \sqrt{r_n} \sum_{j=1}^{n} r_j + a\sqrt{r_n}}{a \sum_{j=1}^{n} \sqrt{r_j}}. \tag{4.5}$$

The remaining k theoretically optimal probabilities can then be derived by noting that

$$p_i^* = (1 - p_n^*) \cdot p_i'^*.$$

The resulting expressions match those given by Eq. (4.5), thus completing the proof by induction. □

Note that p_i^* depends on the estimates of r_i, thus necessitating the interference estimation of Section 4.2. Also note that p_i^* depends on the total arrival rate, a. This is to be expected since, for example, if the arrival rate is very low, we can send all requests to the VM with the highest throughput to minimize response times; however, if the arrival rate is very high, then a single VM cannot handle all requests, and we have to distribute the load. Importantly, both r_i and a can change unpredictably at any time (r_i due to interference and a due to variable customer traffic), motivating the need for a *dynamic* solution instead of existing static solutions.

4.3.2 Minimizing Tail Response Times for OLAP Applications

We now extend the above analysis to the case where only a subset of workers (replicas) can serve an incoming request due to data locality, as in the case of OLAP systems. Let the number of replicas be c. Let us first consider the case where one worker, say w, out of n, is under interference. Let the non-interference throughput be r and that of w be $r_w < r$.

In the absence of interference, $1/c$ fraction of requests that have a replica on w would be sent there by the LB; further, the arrival rate to w would be a/n, assuming a fair distribution of replicas among workers. In the presence of interference, let the fraction sent to w be p, and so the fraction sent to the remaining $(c-1)$ replicas is $\frac{1-p}{c-1}$. Thus, the arrival rate into w is now $\frac{a}{n} \cdot \frac{p}{1/c} = \frac{a}{n} \cdot p\ c$, and the fraction of *all* requests that go to w is $q = \frac{p\ c}{n}$. Likewise, arrival rate into *each* non-interference worker is $\frac{a}{n} + \frac{a}{n} \cdot \frac{1-p\ c}{(n-1)} = a \cdot \frac{1-q}{n-1}$, and the fraction of all requests that go to each non-interference worker is $\frac{1-q}{n-1}$.

Using the M/M/1 model [56], we have, similar to Eq. (4.2):

$$T_x \approx q \cdot \frac{-\ln(1-x/100)}{r_w - a \cdot q} + \cancel{(n-1)} \cdot \frac{1-q}{\cancel{n-1}} \cdot \frac{-\ln(1-x/100)}{r - a \cdot \frac{1-q}{n-1}} \tag{4.6}$$

Observe that Eq. (4.6) is exactly the same as Eq. (4.2) when $r_1 = r_w$ and $r_i = r$ for $i \neq 1$, except that p_1 is replaced by q. Thus, the theoretically optimal solution, via Eq. (4.3), is $q^* = p_1^*$, and thus the theoretically optimal split for the worker under interference is $q^* \cdot \frac{n}{c} = p_1^* \cdot \frac{n}{c}$. Intuitively, this result says that more load needs to be placed on the interference worker in case of OLAP when compared to web applications; this makes sense as there are fewer alternative workers in case of OLAP applications as opposed to web applications, i.e., $(c-1)$ as opposed to $(n-1)$. We can similarly obtain the theoretically optimal split when one more than worker is under interference.

4.3.3 The DIAL Control Flow

The control flow for our DIAL implementation (for web and OLAP applications) is as follows:

1. Monitoring: DIAL monitors the fg application's T_x, arrival rate, a, load, $\rho_{fg,i}$, and classification metrics (e.g., connection time), averaged every interval, for all fg VMs.

2. Detection: DIAL signals interference if T_x exceeds its 95% confidence bounds for successive monitoring intervals.

3. Classification: DIAL next employs the decision tree to identify the dominant resource under contention.

4. Estimation: DIAL then uses the T_x and $\rho_{fg,i}$ values in Eq. (4.1), with the dominant resource-specific coefficients, to estimate the interference, $\rho_{bg,i}$. The interference-aware throughput for fg VM i is adjusted by $(1 - \rho_{bg,i})$.

5. Interference-aware load balancing: Given these estimates, and the monitored a value, DIAL derives the LB weights, $\vec{p^*}$, via Eq. (4.3), and inputs them to the LB.

We continue monitoring the VMs' performance to detect further changes in interference and to detect the end of interference. To this end, we ensure that a small number of requests are sent to the affected VMs so we can monitor the progress of interference; we can also use probes for this purpose, as suggested by recent work [93]. When T_x returns to normal (for successive intervals), we reset the LB weights.

The monitoring interval length employed by the controller depends on the stability of the fg application and the noise in the system. We use a length of 10s based on the sensitivity analysis for our specific setup [63].

DIAL requires some model training to build the decision tree (Section 4.2.2) and derive the coefficients of the estimation model (Eq. (4.1)). The above training tasks can be performed offline on a dedicated server in a cloud environment by controlling the bg VMs to run microbenchmarks at different intensities while monitoring relevant metrics. In a private cloud environment, such as OpenStack, we can set aside a dedicated host using Availability Zones and Host Aggregates. In some public cloud environments, such as Amazon, dedicated hosts can be rented on a pay-as-you-go basis. We use these options for training the DIAL controller using a simple set of microbenchmarks and then highlight the performance improvements under a different set of realistic workloads that were not used for training.

4.3.4 Assumptions for DIAL controller training

The above-described DIAL control flow and training makes certain implicit assumptions about the incoming workload. Specifically, by training at different load intensities, DIAL assumes that (i) the workload request mix does not change significantly at runtime, and (ii) the distribution of inter-arrival times does not change significantly at runtime. When the request mix changes, for example, to a more database-heavy request mix, then the workload will have a greater sensitivity to disk I/O contention. This will thus require a retaining of the DIAL controller to infer the new model parameters. Note that the mean arrival rate and/or the number of workers may change dynamically, and this is already monitored by DIAL and is taken into account when determining the theoretically optimal load balancing weights via the a and n parameters, respectively. We show, in Section 4.6.3, that DIAL works well even under an abrupt request rate change and a change in the number of workers.

4.4 Evaluation Methodology

To evaluate the efficacy of DIAL, we implement it for realistic applications and study the reduction in tail latency when worker nodes face interference. This section describes the foreground applications for which we implement DIAL and background applications that are used to create contention for different resources. We also explain our resource monitoring approach for worker nodes.

4.4.1 Foreground (fg) Applications

Web applications: Figure 4.4 illustrates a typical multi-tier web application. The worker tier, which we focus on, is the application server tier and is highlighted in Figure 4.4. The incoming requests are distributed among application servers via a load balancer. Our focus in this dissertation is on the application tier which is behind the load balancer. The web application is hosted on multiple foreground (fg) VMs, each of which is hosted on a physical machine (PM); we highlight the application tier fg VMs in Figure 4.4. The incoming requests for the user application are load balanced among fg VMs via a load balancer (LB). Note that the fg user does not have visibility into the bg VMs; in fact, the fg user is unaware of bg VMs.

We employ two multi-tier web benchmarks as our fg application:

1. CloudSuite [39]: The CloudSuite 2.0 Web Serving benchmark is a multi-tier, multi-request class, PHP-MySQL based social networking

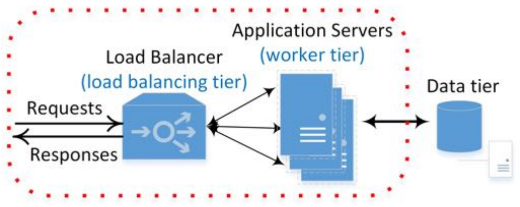


Figure 4.4: Illustration of a typical multi-tier web application.

application. The benchmark uses several request classes, e.g., HomePage, TagSearch, EventDetail, etc.

Our CloudSuite setup consists of: (i) Faban workload generator for creating realistic session-based web requests. We set the number of users to 1000 for OpenStack and 5000 for AWS; the think time is 5s (default). (ii) HAProxy LB distributes incoming http requests (from Faban) among the back-end application tier VMs. We use the default Round Robin policy, unless stated otherwise (as in Section 4.5.3 where we compare with other policies). (iii) Application VMs installed with Apache, PHP, Memcached, and an NFS-Client. We employ 3 application VMs in OpenStack and 10 in AWS. (iv) A MySQL server and an NFS server, hosting the file store, are installed on separate, large VMs (to avoid being the bottleneck).

2. WikiBench [135]: WikiBench is a Web hosting benchmark that mimics wikipedia.org. Our WikiBench setup consists of: (i) wikijector load generator to replay real traffic from past traces of requests to Wikipedia, (ii) HAProxy LB, and (iii) three VMs running the MediaWiki application (the same application that hosts wikipedia.org), and (iv) a MySQL database to store the Wikipedia database dump.

Unless specified otherwise, we use CloudSuite in foreground.

OLAP Applications: We use the open-source Pinot [108] system as our representative OLAP application. Figure 4.5 illustrates the Pinot architecture including the three main components that we focus on: (1) controller, (2) broker, and (3) historical worker nodes. Section 2.2.1 provided more details about Pinot. The brokers act as our load balancing tier (LBT), see Section 4.1. The historical worker nodes host data segments and respond to queries that originate from the broker. The worker nodes constitute our worker tier.

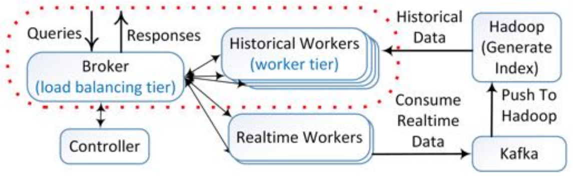


Figure 4.5: Overview of Pinot's architecture.

For all the above fg applications, we use the suggested default configuration values, resulting in average CPU utilization of about 25% for CloudSuite, 34% for WikiBench, and 62% for Pinot, without interference. Recent studies, including those at Azure [22] and Alibaba [1], reported average CPU utilizations of about 20% for fg VMs.

4.4.2 Background (bg) Workloads

In our experiments, we emulate interference by employing several bg workloads to create contention for the fg application. This approach of creating interference, which is similar to other prior works such as ICE [84], Paragon [30], and DeepDive [99], allows us to reproduce the same interference pattern to fairly evaluate performance with and without DIAL. The bg workloads are hosted on VMs colocated with the fg application layer VMs. Each fg VM under interference is hosted separately from other fg VMs, and is colocated with bg VMs. We first employ *microbenchmarks* to stress individual resources for analyzing fg interference. We then employ *test workloads* to evaluate DIAL for fg applications under realistic cloud workloads.
Microbenchmarks: We employ: (i) stress-ng tool on bg VMs to create controlled CPU contention; (ii) httperf load generator (on a separate VM and PM) to retrieve hosted files from the colocated bg VMs at different, controllable request rates to create NET contention; (iii) dcopy benchmark on bg VMs to create LLC contention; and (iv) stress on bg VMs to create DISK contention.
Test workloads: We employ: (i) SPEC CPU to create CPU contention, (ii) Memcache server (driven by mutilate client) to create NET contention, (iii) STREAM to create LLC contention, and (iv) Hadoop running TeraSort with a large data set to create DISK contention.

4.4.3 Resource Usage Monitoring

We study resource contention for four important resources: (i) network (NET), CPU, Last-Level-Cache (LLC), and disk. We now explain how we monitor fg and bg resource usage, from within the VMs, that is required for our model training.

- NET: We use the dstat Linux tool to monitor the used network bandwidth for bg and fg VMs. We then normalize their sum by the peak bandwidth.
- CPU: We consider fair-sharing of the possibly over-committed PM cores among VMs to compute CPU usage. If a PM has n cores available and all VMs together require m cores, then the CPU usage of each VM is normalized by $max\{m, n\}$. For example, a PM may have 12 cores ($n = 12$); if we launch 4 VMs with 4 vCPUs each on this PM, since oversubscription is allowed, then the total request is 16 ($m = 16$). If one of the VMs has a CPU usage of x% out of 400% (or y% out of 100%), then we estimate its CPU usage as $\frac{x}{16}$ (or $\frac{4 \cdot y}{16}$). Thus, if all 4 VMs are at 400% each (or 100% per vCPU), then total usage is 1 or 100%.
- LLC: Since memory bandwidth for a VM cannot be easily monitored, we employ the RAMspeed benchmark to measure the available memory bandwidth. We obtain this bandwidth for each experiment and then estimate the LLC usage by computing the difference between peak bandwidth and experiment bandwidth. Finally, we normalize this difference by peak bandwidth to estimate LLC usage.
- DISK: Disk usage typically depends on the access pattern (sequential vs. random). We thus use the same approach as for LLC, but with sysbench instead of RAMspeed, for estimating DISK usage.

4.5 DIAL for Web Applications

We first present our evaluation results for *web* applications; the next section focuses on OLAP applications. We first explain our DIAL implementation, and then present evaluation results for CloudSuite and WikiBench.

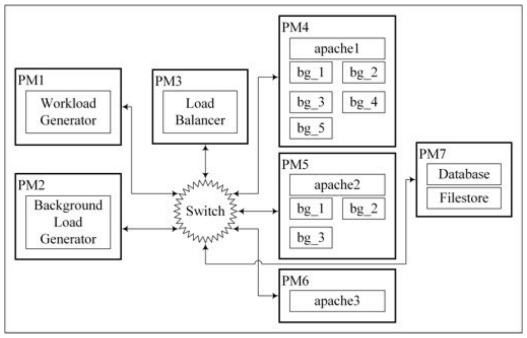


Figure 4.6: Illustration of our OpenStack cloud setup.

4.5.1 DIAL Implementation

For DIAL web application deployment, we implement the DIAL controller logic using: (i) a C program to execute the detection, classification, and estimation tasks, and (ii) a set of bash scripts to monitor metrics from the `/proc` subsystem (from within the VM) and the LB logs, and to communicate with the LB to reconfigure the weights. The overhead of the DIAL controller is negligible in practice since the decision tree building, response time modeling, and LB weights optimization are performed offline, and only need to be leveraged during run time using the monitored metrics. Our evaluation results show that the average increase in CPU utilization of the LB VM under DIAL is about 2%. Of course, if the CPU usage at the LB VM is a concern, we can implement DIAL on a separate VM. In our experiments, we use the HAProxy LB [55]; however, DIAL can also be integrated with the nginx and Apache LBs.

4.5.2 Cloud environments

We set up two cloud environments for our evaluation, an OpenStack based private cloud environment and an AWS-based public cloud environment. Unless specified otherwise, we use the OpenStack environment.

OpenStack-based private cloud: Figure 4.6 depicts our experimental setup. We use an OpenStack Icehouse-based private cloud with several dedicated Dell C6100 physical machines, referred to as *PMs*. Each PM has 2 sockets with 6 cores each, and 48GB memory. The host OS is Ubuntu 14.04. All PMs are connected to a network switch via a 1Gb Ethernet cable. Our ex-

periments reveal that the maximum achievable network bandwidth is about 115 MB/sec (we flood the network using a simple load generator, httperf, and measure the peak observed bandwidth under various request rates and request sizes). Likewise, we find that the maximum achievable memory and (sequential) hard disk drive I/O bandwidths are about 11GB/sec (using RAMspeed) and 50MB/sec (using sysbench), respectively.
AWS-based public cloud: We rent 10 c4.large instances (2 vCPUs and 3.75GB of memory) in AWS EC2's US East (N. Virginia) region. We also rent a c4 dedicated server (PM) for hosting one of the instances colocated with bg VMs.

4.5.3 Evaluation

We first present results for classification and estimation of test workloads. We then present results for performance improvement (reduction in T_{90}) under DIA for OpenStack and AWS setups for CloudSuite and WikiBench. Unless mentioned otherwise, we compare performance under DIAL with performance without DIAL, referred to as *baseline*. In Section 4.5.3 we compare DIAL against existing interference-aware techniques that are popularly employed. We use a metrics monitoring interval length of 10s for the this evaluation. Experimentally, we find that shorter interval lengths can lead to inaccurate classification and estimation due to system noise and load fluctuations [63].

Evaluating detection, classification, and estimation

Detection: The crosses in Figure 4.7 show the impact of different resource contentions, created by microbenchmarks, on CloudSuite's HomePage request class response time under the OpenStack setup. Every data point (cross) in Figure 4.7 is obtained by averaging the 90%ile of response times in every monitoring interval over three different experiments, each of which takes 300s. To detect contention, we use the 95% confidence intervals around the mean (see Section 4.2.1) to obtain the following detection rule for both the OpenStack and AWS setups: $T_{90} > 5ms$, for HomePage; similar rules can be derived for other request classes. We run several experiments using the bg test workloads and find that our detection rule results in a low false positive rate of 5.7%.

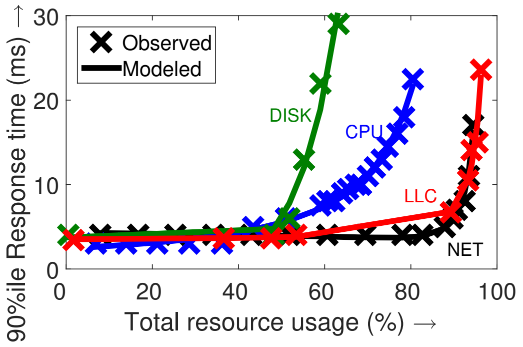


Figure 4.7: Observed and modeled response times for CloudSuite under resource contention via microbenchmarks. Average modeling error: 6.1%.

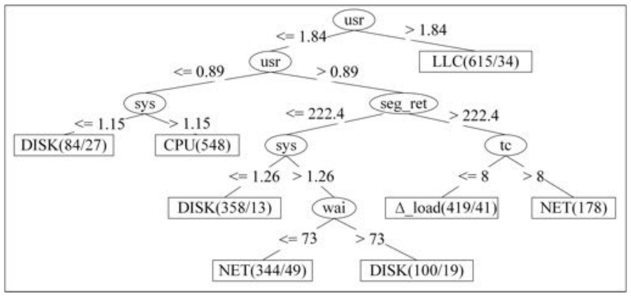


Figure 4.8: Illustration of our trained decision tree created using WEKA. Leaves represent the contention classification. Numbers in the leaves represent the total classification instances (left) and the number of misclassified ones, if any (right).

Classification: We monitor the user space CPU utilization, usr, the kernel space CPU utilization, sys, the I/O wait time, wai, the rate of segments retransmitted, seg_ret, and the 90%ile time taken to establish a connection to the application VM, T_c (via HAProxy logs). Note that all metrics are monitored from within the VMs, to comply with the user-centric design of DIAL. We normalize usr and sys using predicted values to distinguish from workload variations, as discussed in Section 4.2.2. The usr and sys metrics can help detect CPU and LLC contention as the processor might have to do more work under these contentions. wai could potentially help classify DISK contention. Finally, seg and T_c could help classify NET contention because of the reduced available network bandwidth.

Our decision tree for CloudSuite, trained using microbenchmarks, is shown in Figure 4.8. The decision tree is generated using WEKA [54]; in particular, WEKA determines the nodes and cutoff values using the J48 algorithm. The tree structure may be different for different applications. However, we expect the high level rules to be the same, as illustrated by our classification results for the Pinot OLAP application in Section 4.6.3. For example, we expect that LLC interference will lead to an increase in CPU usage.

Our 10-fold cross-validation error is *7.8%*. Our classifier shows that high (normalized to predicted contention) *usr* signals LLC contention, possibly because more work has to be done to service the LLC misses. A high T_c signals NET contention, which seems intuitive. A moderate drop in *usr* and moderate rise in *sys* signals CPU contention; we believe this is because throughput decreases under contention, resulting in lower *usr*, and thus exhibiting a relative rise in *sys*. A high *wai* suggests DISK contention. Finally, a moderate rise in *seg_ret* and T_c signals workload variations (denoted as Δ_load in Figure 4.8).

We also evaluate our classifier using test workloads that were *not* seen during classifier training. We run 50 total experiments using 10 experiments each for Memcache (NET contention), SPEC (CPU contention), Hadoop (DISK contention) and STREAM (LLC contention), in addition to 10 experiments under varying CloudSuite application load. Our decision tree successfully classifies 44 of the 50 test instances; the misclassifications are observed for change in workload and DISK contention. The "misclassifications" for DISK contention (as LLC) under Hadoop are because of the numerous memory accesses made by the colocated Slave VMs; we believe that Hadoop interference cannot always be classified as a single resource due to its complex and dynamic resource needs.

Effect of monitoring interval length: We use a metrics monitoring interval length of 10s for the above evaluation. Experimentally, we find that shorter interval lengths can lead to inaccurate classification and estimation due to system noise and load fluctuations. For example, Figure 4.9 shows the prediction error for CPU metrics used in our decision tree classifier (see Section 4.2.2). We see that an interval of 1s or 5s can lead to high inaccuracy. On the other hand, intervals larger than 10s do not significantly improve accuracy. Likewise, we find a significant increase in the number of false positives for interference detection at smaller interval lengths, leading to cases where DIAL overlooks interference. For these reasons, we choose an interval length of 10s; prior work has also reported such reaction times to

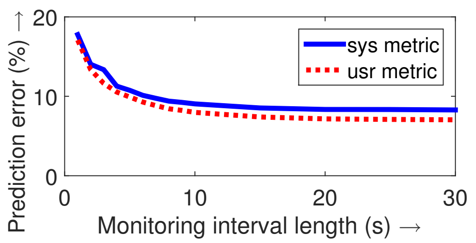


Figure 4.9: Smaller monitoring interval lengths lead to higher inaccuracy.

avoid rash decisions [168, 86, 24].

Evaluating DIAL under real workloads

Figure 4.10 shows our experimental results for CloudSuite under OpenStack for various time-varying contentions created using test workloads in bg VMs. The y-axis shows the tail latency for CloudSuite across all request classes. We create NET, DISK, and LLC contention for apache1 VM using Memcache, Hadoop (TeraSort), and STREAM, respectively. We use SPEC to create CPU contention for apache2.

We see that DIAL significantly reduces tail response times, when compared to the baseline, under all contentions; the reduction ranges from 16% under DISK contention to 59% under LLC contention. The relatively low improvement under DISK contention is because Hadoop intermittently utilizes disk I/O bandwidth; further, not all CloudSuite request classes require (or contend for) disk access.

Without DIAL, the tail response time can be as high as 20-30ms; with DIAL, the tail response time is almost always around 4-5ms. Note that DIAL requires some time (at least two successive intervals of high response time) for interference detection during which response time continues to be high, as seen at the start of each contention.

Figures 4.11 and 4.12 show our classification metrics for apache1 and apache2, respectively; we only show T_c, wai, sys, and usr (and not seg_ret) for ease of presentation. Note that the y-axis range in Figure 4.12 is intentionally smaller to focus on the rise in the sys metric. For apache1, under NET contention, T_c is high while the other metrics are unaffected. For DISK and LLC contentions, sys is high, especially for LLC; further, usr is also high under LLC contention. Finally, the wai metric, though noisy, is higher under DISK contention. By contrast, these metrics are unaffected for the

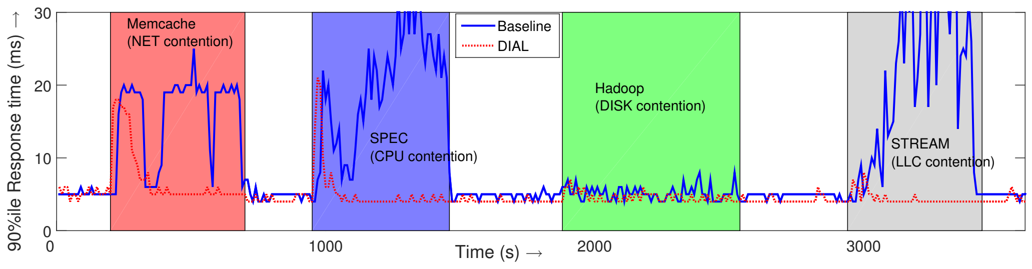


Figure 4.10: Performance comparison between DIAL and baseline for test background workloads. The red, blue, green, and gray regions represent NET, CPU, DISK, and LLC contention, respectively. DIAL reduces 90%ile response time during these contentions by 39.1%, 56.3%, 16.2%, and 59.2%.

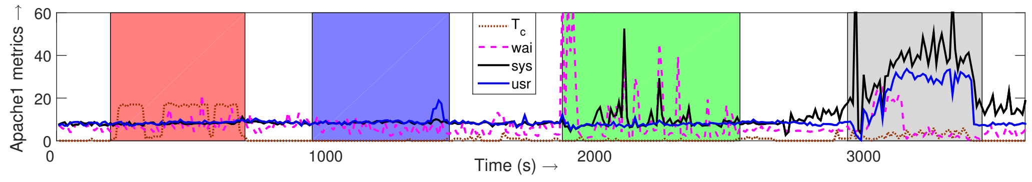


Figure 4.11: T_c, wai, sys, and usr metrics for apache1 application layer VM for the experiments in Figure 4.10. apache1 VM experiences NET, DISK, and LLC contention, and shows an increase in relevant metrics under those contentions.

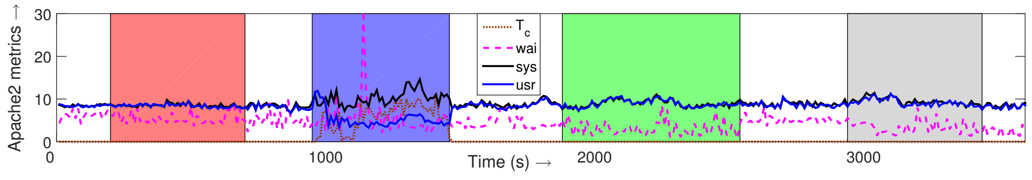


Figure 4.12: T_c, wai, sys, and usr metrics for apache2 application layer VM for the experiments in Figure 4.10. apache2 VM experiences only CPU contention, and consequently shows an increase in relevant metrics under CPU contention.

corresponding time periods under apache2.

Likewise, for apache2, for CPU contention, *sys* is moderately high but not as high as that under DISK and LLC contention under apache1. Again, the metrics are unaffected for the CPU contention period under apache1. This shows that the relevant metrics on the compromised VM change under contention, but are *unaffected for uncompromised VMs*. Further, the change

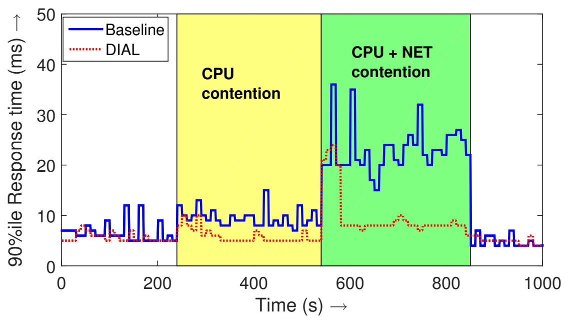


Figure 4.13: DIAL reduces the response time of all request classes by 37% and 56% under CPU and combined CPU + NET contention, respectively.

in metric values under the contention periods are in agreement with the rules of the decision tree classifier in Figure 4.8, even though the classifier was trained on microbenchmarks and not on these test workloads. This highlights the efficacy of our classifier.

While it is possible for several resources to be *simultaneously under contention*, as in the case of Memcache (NET, LLC, and DISK), it is typically the dominant resource that has the greatest impact on performance. In the case of Memcache, the server is hosted on a bg VM and is driven by mutilate clients (on different hosts) issuing a high request rate for a small set of key-value pairs, resulting in NET being the dominant resource. DIAL correctly classifies this Memcache bg VM as creating NET contention. For Hadoop, there is significant demand for disk and memory bandwidth; however, our classifier suggests DISK contention.

Evaluating DIAL under multiple contentions

DIAL is also capable of dynamically responding to multiple compromised VMs. This is because our model allows for arbitrary levels of interference on different VMs simultaneously. The optimization in Section 4.3.1 provides estimates for LB weights, via Eq. (4.3), for all VMs. This is different from the case of multiple resource contentions on the *same* VM, which is beyond the scope of this dissertation.

Figure 4.13 shows our experimental results for CloudSuite where initially apache2 VM is under CPU contention, but then, after about 5 mins, apache1 (on a different host) also starts experiencing NET contention, resulting in

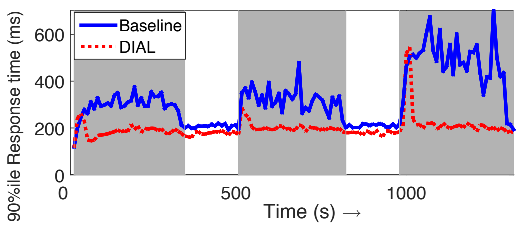


Figure 4.14: Performance under LLC contention for fg WikiBench. DIAL reduces response times by ∼23.6% during contention (gray regions).

very high interference for the application. After an additional 5 mins, both contentions are terminated. We see that DIAL substantially reduces T_{90} under interference. This example highlights the dynamic nature of DIAL. Compared to existing techniques that employ (static) VM placement to mitigate interference, DIAL is able to adapt to *variations in interference* by constantly updating its estimates and re-distributing load accordingly. For the above experiment, for CPU contention, the DIAL weights are $\{0.45, 0.1, 0.45\}$ (apache2 under contention), and for combined CPU and NET contention, the weights are $\{0, 0.27, 0.73\}$ (apache1 under severe NET contention). Note that if several VMs are under severe contention, we may have to scale out to maintain acceptable response times.

Evaluating DIAL for the WikiBench fg application

Figure 4.14 shows our results for WikiBench under LLC contention created by the dcopy microbenchmark. Here, we have two application VMs and one of them is under contention. The figure shows the response time for baseline and DIAL for all request classes. We create three different contention levels for this experiment, shown in gray. DIAL reduces response time by about 23% when compared to the baseline. We also measure the *usr* and *sys* metrics for classification and find that both increase considerably, by about 62% and 41%, respectively, under interference; this is in agreement with our decision tree classifier.

Evaluating DIAL in the AWS setup

Figure 4.15 shows our results for CloudSuite under LLC contention created by the dcopy microbenchmark in the AWS setup. Here, we have 10 application

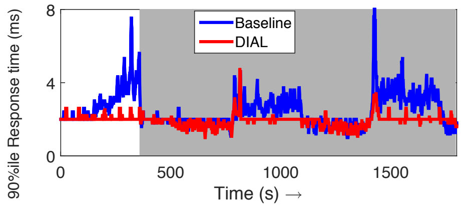


Figure 4.15: Performance under LLC contention for AWS setup. DIAL reduces response times by around 22.3%.

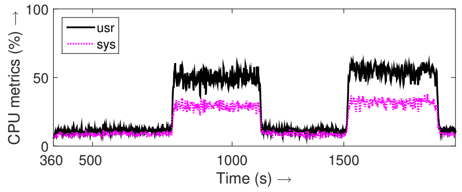


Figure 4.16: *usr* and *sys* metrics for the gray region in Figure 4.15. These metrics clearly increase during contention.

VMs and only one of them is under contention. The figure shows the response time for baseline and DIAL for all request classes served by all VMs in the AWS setup. We create several different contention levels for this experiment. We see that DIAL reduces response time by about 22% when compared to the baseline. This shows that *even one* compromised VM (out of 10) can considerably impact the overall response time.

Figure 4.16 shows the *usr* and *sys* metrics for the shaded region in Figure 4.15 to assess classification. Clearly, both the *usr* and *sys* metrics increase considerably during contention when compared to the low, flat lines during no contention. Further, the regions of contention can be easily discerned from the figure, resulting in good detection accuracy.

Comparison with existing user-centric techniques

We now compare DIAL with existing user-centric interference mitigation strategies. We do not consider cluster management strategies, such as CloudScope [14], and Tarcil [32], since these cannot be implemented by the cloud user who does not have a global view of the infrastructure.

Utilization-based strategies. Figure 4.17 shows our experimental results for high CPU contention under DIAL and under ICE [84]. Similar to DIAL,

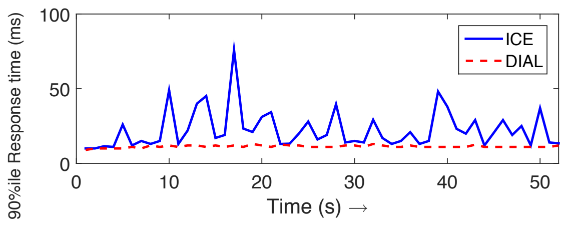
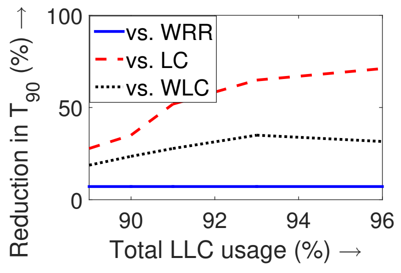
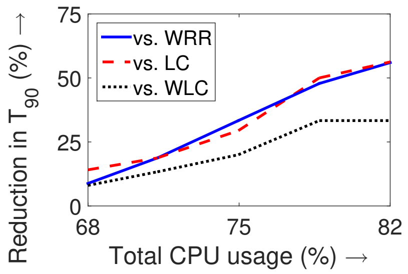


Figure 4.17: Comparison of DIAL with ICE under CPU contention. DIAL reduces T_{90} by 25-48% for all request classes.



(a) TagSearch for LLC contention.



(b) TagSearch for CPU contention.

Figure 4.18: Comparison of DIAL with other LB heuristics.

ICE is an interference-aware load balancing strategy that adjusts the traffic directed towards compromised VMs. However, instead of using LB weights, ICE ensures that the CPU utilization for the compromised VMs stays below a certain threshold. The authors do not mention this threshold value in the paper, and so we experimentally determine the best threshold value across experiments. Unfortunately, we find that the optimal CPU utilization threshold varies with the amount and type of interference. For example, we find that 15% CPU utilization works well for moderate CPU interference under ICE, but does not work well for high CPU interference, as shown in Figure 4.17. Under DIAL, with theoretically optimal weights, response time is significantly lower, and the observed CPU usage at the compromised VM is about 8-10%; results are similar for other contentions. While interference-aware thresholds for ICE can help, this will require a relationship between threshold, type, and amount of interference. Since ICE does not estimate interference, the threshold is static.

Queue-length based strategies. Queue-length or load-based strategies send traffic to the VM that has the lowest load. In particular, we consider the Least Connections (LC) strategy that directs the next incoming request to the application VM that has least number of active connections; Amazon leverages LC for its elastic load balancer [5] for this purpose. Under interference, the outstanding requests for the compromised VM will be higher, resulting in fewer additional requests being sent to it under LC.

Figure 4.18 shows the reduction in T_{90} afforded by DIAL over LC (and other heuristics that we discuss next) for the TagSearch request class under CPU and LLC contentions; results are similar for other classes and for NET and DISK contention. We see that DIAL lowers response time significantly, by as much as 70-80%, when compared to LC (red dashed line). The improvement is greater at higher contentions. The reason for this improvement is that the compromised VM does not just have lower capacity, but also requires (non-linearly) *more time* to serve each request. The weights under DIAL take both these into consideration, as opposed to LC that only addresses the former.

Weighted load balancing strategies. We now compare DIAL with other weighted load balancing heuristics, such as Weighted Round Robin (WRR) and Weighted Least Connections (WLC). For WRR and WLC, we use proportional interference-aware weights, as discussed in Section 4.3. Figure 4.18 shows the reduction in T_{90} afforded by DIAL over WRR (blue solid line) and WLC (black dotted line). We see that DIAL lowers response time considerably when compared to these heuristics. It is interesting to note that WRR is typically worse than WLC under CPU contention, but better than WLC under LLC contention; this observation reaffirms the fact that the impact of interference depends on the type of resource under contention.

4.6 DIAL for Pinot

We now present our implementation of DIAL and its evaluation for a widely used OLAP solution, Pinot [66].

4.6.1 DIAL Implementation

The Load Balancing Tier (LBT) for Pinot consists of the Broker nodes (see Section 4.4.1), that distribute queries to the back-end workers nodes. The Brokers rely on routing tables, stored in Broker memory, to determine which worker nodes host the data segments that are needed to serve the incoming query. Each routing table is a map from every segment to one worker node; since each segment is stored on several replicas, numerous unique routing tables can be generated. Depending on the size of the cluster, Pinot creates a fixed number of routing tables; the tables are dynamically updated when new segments are uploaded or when existing segments expire, as reported by the Controller. By randomly selecting a routing table for each incoming query, the Brokers balance load among the worker nodes.

We implement DIAL on the Broker using ∼300 lines of Java code. Once interference is detected, DIAL updates the routing tables to remap segments that were initially assigned to the worker(s) under interference to other replicas, based on the theoretically-derived optimal fractions, q^* (see Section 4.3.2). Likewise, once interference ceases, the routing tables are updated to the default balanced weights.

4.6.2 Cloud Environment

We use several blade servers from a HP Proliant C7000 Chassis, referred to as PMs (Physical Machines). Each server has 2 processor sockets with 4-core CPUs (8 hardware threads) each, and 32 GB memory.The host OS is Ubuntu 16.04. The servers are connected through 1Gb/s network links. We use KVM (on top of Ubuntu 16.04) to deploy VMs on these PMs.

We deploy 6 Pinot worker nodes on 1 vCPU, 16GB memory VMs; each VM is on a separate PM. We deploy the Pinot Controller and 2 Pinot Brokers using VMs with 8 vCPUs and 16GB memory, on different PMs.

We experiment with CPU and LLC contention for Pinot. For CPU contention, we use a 1 vCPU bg VM that is statically pinned to the same core as the fg VM (via hyper-threading). For LLC contention, we use a 3 vCPU bg VM that is pinned to the remaining 3 cores of the 4-core socket that hosts the fg VM; in this way, we do not share the same core as the fg VM to avoid CPU contention.

4.6.3 Evaluation

We use the Pinot benchmark explained with details in Section 2.2.1 . We use a warm-up time of 120s for all our experiments in this section. We focus on 95%ile response times for Pinot.

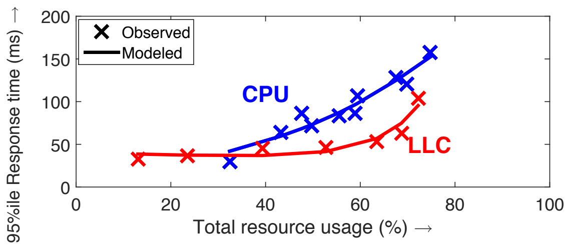


Figure 4.19: Observed and modeled response times for Pinot under resource contention via microbenchmarks. Average modeling error is 11.5%.

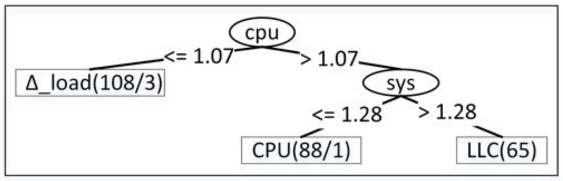


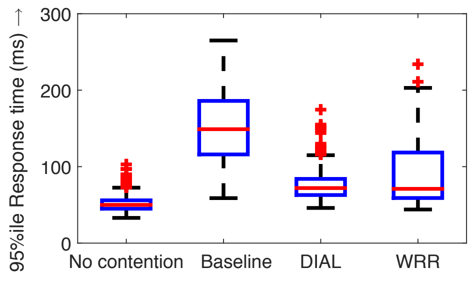
Figure 4.20: Our trained decision tree classifier for Pinot. Numbers in the leaves represent the total classification instances (left) and the number of misclassified ones, if any (right).

Evaluating detection, classification, and estimation

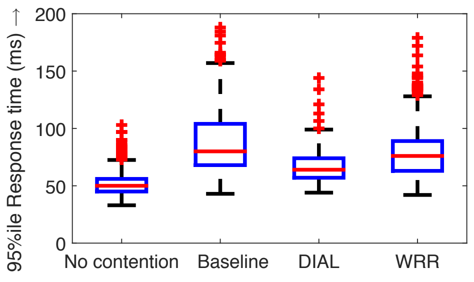
Detection: The crosses in Figure 4.19 show the impact of CPU and LLC contention on Pinot response times. Every data point (cross) in Figure 4.19 is obtained by averaging the 95%ile of response times in every monitoring interval over three different experiments, each of which takes 300s. The detection rule of $T_{95} > 61ms$ is obtained based on the discussion in Section 4.2.1. We run several experiments using the bg test workloads and find that our detection rule results in a low false positive rate of 3.3%.

Classification: We monitor total CPU usage, *cpu*, and the system space CPU utilization, *sys*; we normalize these values using predicted values to distinguish from workload variations, as discussed in Section 4.2.2. Our decision tree for Pinot is shown in Figure 4.20. Our 10-fold cross-validation error is 2.3%. The classification rules in Figure 4.20 closely resemble those for the web application in Figure 4.8.

We now evaluate our classifier using test workloads that were *not* seen during training. We run 10 experiments each for SPEC (CPU contention) and STREAM (LLC contention), and 10 experiments under varying Pinot



(a) CPU contention



(b) LLC contention

Figure 4.21: Comparison of DIAL with other heuristics for Pinot.

workload. Our decision tree classifier is able to accurately classify all instances, except one CPU interference instance which is misclassified as LLC interference. Our classification accuracy based on these 30 experiments is 96.7%.

Estimation: The solid lines in Figure 4.19 show our modeling results for Pinot interference estimation (as discussed in Section 4.2.3) under different resource contentions via training. Our average modeling error across all contentions is *11.5%*.

Evaluating DIAL for Pinot under real bg workloads

Figure 4.21 shows our experimental results for Pinot under our KVM setup for CPU and LLC contentions created using test workloads SPEC and STREAM, respectively. The contention is created in bg VMs on one of the six PMs hosting the Pinot worker VMs. We show the tail response time values for no contention, baseline (with contention), DIAL, using theoretically optimal interference-aware weights from Section 4.3.2, and Weighted Round Robin (WRR), which uses proportional interference-aware weights, as discussed in Section 4.3. The response time is the query completion time monitored at the Broker, and thus depends on the performance of all Pinot workers.

We see that DIAL significantly improves tail response times when compared to baseline; the average reduction in 95%ile response times for CPU and LLC contention is 40.5% and 25.8%, respectively. Compared to WRR, DIAL provides some improvement; the average reduction in 95%ile response times for CPU and LLC contention is 16.1% and 16.5%, respectively. Thus, we conclude that DIAL's interference-aware load balancing can help improve

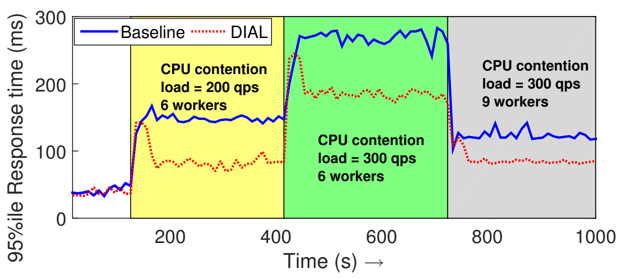


Figure 4.22: Performance of DIAL and baseline using Pinot under CPU contention and under dynamic workload and cloud conditions.

performance for OLAP applications like Pinot, in addition to web applications like CloudSuite and WikiBench.

Evaluating DIAL for Pinot under dynamic conditions

We now evaluate the performance of DIAL under dynamic variations in request rate, and in response to a scale-out. Note that our focus here is on the performance of DIAL, and not on the specifics of the scale-out policy itself; for this experiment, we assume that the scale-out policy reacts to the change in request rate and provisions the required additional number of workers.

Figure 4.22 shows the 95%ile response time (tail latency) for Pinot under DIAL and baseline for our dynamic workload experiment. Here, we start with a load of 200 queries/s (or, qps) and no interference; as before, we have 6 Pinot workers and a replication factor of 3. Then, in the next phase (yellow shaded region), one of the fg worker VMs experiences CPU interference due to a colocated VM running SPEC. DIAL responds, after monitoring and detection, by setting the theoretically optimal load balancing weights for the 3 replicas of segments hosted on the under-interference worker (see Section 4.6.1). For this experiment, the theoretically optimal weights in this phase are $\{0.16, 0.42, 0.42\}$, obtained via the analysis discussed in Section 4.3.2. By setting these weights, the tail latency lowers from about 147ms under the baseline to 88ms (39.9% improvement).

In the next phase (green shaded region), Pinot experiences an increase in load to 300 qps, severely impacting tail latency. DIAL detects this load change via request rate monitoring (see Section 4.3.3), and updates the load balancing weights to $\{0.2, 0.4, 0.4\}$, thus lowering tail latency from 266ms under the baseline to 189ms (28.2% improvement). Our theoretically derived weights from Section 4.3.2 already take request rate into account (via the a parameter), and thus the updated weights can be easily obtained.

To handle the increased load, Pinot eventually scales-out by adding 3 new workers and redistributing data segments across all workers. We assume that the scale-out and data segment mapping is handled by an external autoscaling entity, which is not the focus of our work. With the additional workers, the tail latency of Pinot decreases, as seen in the last phase (gray shaded region). DIAL again updates the weights for this new configuration, by updating the n parameter (that represents the number of workers), resulting in a further lowering of tail latency from about 123ms under the baseline to 87ms (28.8% improvement).

We repeated the experiments for a total of 5 runs. The results were qualitatively similar to Figure 4.22, with the average improvement in 95%ile response time across all runs afforded by DIAL in the three shaded phases being about 33.1%, 29.8%, and 30.7%. We also repeated the experiment with LLC contention, and obtained qualitatively similar results with an average improvement of up to 20%. This shows that DIAL can dynamically respond to changes in request rate and the number of workers by updating its load balancing weights.

4.7 Conclusion

This chapter addresses the request scheduling problem for load-balanced applications with the goal of having minimum tail latency where the applications are running on a cluster of VMs facing unpredictable performance.

We presented DIAL, a user-centric Dynamic Interference-Aware Load balancing framework that can be employed directly by cloud users without requiring any assistance from the hypervisor or cloud provider to reduce tail response times during interference. DIAL works by leveraging two critical components: (i) an accurate, user-centric response time-monitoring based interference detector, classifier, and estimator, and (ii) a framework for deriving theoretically optimal load balancer weights under interference. We use analytical tools for both components resulting in a rigorous and generic methodology that can be extended to other scenarios. Our experimental results for web and OLAP applications on several cloud platforms, under interference from realistic benchmarks, demonstrate the benefits of DIAL.

Chapter 5

Application-Agnostic Batch Workload Scheduling

Resource under-utilization is common in cloud data centers. Running batch workloads in the background is a common practice to improve server utilization in cloud data centers. However, cloud user (foreground) application performance can severely be impacted due to the resource contention created by background workloads. Therefore, we have a batch workload scheduling problem with two competing goals: (1) foreground workloads' SLO is not violated, and (2) background workloads' progress rate is maximized; the progress rate is directly correlated with resource utilization.

Despite the considerable work in this area, a significant challenge that has not been adequately addressed is considering the foreground workloads as a black-box. This consideration is critical since cloud providers are not aware of cloud tenants' workloads and their dynamics. We present Scavenger, a batch workload scheduler that opportunistically runs containerized batch jobs next to tenants' Virtual Machines (VMs) to improve utilization. Scavenger dynamically regulates the resource usage of batch jobs, including processor usage, memory capacity, and network bandwidth, to ensure that the tenants VMs' resource demand is met at all times. We experimentally evaluate Scavenger and show that it increases resource usage without compromising on the resource demand of customer VMs. Importantly, Scavenger does so without requiring any offline profiling or prior information about the tenant workloads.

This study is under review for SOCC 2019 conference. We introduce the problem and scope of this chapter in Section 5.1. While we discussed the prior works in Section 3.2, in Section 5.2, we highlight some of the prior works to put Scavenger in context. We illustrate Scavenger design principles and evaluation methodology in Section 5.3 and Section 5.4, restrictively. Finally, we provide the evaluation results in Section 5.5 and show that Scavenger can realize the outlined goals.

5.1 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 [4] or Google [51]) on multiple servers in a data center.

Servers in cloud data centers often experience low resource utilization [31, 158]. A study focused on Amazon EC2 observed that cloud server usage is often below 10% [81]. 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% [22] (see Section 2.4.2).

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 [170, 82, 50]. 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* [63]. Worse, this interference is *dynamic* due to resource demand variations in tenant and batch workloads [45, 169].

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 [87] or shared clouds at Universities, more aggressive resource management can be employed to improve utilization.

While there has been prior work on background workload management (see Section 3.2), 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 2.4.3) 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 [83].

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 [169, 22], or (ii) benchmark tenant VM performance to carefully colocate background workloads [30, 31], or (iii) regulate the resource usage of background workloads to avoid SLO violations for the foreground VMs [13, 82, 60]. Such solutions are ineffective and, at times, infeasible in cloud environments since tenants do not expect their VMs to be instrumented [99], and are not required to share their performance SLOs with the provider [42]. 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 [63].
2. *Need for a dynamic and tunable solution.* Another class of solutions focuses on careful VM placement to avoid interference in the first place [141]. 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 [96, 32]. 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 [170, 153, 60]. Given that, for realistic workloads, several resources may simultaneously be under contention, such resource-specific solutions are inadequate [63].

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 [140]) 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 [141].

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 stop 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 [39] and TailBench [67], 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%.

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.

5.2 Novelty of Scavenger in the Context of Prior Wor

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 tradeoff 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 [153, 60, 80, 136, 127] and at least 1 in 2019 [13]. While we discuss related work in detail in Section 3.2, 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 [82], Borg [141], and Bistro [50], or where the foreground workload's performance can be monitored by the provider, e.g., PARTIES [13]. 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 [22], Zhang et al. [169], and TR-Spark [158], or *can be accurately profiled*, e.g., Paragon [30] and Cuanta [52]. 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 [170]), LLC (e.g., dCat [153]), or network (e.g., QJUMP [53]).

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 their VMs may have unpredictable resource usage [63]; thus, the tenant's workload and performance SLOs are *unobservable* [99]. 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 3.2) that either assume the tenant is a white box or require a one-time profiling of the tenant (e.g., PerfIso [60]), Scavenger is truly black-box, or application-agnostic, in nature.

5.3 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 dissertation 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 [128] or Spark [163] 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 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, batch jobs within the containers can be killed to immediately release resources for foreground VMs. The monitoring and resource regulation is managed via Scavenger daemons that run on each cloud PM, thus making Scavenger distributed in nature. In this dissertation, we consider contention for processor (including CPU and last-level cache (LLC)), memory capacity, and network bandwidth resources.

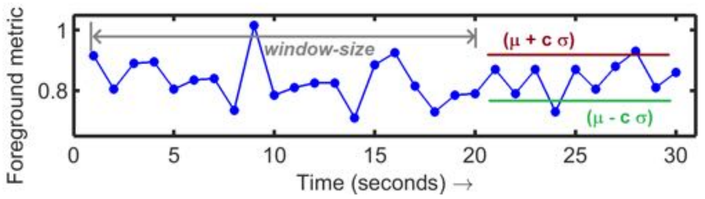


Figure 5.1: *Illustration of Scavenger's generic algorithm.*

5.3.1 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 [141, 153, 52].

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 [149], 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 [62], 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).

Figure 5.1 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 instance, Scavenger can be more conservative in public clouds, but can be aggressive in private clouds.

5.3.2 Mitigating memory capacity contention

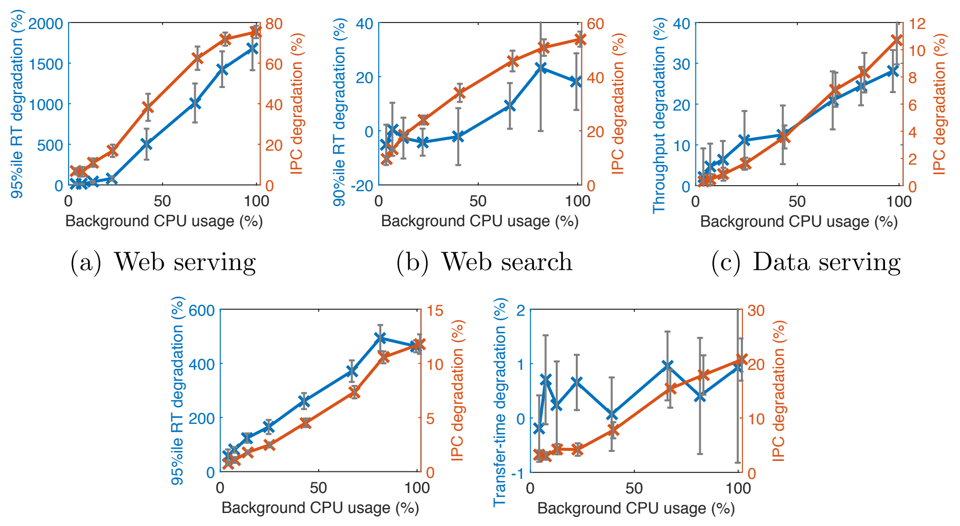
We closely follow the generic algorithm from Section 5.3.1 for regulating the memory allocation of the background jobs and use the per-second memory usage of the foreground VM as the monitored metric. Based on the initial *window-size* seconds of observation, we compute the sample mean and sample standard 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 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 value (that caused the violation). On the other hand, if the foreground memory usage goes below $\mu - c \cdot \sigma$ for w (*window-size*) 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 5.5.1.

5.3.3 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,



(a) Web serving (b) Web search (c) Data serving

(d) Data caching (e) Media streaming

Figure 5.2: *Impact of background LLC workload on CloudSuite performance (left y-axis) and its IPC (right y-axis).*

we use Linux's traffic control mechanism [125]. In particular, we use the token bucket filter to enforce bandwidth limits on the background jobs' egress traffic; we do not impose any limits on the foreground workload traffic.

5.3.4 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) [98]. In order to target generic processors, we do not assume access to CAT. We discuss how Scavenger can be integrated with CAT in Section 5.6.

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 [153]. We find that this is not always the case.

We experimented with the SPEC CPU benchmark suite colocated with dCopy [25] (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 [35, 91, 112, 117, 166]. Some recent works have also used IPC and related metrics as a proxy for cloud workload performance [167, 86]. 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 5.4.3) and run the dCopy LLC microbenchmark [25] on a container using the other three cores; see Section 5.4.2 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 5.2 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 [60], 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 5.3.1, 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 5.5.1.

5.3.5 Mitigating CPU core contention

As noted in prior work, sharing of CPU cores between foreground and background jobs can result in unpredictable contention [75, 73]. 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. In particular, with CPU core sharing between foreground and background, tail latency increased by about 900%, on average, for the latency-critical CloudSuite workloads, for a negligible increase (less than 10%) in CPU utilization.

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.

5.4 Evaluation Methodology

This section describes the evaluation methodology we employ for the performance evaluation results presented in Section 5.5. We start by detailing our Scavenger prototype implementation, followed by our experimental setup and the workloads we employ for evaluating Scavenger.

5.4.1 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 5.3.2 – 5.3.5 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 [78]) 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 [125] 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.

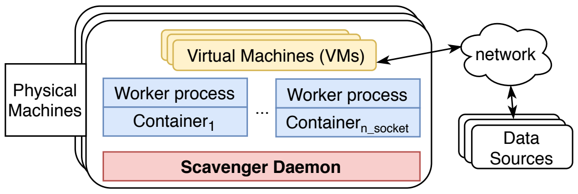


Figure 5.3: *Illustration of our Scavenger deployment.*

Deployed architecture: Figure 5.3 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 on each PM in Figure 5.3 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 5.4.4). 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.

5.4.2 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 [18] (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.

5.4.3 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 [39]**: We use the latest version, CloudSuite 3.0 [106] explained in Section 2.2.1.
- **TailBench [67].** TailBench is a recent benchmark suite specifically designed for analyzing latency-critical applications. Section 2.2.1 overviews TailBench briefly.

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).

5.4.4 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* [25] copies vectors repeatedly to stress the cache; (ii) *stress-ng* [123] is a cpu stress benchmark; and (iii) *iperf* [130] is a network bandwidth measurement tool that we employ to stress the network.

Spark jobs. Spark [163] 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 [119] as the storage core. We also employ Yarn [140] (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 [47] and Spark-Bench [120], such as FFT, KMeans, Sorting, etc.

5.5 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.

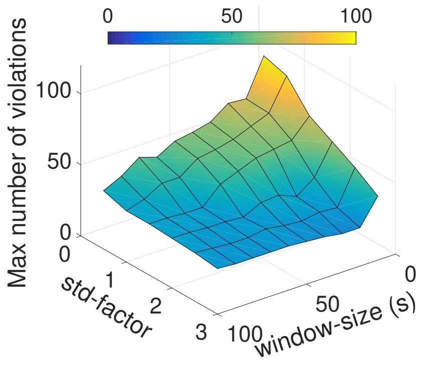
5.5.1 Sensitivity analysis

We use sensitivity analysis to determine the parameter values to be used for the resource regulation algorithms from Section 5.3.2 – 5.3.4; note that the CPU cores regulation algorithm from Section 5.3.5 has no tunable parameters. Our analysis must be black-box and should not involve workloads that will serve as foreground in the evaluation.

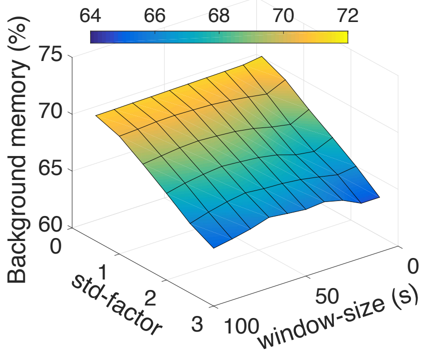
Memory regulation algorithm sensitivity analysis. To determine the right values for the *window-size* and *std-factor* parameters of our memory regulation algorithm from Section 5.3.2, we require a black-box approach that does not involve the foreground workload. We resort to simulations for sensitivity analysis and use the recent Alibaba traces [1] containing foreground memory usage, sampled every 10s, for about 4,000 servers for 8 days.

Figure 5.4 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 5.3.2. 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. Note that with these parameter values, we afford about 68.3% background memory usage. By contrast, the Alibaba traces show an average background memory usage of 60.7%.

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 [144] for our sensitivity analysis. Our analysis suggests that *std-factor* = 2 and *window-size* = 30s



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



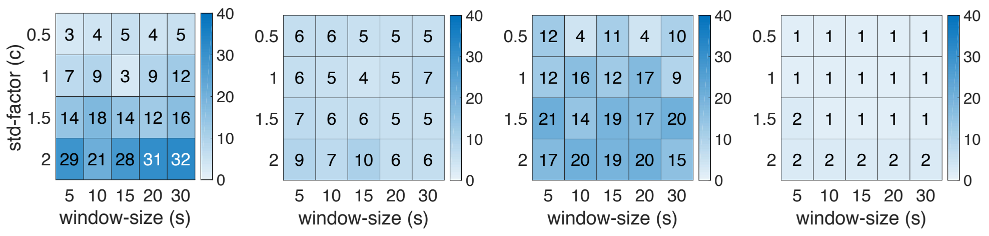
(b) Background memory afforded (higher is better).

Figure 5.4: *Sensitivity analysis for std-factor and window-size parameters of the memory regulation algorithm.*

work well.

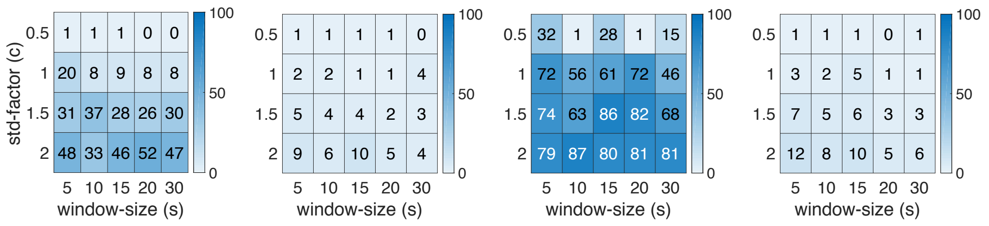
Processor cache regulation algorithm sensitivity analysis. Employing the same trace-driven approach as above for cache regulation algorithm is infeasible as we require information on how the foreground IPC will degrade under different algorithm parameters. Instead, we conduct actual experiments using 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 subsections; instead, we will use TailBench, which is not employed for sensitivity analysis.

There are four parameters for cache regulation algorithm (see Section 5.3.4): *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 [105], 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



(a) Web serving (b) Web search (c) Data serving (d) Data caching

Figure 5.5: *Degradation of foreground IPC (lower is better) colocated with dCopy under processor regulation.*

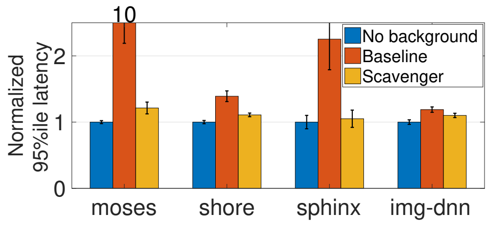


(a) Web serving (b) Web search (c) Data serving (d) Data caching

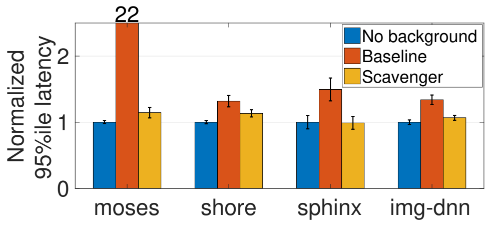
Figure 5.6: *Background CPU usage afforded (higher is better) under the processor regulation algorithm.*

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 5.5 and 5.6, 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 5.3.4, 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



(a) SparkPi in the background.



(b) KMeans in the background.

Figure 5.7: *Performance degradation of individual TailBench workloads in Lab testbed colocated with Spark.*

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.

5.5.2 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 [171] or Bistro [50] since they require SLO and latency monitoring of the foreground workload.

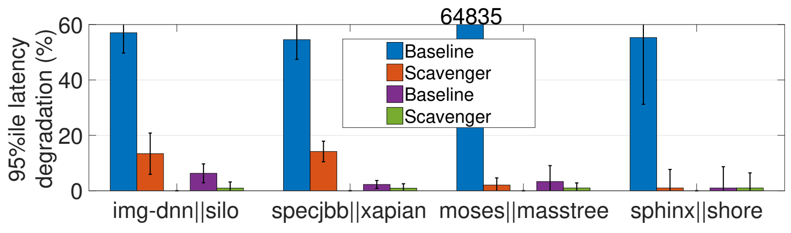
TailBench workloads as foreground. We start with the case of TailBench workloads in the foreground. We perform experiments on both testbeds. For the Lab testbed, we run a TailBench workload on a 1-vCPU VM and use the remaining 3 cores (via cpuset) to launch Spark containers; this 1:3 core allocation represents the case of heavy background usage. Figure 5.7 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.

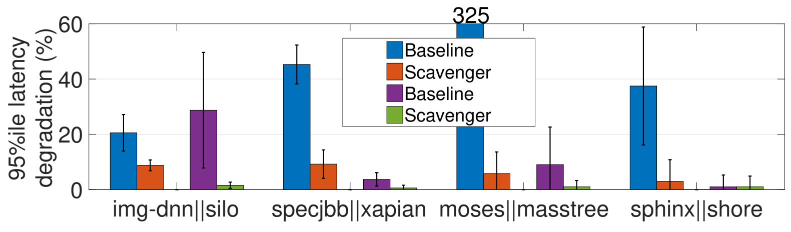
In terms of utilization, Scavenger increases average CPU usage across all workloads, compared to no background, by about 170% and 198%, respectively, when using SparkPi and KMeans in the background. Likewise, the memory usage increases by 142% and 230%, respectively. The highest gains in CPU usage, of 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 5.5.3.

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 5.8 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 re-



(a) SparkPi in the background.



(b) KMeans in the background.

Figure 5.8: *Performance degradation of multiple TailBench workloads in Cloud testbed colocated with Spark.*

duces 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 5.8; 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 5.7; *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.

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.

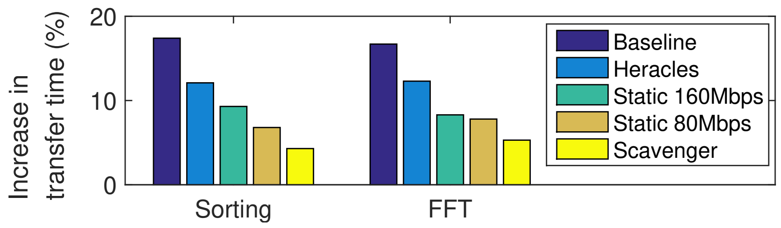


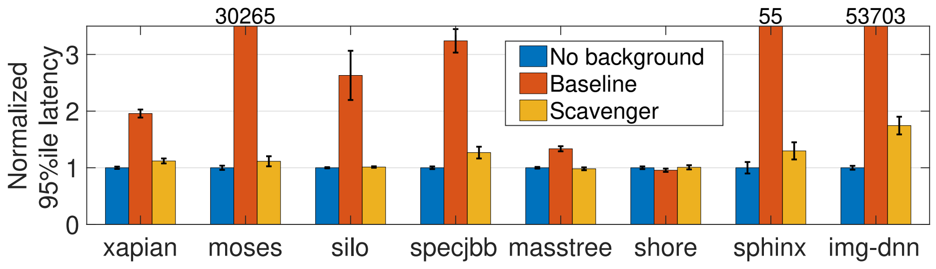
Figure 5.9: *Performance degradation of Media streaming (in Lab testbed) when colocated with Spark jobs.*

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.

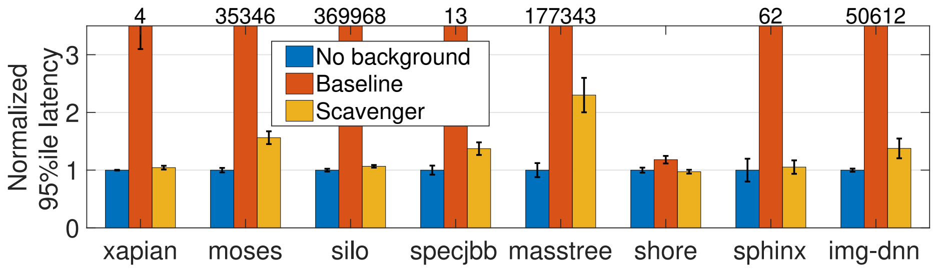
Figure 5.9 shows our results, averaged over 3 runs, for different background jobs under network regulation. We show results for baseline (no regulation), Heracles network regulation, static background limits (via TC [125]), and Scavenger network regulation; for Heracles, we implement the regulation algorithm from the paper [82], 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



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



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

Figure 5.10: *Performance degradation of foreground TailBench, colocated with dCopy, under processor regulation.*

Spark. Given its dynamic nature, Scavenger outperforms static approaches while affording higher background usage.

5.5.3 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.

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

Figure 5.10(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). 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 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 also repeated the above set of experiments by replacing dCopy with the CPU-intensive stress-ng microbenchmark [123] 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 5.10(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. 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) likely because the application times out under the high cache contention. 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.

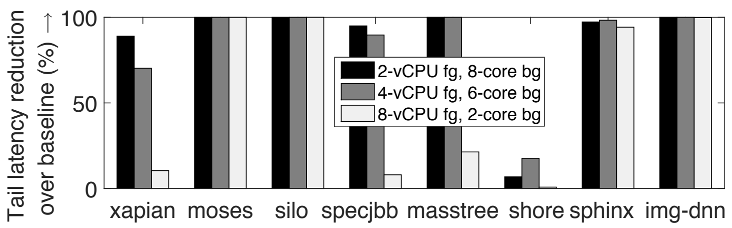


Figure 5.11: *Scavenger's latency reduction over baseline for different foreground (fg) and background (bg) sizes.*

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 5.11 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 5.10(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 5.12 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.

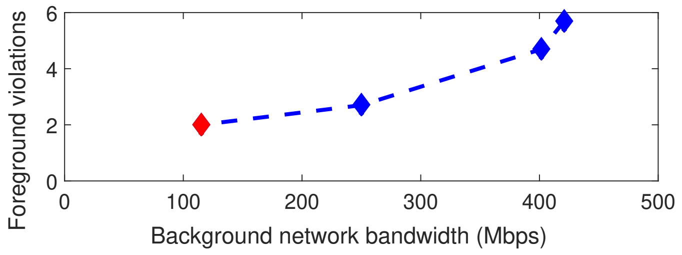


Figure 5.12: *Trade-off between foreground violations and background bandwidth for different* std-factor *settings. The default* std-factor *setting of 2 is shown in red.*

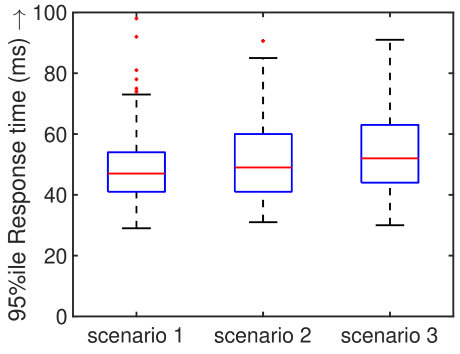
5.5.4 Scavenger along with DIAL

It is important to see how both Scavenger and DIAL can be deployed simultaneously in cloud environments. DIAL provides theoretically-optimal load-balancing weights for cloud-deployed load-balanced applications where worker VMs can have variable resource capacity and Scavenger tries to ensure that tenant workloads' performance is not impacted by the background workloads. It is therefore interesting to see how these two approaches work together.

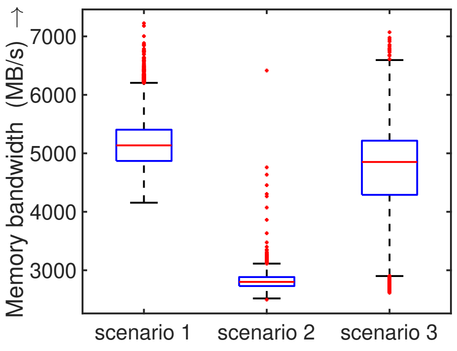
We address the outlined question experimentally. In our setup, there are two foreground VMs and one background container (in Lab testbed). The first foreground VM has one vCPU and 16 GB memory, and runs Pinot which is a load-balanced application (see Section 4.6.2). The second foreground VM has 2 vCPUs and 16 GB memory and runs Stream, continuously measuring the memory bandwidth. The background container has 1 CPU core and runs DCopy continuously creating LLC pressure. Every experiment starts with 4 minutes of warm-up period and continues with 10 minutes of steady-state. We report 95%ile response time for Pinot and memory bandwidth for Stream. We define three scenarios and run each of them 10 times:

1. Scenario 1: The two foreground VMs run their assigned workloads and DIAL is enabled for Pinot. The background container does not run DCopy and Scavenger is disabled.

2. Scenario 2: In addition to the two foreground VMs running their assigned workloads, the background container runs DCopy but Scavenger is still disabled.



(a) Pinot tail latency.



(b) Stream memory bandwidth.

Figure 5.13: *Comparison of foreground workloads' performance metrics in three different scenarios. DIAL is enabled in all the three scenarios. Scavenger is disabled in Scenario 1 and 2, and is enabled in Scenario 3.*

3. Scenario 3: Both foreground workloads and the background workload are running, and Scavenger is enabled.

Figure 5.13 compares Pinot tail latency and Stream memory bandwidth for the three scenarios. DIAL is enabled in Scenario 1 and the impact of 2-vCPU VM running Stream on Pinot latency is negligible. In Scenario 2, where Scavenger is disabled, 95%ile response time of Pinot and Stream average memory bandwidth degrade by 5.5% and 45.4%, respectively. We then enable Scavenger in Scenario 3 and while there is still some degradation for Pinot tail latency (12%), Stream average memory degradation is only 11%, which is much less than Scenario 2. Furthermore, Scavenger improved the average CPU and memory utilization by 10.3% and 33.5%, respectively. These results show that Scavenger and DIAL can work together to improve utilization while balancing the impact on foreground VM performance.

5.6 Discussion

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. Having a tunable algorithm is necessary when deploying colocation solutions on different environments, such as public cloud (high performance isolation needs) versus private cloud (a balance between performance isolation and resource usage).

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 5.3.4, 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.

5.7 Conclusion

This chapter 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.

Chapter 6

Efficient Segment Assignment Strategy for OLAP Systems

In this chapter, we address the data segment scheduling problem in Online Analytical Processing (OLAP) platforms. The problem is challenging due to the varying workload demand and data popularity that results in resource contention and load imbalance. OLAP systems typically split a big table to several data segments and distribute these data segments among a cluster of worker nodes. To serve a query, every worker node runs the query on its' assigned data segments, and then these local results are integrated to compute the final response. To prevent hotspots and high tail latency, the load induced by queries targeting a data segment should be taken into account for deciding which worker nodes will host the data segment. The induced load by a data segment is dynamic and could be different for various tables.

We present EASY, an efficient segment assignment strategy that leverages analytical modeling to predict the future load induced by data segments, thus allowing for long-term balancing of load across worker nodes. Our implementation and evaluation of EASY on Pinot shows that we can significantly reduce query tail latencies in the presence of dynamic load.

This study has been published in ICDCS 2018 [65]. We introduce the problem and discuss the scope of this chapter in Section 6.1. We then illustrate the solution architecture of EASY in 6.2. A load-aware cost function is the core design principle of EASY which we discuss it in Section 6.3. Apart from the related works we discussed in Chapter 3, we provide an overview of prior work in the context of EASY in Section 6.4. Finally, we evaluate EASY and present the evaluation results in Section 6.5.

6.1 Introduction

Large-scale and real-time Online Analytical Processing (OLAP) is a significant requirement for customer-facing companies. A popular distributed near-realtime OLAP solution is Pinot [108], that is extensively used at LinkedIn and Uber for serving user queries (such as the Profile View functionality of LinkedIn) and for internal analysis.

Pinot leverages a simple architecture (see Section 2.2.1) where every table is divided into data ”segments” distributed among worker nodes. Every segment typically contains information for a period of time (e.g., one hour or one day). An incoming query from a client to Pinot is run simultaneously across workers hosting the target segments. The end-to-end response time of a query in Pinot depends on the longest query latency among target workers, as all individual (per-worker) results need to be integrated by the broker node(s) before sending the response back to the client.

In such distributed data store systems, the Segment Assignment Strategy (SAS) has a significant impact on query latency. SAS dictates the placement of new segments on worker nodes; new segments are created dynamically as time passes. In other words, SAS is a scheduling problem in which a subset of worker nodes should be selected to host the newly created data segment and serve the queries targeting this segment. Naive SAS such as round-robin can result in hotspots, severely impacting query tail latencies (see Section 6.5).

Existing SAS in production systems often employ a decentralized and scalable utility function (or cost function) approach whereby each server is assigned a cost that can be easily computed; incoming segments are then assigned to the lowest cost server, whose cost is then updated. While popular open source OLAP solutions such as Pinot and Druid [159] have their own cost-based SAS, these default strategies have their shortcomings. The Pinot SAS aims to balance the number of segments across workers. Our experimental results show that this SAS leads to unbalanced load and high tail latencies. Druid implements a more advanced SAS by taking the time range of segments into account. However, as we show in Section 6.5, there is much scope for improvement, especially when there are multiple tables in the data store.

We propose a new load-aware SAS, EASY (Efficient segment Assignment StrategY) [65], that outperforms existing SAS solutions regarding load distribution among workers and, importantly, regarding query tail latency. EASY

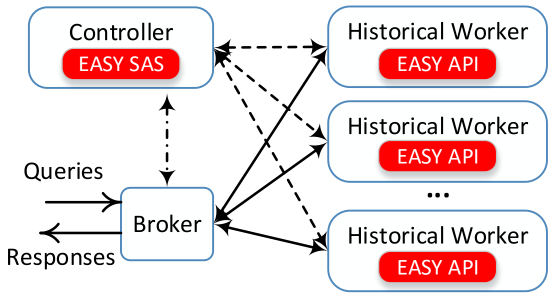


Figure 6.1: EASY's solution architecture. Components we add are shaded in red.

works by first *passively computing* the server load created by segments as queries operate on them. Then, EASY models this segment load and *predicts, at run time*, the future load induced by a segment during its remaining (finite) lifetime. This task is complicated by the fact that load depends critically on the age of a segment; we find that, as time passes, the popularity *and* load contribution of a segment decreases non-linearly.

We implement EASY on top of Pinot and experimentally evaluate our SAS using a custom LinkedIn-like data and query set (guided by the first author's understanding of LinkedIn's Pinot system while he was interning there); we open source all our implementation and code [102]. Our results show that EASY significantly improves the load balance among worker nodes, reducing query tail latencies by up to 6–21% when compared to the default SAS of Pinot and Druid. Importantly, EASY requires few changes and creates negligible overhead.

In summary, the contributions of this chapter are:

- We present a novel and efficient load-aware SAS for Pinot.
- We design and implement a realistic dataset and benchmark for evaluating Pinot, and open source it [102].
- We implement our SAS on Pinot (publicly available [102]), and experimentally evaluate it by comparing with the default SAS of Pinot and Druid.

6.2 Solution Architecture

Figure 6.1 shows the solution architecture of EASY; the components that make up EASY are shown in red. Since the controller manages SAS, we implement our EASY SAS in the controller; the mathematical details of our SAS are presented in the next Section. When a new segment is generated, the controller sends a request to all workers. Each worker, in turn, computes its cost function and returns the value to the controller via an API call. The controller then picks the r workers that have the smallest cost values, and places r replicas of the incoming segment on these workers. r is a user-specified value; we set $r = 1$ in our implementation. To facilitate the computation of the cost function, each worker logs the total CPU time spent, cpu_time_Q, and the total number of rows scanned, row_scan_Q, by each query Q. Note that Q will likely span multiple segments; we thus also log a list of segments scanned by Q. However, we do not log segment-level information, such as segment-level CPU time and rows scanned, as this information logging requires significant overhead and may be computationally infeasible. Instead, we estimate segment-level information from cpu_time_Q and row_scan_Q, as discussed next.

6.3 EASY's Cost Function

Recall that Pinot selects the lowest cost worker nodes for each incoming segment (Section 6.1). The default cost function in Pinot assigns one unit of cost for each segment in a worker node, thus assigning an incoming segment to the r workers with the lowest number of segments. Unfortunately, this cost function does not take into account the server load that each segment contributes and may contribute in the future. The cost function for EASY is specifically designed to address this shortcoming efficiently.

High-level idea. The high-level idea behind EASY's cost function is to estimate the server load that each segment will induce *during its remaining lifetime*. The server load contribution of a segment is challenging to compute as it depends on several factors, including (i) the popularity of the segment, (ii) the size of the segment (number of rows), (iii) the query mix that typically targets the segment and its relative complexity, and (iv) the structure of the segment (number of columns and their content). Worse, predicting the load that a segment may contribute to in the future requires an understanding of how induced load changes with time. Clearly, modeling all of these factors will require significant time and effort, leading to inefficient SAS design.

Instead, EASY directly models the total server load contribution of a segment of a given table based on previously observed data. Explicitly, we compute the *total CPU time spent by all queries actively scanning a segment*, and use this as a proxy for load contribution. We find that this CPU time per segment *per* query decreases with the age of a segment, possibly because of caching. We thus also model this decaying trend of CPU time as a function of the *segment age* (the difference between current time and segment start time).

To enable predictions of future load that a segment may induce, we learn the *cpu_time per row as a function of segment age* for a typical segment of each table. Then, for any segment of a table, we predict its cpu_time contribution based on its number of rows during its entire lifetime as it ages (since segments expire after some expiry time).

Our approach differs from existing approaches since we predict the *future load* induced by any segment. Further, we model the actual load induced by a segment as opposed to only modeling its popularity or frequency of access, which are not accurate enough estimators of load (see Section 6.5).

6.3.1 Passive Model Training

EASY passively computes its estimates of load per segment based on the measured load induced by incoming queries on existing segments. Further, to account for changes in workload, EASY periodically updates its estimates in each interval (one hour, in our implementation).

Computing cpu_time per segment. As discussed in Section 6.2, we track the total cpu_time of each query Q, say cpu_time_Q, at each worker. To determine the contribution of individual segments to this cpu time, we also keep a track of the segments, and the specific time range within the segments, that each query scans. Let S_Q be the set of segments scanned by query Q, and let t_s be the time range, in hours, of segment $s \in S_Q$ that Q scans (obtained via the WHERE clause of Q). In our implementation of Pinot, each segment represents one day, and so the fraction of segment s that is scanned by Q is $f_s = t_s/24$. We now estimate the number of rows of s scanned by Q (not directly available via Pinot) as $f_s \times row\_count_s$, where row_count_s is the total number of rows in segment s and is already known to Pinot. Finally, we estimate the contribution of segment $s \in S_Q$ to cpu_time_Q

as:

$$cpu\_time^s_Q = cpu\_time_Q \times \frac{f_s \times row\_count_s}{\sum_{x \in S_Q} f_x \times row\_count_x} \tag{6.1}$$

The total cpu_time contribution of s based on all queries observed in the past interval is then estimated as:

$$cpu\_time^s = \sum_{observed\ Q} cpu\_time^s_Q \tag{6.2}$$

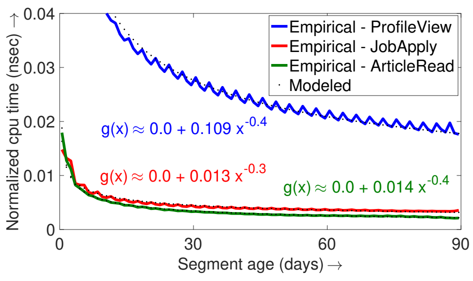
Computing row_scan per segment. We use a similar approach to estimate the number of rows scanned for segment s by all queries in the past interval as:

$$row\_scan^s = \sum_{observed\ Q} \frac{row\_scan_Q \times f_s \times row\_count_s}{\sum_{x \in S_Q} f_x \times row\_count_x}, \tag{6.3}$$

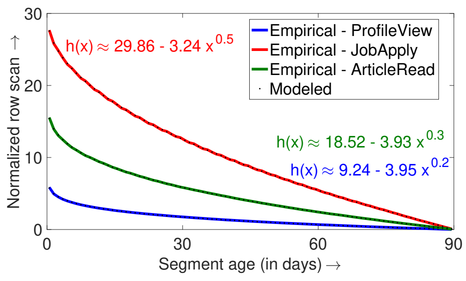
where row_scan_Q (logged by EASY) is the total number of rows, across all segments, scanned by query Q.
Load modeling as a function of age. We now model the load induced by any segment based on its age; this will allow us to online predict the future load created by a segment in Section 6.3.2. To enable load prediction for any segment size, we normalize cpu_time^s by row_scan^s; we refer to this as:
normalized cpu_time: the total cpu time per scanned row of segment s incurred by all queries in the last interval.
Likewise, we normalize row_scan^s by row_count^s to get:
normalized row_scan: the total rows scanned per row contained in segment s by all queries in the last interval.

Figure 6.2 shows our empirical results for normalized *cpu_time* and *row_scan* for three different Pinot tables (see Section 6.5.3 for details on our experimental setup). We see that both values decrease non-linearly with segment age; the decrease for row_scan is to be expected as segment popularity drops with time (older segments are queried less frequently compared to newer segments).

To enable efficient predictions for new segments, we model the empirical observations. Given that popularity for segments is Zipf distributed, we fit the empirical values as $c_0 + c_1/x^\alpha$, where c_0 and c_1 are coefficients to be learned and α is a parameter. Our regression results for these models are shown as dotted lines in Figure 6.2 along with the modeled equations. The



(a) Normalized *cpu_time* as a function of segment age.



(b) Normalized *row_scan* as a function of segment age.

Figure 6.2: Empirical and modeled estimates for normalized cpu_time and row_scan for segments of three different tables. Also shown are the regression fit model equations for each case. The mean modeling error is less than 5% for cpu_time and less than 3% for row_scan for all tables.

regression fit is very close to the empirical observations, thus the dotted lines coincide with the solid (empirical) lines in the figure. The modeling error for cpu_time (g(x) in Figure 6.2(a)) is 3.24%, 4.11%, and 2.75% for ProfileView, JobApply, and ArticleRead tables, respectively. The modeling error for row_scan (h(x) in Figure 6.2(b)) is 2.94%, 1.16%, and 0.97% for ProfileView, JobApply, and ArticleRead tables, respectively.

6.3.2 Online Load Prediction

To predict the future load induced by a segment, EASY leverages the above described models of $g()$ and $h()$, and integrates the predicted load over the remaining lifetime of the segment. In particular, at time t, for a segment s with segment start time $start_s$ and row_count_s total rows, EASY predicts its future load as:

$$load_s(t) = row\_count_s \times \int_{t-start_s}^{expiry} g(x)\ h(x)\ dx, \tag{6.4}$$

where $t-start_s$ is the age of s and $expiry$ (3 months in our implementation) is the expiration duration of s. Note that $g(x)\times h(x)$ represents total CPU time per row of segment s, and thus multiplying this quantity with row_count_s gives us the total CPU time for segment s; integrating over the remaining lifetime gives us the predicted load induced by s.

Since our accurate models for normalized cpu time, $g(x) = a + b \cdot x^{\alpha}$, and row scan, $h(x) = c + d \cdot x^{\beta}$, are relatively easy to express (where a, b, c, d, α, and β are regression coefficients, as shown in Figure 6.2), we can obtain Eq. (6.5) in closed-form as:

$$\begin{aligned} load_s(t) = row\_count_s \cdot \Bigg(& a.c(expiry - t + start_s) + \frac{a.d}{(\beta+1)} \left(expiry^{\beta+1} \right. \\ & \left. - (t - start_s)^{\beta+1}\right) + \frac{b.c}{(\alpha+1)} \left(expiry^{\alpha+1} - (t - start_s)^{\alpha+1}\right) \\ & + \frac{b.d}{(\alpha+\beta+1)} \left(expiry^{\alpha+\beta+1} - (t - start_s)^{\alpha+\beta+1}\right) \Bigg) \end{aligned} \tag{6.5}$$

Given this closed-form expression, computing the segment load under EASY is computationally efficient; hence the name EASY (Efficient segment Assignment StrategY).

6.3.3 Putting It All Together

We are now ready to define our cost function. For a worker w with current set of segments S_w at time t, the EASY cost is:

$$cost(w, t) = \sum_{s \in S_w} load_s(t) \tag{6.6}$$

Finally, for an incoming segment at time t, EASY selects the r workers with lowest $cost(w, t)$ for placement.

6.4 Prior Work

We now discuss important prior work on SAS. We implement EASY on top of Pinot by modifying Pinot's SAS. The default SAS for Pinot balances the number of segments across workers. By contrast, EASY aims to minimize query tail latencies by reducing the load imbalance between workers; we show in Section 6.5 that EASY significantly outperforms the default Pinot SAS.

The closest systems to Pinot are Druid [159] and ClickHouse [17]. Druid's SAS [159] is similar to EASY, except that Druid's cost function depends only on the time range of a segment and not its load. As we show in Section 6.5, EASY outperforms Druid by specifically taking segment load into account. ClickHouse [17] is also an OLAP system but does not employ time-ranged segments, like Pinot. Data is distributed over workers based on weights that must be manually assigned by cluster administrators.

Getafix [48] uses a modified bin packing approach to distribute incoming segments across workers based on their popularity. The authors define segment popularity in terms of access count of the segment, and popularity is aged exponentially. Likewise, Copeland et al. [21] distribute data segments to worker nodes so as to balance the access frequency of resident data objects.

Furtado [40] proposes a data placement schema based on hash-partitioning to favor most frequently accessed keys for a relational database. BlowFish [69] maintains a request queue per segment and uses queue length as an estimator of segment load; this queue length information is then used to distribute segments across servers. However, the access frequency or outstanding requests for a segment may not directly correlate with the segment load. For example, a less popular segment may still contribute significantly to server load because of its size or its structure (e.g., number of columns). By contrast, EASY models popularity based on its estimated load, which is a more direct indicator of the cost of a segment than its access frequency.

We now discuss related works that address problems similar to segment assignment in Pinot, but in different types of systems or scenarios. Curino et al. [23] propose a resource estimation technique to better consolidate multiple online data processing workloads on physical servers. However, they do not take the time range of data into account, which is an essential factor in accurately estimating segment load. Wong et al. [151] consider the subset of segments required to service a relational database query, and use this information to consolidate segments onto servers. However, under Pinot, since segments are created over time, the subset of segments required by a query changes dynamically. Ozmen et al. [104] address the problem of generating an optimized layout for a given set of database objects by formulating it as a non-linear program. The resulting layout both balances load and avoids interference. By contrast, EASY's approach is much more efficient and only relies on load and popularity estimates, which can be easily obtained. Pinot partitions data based on timestamps as queries are expected to apply to a particular range of time. This is not the case for general OLAP where all dimensions may have equal importance. VOLAP [29] migrates data shards among OLAP workers to reduce load imbalance.

There are also related works that address the problem of tenant placement in Database-as-a-Service deployments (e.g., STeP [124] and Pythia [36]) or placement of different databases across servers (e.g., Schaffner et al. [116]). While similar, the SAS problem is distinguished by the concept of time-ranged segments which complicates the load distribution challenges.

6.5 Evaluation

We first describe our experimental setup and evaluation methodology, and then present our evaluation results comparing EASY to Pinot SAS and Druid SAS.

6.5.1 Pinot Benchmark

Before illustrating our Pinot benchmark, we provide more insights on Pinot basics (see Section 2.2.1). Pinot processes recent data (e.g, a few days old) using Realtime Workers and older data (may overlap with realtime data) using Historical Workers, as shown in Figure 2.2. Realtime data is pushed to Historical Workers as time passes (e.g., daily) or when a given number of records have been ingested; the data is pushed via Kafka and HDFS. In this thesis we focus on Historical Workers, that store the bulk of the data.

Historical Workers store data in the form of a pre-built index called *segment*; every table has its own segments. Segments store contiguous data for a given time range; there is a row of data columns for each time interval within the range. Every segment thus has an associated start time and end time for its data (in the Time column). Note that there may be a table-specific expiry time that dictates how long segments should be retained by Historical Workers. Once the expiry time, say 3 months, elapses, the associated segments are deleted.

We implement a query generator benchmark for Pinot. To mimic the LinkedIn functionality, we create the following (self-explanatory) tables: ProfileView, JobApply, and ArticleRead. Each table has several columns; for example, ProfileView has columns: Time, ViewerProfileId, ViewerWorkPlace, WereProfilesConnected, etc. For each table, we create several relevant queries. An example query for the ProfileView table is "SELECT * FROM ProfileView WHERE ViewStartTime $> t_1$ AND ViewStartTime $< t_2$", where t_1 and t_2 are (randomized) query parameters. For every table, the exact mix (or ratio) of queries, parameter values, and query rate, can be specified. Every query requests data from a table with time range length (based on WHERE clause) being Zipf distributed and end time being the wall clock time when the query is issued.

Our benchmark is implemented in ∼2000 lines of code and schema files. The table and query design is guided by our understanding of the Pinot system used by LinkedIn (based on the first author's internship at LinkedIn). All implementation details, including code, tables, and queries, have been open sourced for reference [102].

6.5.2 Implementing EASY on Pinot

We implement EASY in Java for integration with Pinot (also written in Java). On the controller side, we implement EASY SAS with ∼200 lines of code. On the worker side, we implement the EASY RESTless API and Pinot logging extensions with ∼500 lines of code. The API is used to compute the $cost(w, t)$ function at each worker w and return the value to the controller. We record cpu_time for each query via java.lang.management.ThreadMXBean; we verified the correctness of our cpu_time implementation with engineering staff at LinkedIn (when the first author was interning at LinkedIn). We also expose the list of segments being targeted by a query in the final log. The overhead of EASY is negligible in practice, especially since we integrate our logging efforts with the efficient LogFactory class used by LinkedIn in their production Pinot implementation. For reference, we have open sourced our EASY-equipped Pinot implementation [102].

6.5.3 Experimental Setup

We use two different setups for our experiments:

1. Physical setup: We use 7 servers for our experiments, with 1 controller, 2 brokers, and 4 worker nodes. All servers are identical with 4 cores (Intel Xeon CPU E3-1231) and 16GB of memory (of which 12GB is assigned to Pinot Java processes). Servers are connected through 1GB network links.

2. Virtual setup: We use 4 Virtual Machines (VMs) for our experiments, with 1 controller, 1 broker, and 2 worker nodes. All VMs are identical with 4 cores and 8GB of memory (of which 4GB is assigned to Pinot

Java processes). Underlying physical servers are connected through 1GB network links.

We use the physical setup, unless otherwise stated. We use the Pinot benchmark explained with details in Section 6.5.1.

6.5.4 Evaluation Methodology

Metrics. We evaluate SAS in terms of two metrics:

1. $\mathbf{T_{99}}$: 99%ile query tail latency as seen by the broker(s); a metric that LinkedIn uses internally [77].

2. $\mathbf{CPU}_\sigma$: standard deviation of the CPU usage across workers, a metric we aim to minimize to, in turn, reduce T_{99}.

Baselines. We compare EASY with the following SAS:

1. **BalanceNum**: This default Pinot SAS aims to balance the number of segments across workers. An incoming segment is assigned to the worker with the least number of segments.

2. **Spread**: This is the Druid SAS in use at Metamarkets which aims to avoid hotspots by spreading apart segments that are closer in time as they are likely to be queried together [34]. For segments X and Y, Spread defines:

$$cost(X,Y) = \int_{x_0}^{x_1} \int_{y_0}^{y_1} e^{-\lambda|x-y|}\ dx\ dy, \tag{6.7}$$

   where $[x_0, x_1)$ and $[y_0, y_1)$ is the time range of X and Y, respectively, and λ is the decay rate. For an incoming segment X, Spread selects the worker k which results in minimum $\sum_{y \in S_k} cost(X,Y)$, where S_k is the set of segments on k. The intuition behind this cost function is to place X at a worker that does *not* contain too many segments which are likely to be queried together with X (have neighboring time ranges) to minimize contention.

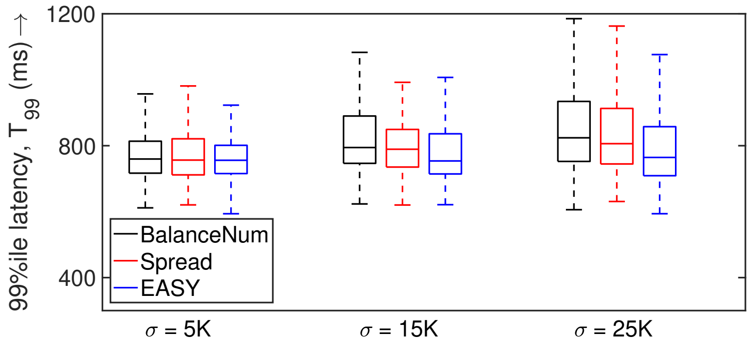


Figure 6.3: Boxplot illustrating the T_{99} for different SAS as a function of increasing standard deviation of segment size (σ). For $\sigma = 5K, 15K,$ and $25K$, EASY reduces T_{99} by 1%, 5%, and 6% when compared to BalanceNum and by 1%, 4%, and 5% when compared to Spread.

6.5.5 Results for Physical Setup Testbed

We illustrate evaluation results under various scenarios. In each case, we use normalized cpu_time and row_scan information about segments from the past interval (one hour) to guide the SAS, as described in Section 6.3.

SAS for different segment size variability. We first consider a scenario where 90 segments (for 90 days of data) are assigned to four worker nodes via the specified SAS. We then run our benchmark and generate queries over these 90 segments for the next 30 minutes. This experiment uses the ProfileView table; segment sizes (row count) are Normally distributed with mean $\mu = 30K$ and varying standard deviation, σ.

Figure 6.3 shows the boxplot (including median and first and third quartiles) for our experimental results for T_{99} under BalanceNum, Spread, and EASY. We find that EASY reduces T_{99} moderately by around 1-6% when compared to BalanceNum and Spread. The improvement is larger for higher variability in segment sizes. This is to be expected as BalanceNum and Spread do not explicitly take segment size into account, while EASY implicitly takes the segment size into account when learning the load contributions of segments (see Section 6.3).

Finally, EASY reduces CPU_σ by 18.38% and 3.51% when compared to BalanceNum and Spread, respectively. These results show that the improvement afforded by EASY over BalanceNum is significant. However, the im-

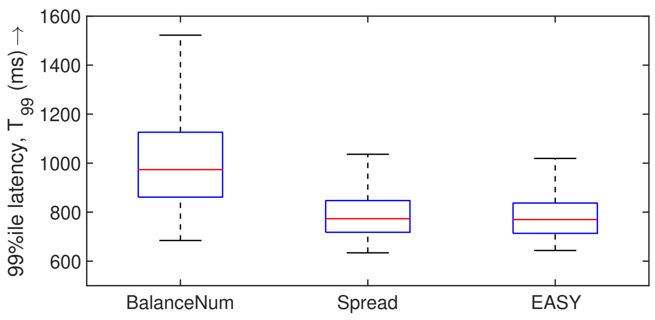


Figure 6.4: Boxplot illustrating the T_{99} under different SAS for the scenario where a worker node is added. EASY reduces T_{99} by 21.55% and 1.61% when compared to BalanceNum and Spread, respectively.

provement over Spread is quite small; we show later that EASY outperforms Spread significantly when there are multiple data tables.

SAS when adding workers. We next consider the more challenging scenario where a new worker node is added to scale capacity and accommodate new segments. Specifically, we start with three worker nodes which are assigned 60 segments via their SAS. Then, a fourth worker node is added and 30 new segments are assigned (across all workers). We monitor query latencies from this point onwards for the next 30 minutes. This experiment uses the ProfileView table; segment sizes are Normally distributed with $\mu = 30K$ and $\sigma = 1K$.

Figure 6.4 shows our experimental results for T_{99} under BalanceNum, Spread, and EASY. We find that EASY reduces T_{99} by 21.55% and 1.61% when compared to BalanceNum and Spread, respectively. Likewise, EASY improves query throughput (not shown) by 13.38% and 1.04% when compared to BalanceNum and Spread, respectively. Finally, EASY reduces CPU_{σ} by 18.38% and 3.51% when compared to BalanceNum and Spread, respectively.

The above results show that the improvement afforded by EASY over BalanceNum is significant. This is because BalanceNum assigns most of the 30 new segments to the fourth (empty) worker node, resulting in a hotspot as newer segments are queried more often. By contrast, both EASY and Spread take recency of segments into account, thus providing better load balancing.

SAS with multiple tables. We now experiment with segments from all three tables (see Section 6.5.3). We assign 28 segments (for the month of

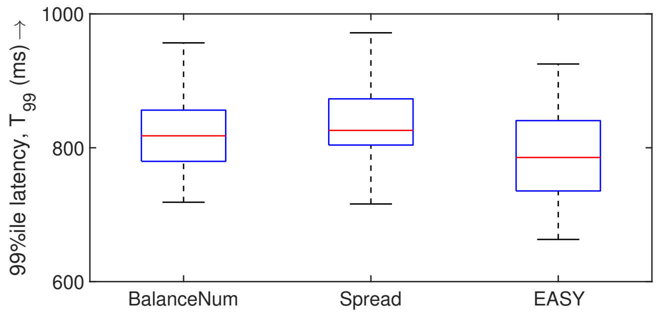
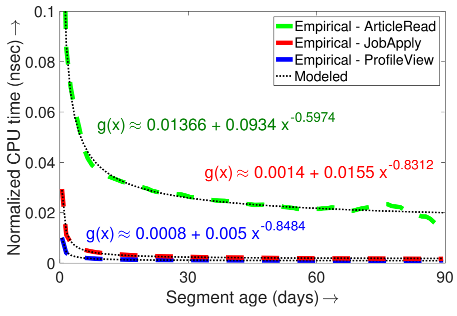


Figure 6.5: Boxplot illustrating the T_{99} under different SAS for the case of multiple tables. EASY reduces T_{99} by 4.93% and 6.33% when compared to BalanceNum and Spread, respectively.

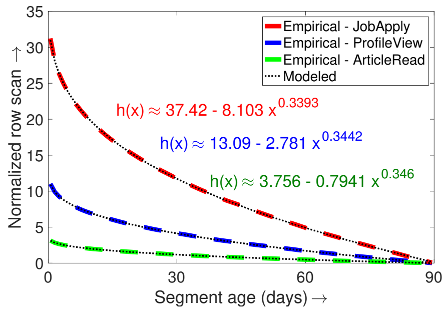
February) for each table to 4 workers; assignment follows the specified SAS. We assign segments chronologically – segments for a given day for all tables, and then segments for the next day for all tables.

BalanceNum tries to balance the number of segments for *each* table across workers. Spread considers segments from all tables on a worker node, but assigns a higher cost in Eq. (6.7) (by a factor 2×) if a pair of segments belong to the same table as they are then more likely to be queried together [34]. EASY does not use a pair-wise cost function (as in Spread), and easily extends to the case of multiple tables by considering segments from all tables on a worker ($s \in S_w$ in Eq. (6.6) can be from any table) when computing the cost for a worker.

Figure 6.5 shows our experimental results for T_{99}. This time, EASY reduces T_{99} by 4.93% and 6.33% when compared to BalanceNum and Spread, respectively. Likewise, EASY improves query throughput (not shown) by 13.38% and 1.04% when compared to BalanceNum and Spread, respectively. Finally, EASY reduces CPU_σ by 18.38% and 3.51% when compared to BalanceNum and Spread, respectively. These results show that EASY affords moderate improvements over Spread as well. Spread performs poorly in this case as it does not take into account the relative difference in the load contributed by segments of different tables. That is, segments of different tables with the same age are treated equally, even though they may induce different loads on the workers due to differences in their structure and content as well as incoming query rate and pattern. By contrast, EASY learns these differences over time and thus treats segments from different tables differently.



(a) Normalized *cpu_time* as a function of segment age.



(b) Normalized *row_scan* as a function of segment age.

Figure 6.6: Empirical and modeled estimates for normalized cpu_time and row_scan for segments of three different tables (cluster of virtual machines). Also shown are the regression fit model equations for each case. The mean modeling error is less than 3.75% for cpu_time and less than 3.15% for row_scan for all tables.

6.5.6 Results for Virtual Server Testbed

To further emphasize the benefits of EASY, we extend Pinot benchmark queries and tune their parameters such that segments of different tables induce varying CPU load. The regression results for the new models are shown as dotted lines in Figure 6.6 along with the modeled equations. The modeling error for cpu_time (g(x) in Figure 6.6(a)) is 2.82%, 4.68%, and 3.76% for ProfileView, JobApply, and ArticleRead tables, respectively. The modeling error for row_scan (h(x) in Figure 6.6(b)) is 1.67%, 3.18%, and 4.51% for ProfileView, JobApply, and ArticleRead tables, respectively.

SAS with multiple tables. We now experiment with segments from all the three tables. We assign 90 segments per table chronologically to 2 workers; assignment follows the specified SAS. We repeat this experiment 10 times. Figure 6.7 shows our experimental results for T_{99}. This time, EASY reduces T_{99} by 19.77% and CPU_σ by 86.84% when compared to BalanceNum. The reduction of T_{99} when compared to Spread is negligible. These results show that EASY affords significant improvements over BalanceNum, but delivers equal improvements when compared to Spread. In the next experiment we show how BalanceNum affords considerable improvements over Spread as well.

SAS when adding workers. We next consider the more challenging sce-

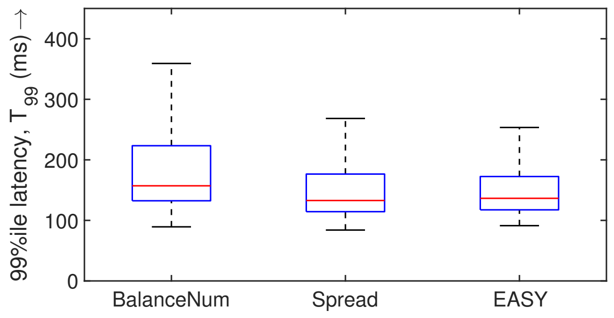


Figure 6.7: Boxplot illustrating the T_{99} under different SAS for the case of multiple tables and cluster of virtual machines. EASY reduces T_{99} by 19.77% when compared to BalanceNum.

nario where a new worker node is added to scale capacity and accommodate new segments. Specifically, we start with one worker node which is assigned 45 segments per table via the SAS. Then, a second worker node is added and 45 new segments per table are assigned (across all workers). We monitor query latencies from this point onwards for the next 10 minutes. We repeat this experiment 10 times.

Figure 6.8 shows our experimental results for T_{99} under BalanceNum, Spread, and EASY. We find that EASY reduces T_{99} by 16.11% and 12.84% when compared to BalanceNum and Spread, respectively. Finally, EASY reduces CPU_σ by 85% and 63.96% when compared to BalanceNum and Spread, respectively.

The above results show that the improvement afforded by EASY over BalanceNum and Spread is significant. This is because BalanceNum assigns all of the 45 new segments (per table) to the second (empty) worker node, resulting in a hotspot as newer segments are queried more often. While Spread takes recency of segments into account and provides better load balancing, EASY delivers significant improvements over Spread as well because EASY takes both recency and varying induced loads of segments into account.

6.6 Conclusion

Efficiently using cloud resources is important for cloud tenants. The scheduling component of cloud-deployed applications plays a crucial role in how

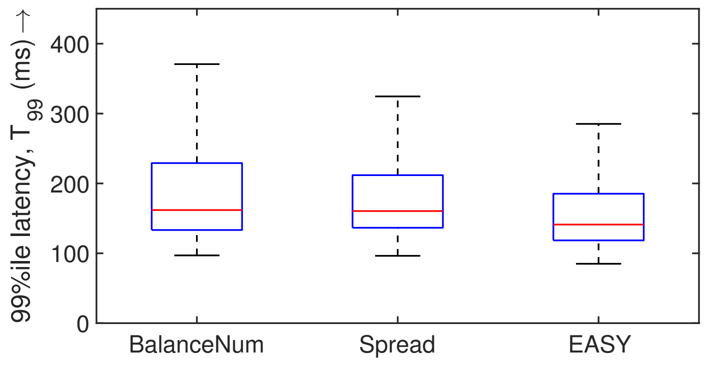


Figure 6.8: Boxplot illustrating the T_{99} under different SAS for the case of multiple tables and cluster of virtual machines where a worker node is added. EASY reduces T_{99} by 16.11% and 12.84% when compared to BalanceNum and Spread, respectively.

efficiently a cluster of cloud resources are used. The broad scheduling challenge is how to prevent hotspots and load imbalance among the cluster of cloud resources. We consider this challenge in the context of OLAP systems that are important due to the emergence of the big-data paradigm. OLAP systems typically split a big table into several data segments and distribute these data segments among a cluster of worker nodes. To serve a query, every worker node runs the query on its assigned data segments, and then these local results are integrated to compute the final response. Accordingly, we have a data segment scheduling problem with the goal of having a balanced load among the worker nodes to prevent hotspots and high tail latency.

In this chapter, we presented EASY, an efficient SAS (Segment Assignment Strategy) for OLAP systems, such as Pinot [108] and Druid [159]. The key idea in EASY is to model the CPU time contribution of each segment, and leverage this modeling to predict the future load induced by segments of a server. Experimental results show that SAS based on our accurate model predictions provides significantly lower query tail latencies when compared to the SAS of Pinot and Druid.

Chapter 7

Conclusions

Cloud computing is a leading technology that delivers virtually unlimited computing services over the Internet. Cloud users often acquire a cluster of resources (e.g., VMs) to run their applications. The scheduling component of applications plays a crucial role in determining the end-to-end application performance (e.g., tail latency). In emerging cloud environments, scheduling decisions are complicated by the fact that underlying resource capacity may vary dynamically due to resource contention, in addition to the traditional challenge of varying workload demand and data popularity. Our goal in this dissertation is to address some of the important scheduling challenges in cloud environments, and to provide practical and analytically rigorous solutions that empower both cloud provider and cloud user. We specifically address three challenges and we now summarize the contributions made by this thesis regarding these three challenges:

1. **Cloud tenant VM's variable resource capacity**: Due to the performance interference challenge, we have a request scheduling problem for load-balanced applications with the goal of having minimum tail latency where the applications are running on a cluster of VMs facing unpredictable performance.

   We presented DIAL (Chapter 4), a user-centric Dynamic Interference-Aware Load balancing framework that can be employed directly by cloud users without requiring any assistance from the hypervisor or cloud provider to reduce tail response times during interference. DIAL works by leveraging two critical components: (i) an accurate, user-centric response time-monitoring based interference detector, classifier, and estimator, and (ii) a framework for deriving theoretically optimal load balancer weights under interference. We use analytical tools for both components resulting in a rigorous and generic methodology that can be extended to other scenarios. Our experimental results for web and OLAP applications on several cloud platforms, under interference from realistic benchmarks, demonstrate the benefits of DIAL.

2. **Background workload impact on cloud tenant VM performance**: Running batch workloads in the background is a common practice to improve server utilization in cloud data centers. However, background workloads can severely impact the cloud tenant VM performance due to the resource contention. Note that cloud providers are not aware of the cloud tenants' (foreground) workloads. Having black-box foreground workloads is a significant challenge that has not been adequately addressed by the several prior works (see Section 3.2). Furthermore, Section 2.4 illustrated that there is still significant room for utilization improvement by analyzing real-world resource usage traces. Accordingly, to address these challenges, batch workloads need to be scheduled next to the black-box foreground workloads (tenants' VMs) with two competing goals: (1) foreground workloads' SLO are not violated, and (2) background workloads' progress rate is maximized.

   Chapter 5 presented Scavenger, an application-agnostic resource manager to improve resource utilization in public cloud servers. Scavenger works by colocating Spark batch job containers with black-box customer 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 customer VMs. The design of Scavenger exploits the cgroups feature in Linux to address CPU and LLC contention; we believe our design ideas could be extended to other cache levels as well. For memory management, we rely on carefully monitoring and regulating the memory assigned to background Spark containers. Comprehensive experimental results on our Lab and Cloud testbeds using KVM and Docker show that Scavenger can significantly increase resource utilization while minimizing the impact on the resource demands of the black-box customer VMs (often less than 10%).

3. **Impact of hotspots and load imbalance on application tail latency**: Efficiently using cloud resources is important for cloud tenants. The scheduling component of cloud-deployed applications plays a crucial role in how efficiently a cluster of cloud resources are used. The broad scheduling challenge is how to prevent hotspots and load imbalance among the cluster of cloud resources. We consider this challenge in the context of Online Analytical Processing (OLAP) systems that are important due to the emergence of the big-data paradigm. OLAP systems typically split a big table into several data segments and distribute these data segments among a cluster of worker nodes. To serve a query, every worker node runs the query on its assigned data segments, and then these local results are integrated to compute the final response. Accordingly, we have a data segment scheduling problem with the goal of having a balanced load among the worker nodes to prevent hotspots and high tail latency.

   In Chapter 6, we presented EASY, an efficient SAS (Segment Assignment Strategy) for OLAP systems, such as Pinot [108] and Druid [159]. The key idea in EASY is to model the CPU time contribution of each segment, and leverage this modeling to predict the future load induced by segments of a server. Experimental results show that SAS based on our accurate model predictions provides significantly lower query tail latencies when compared to the SAS of Pinot and Druid.

The design of our scheduling solutions in this dissertation demonstrated that using analytical approaches for dynamic scheduling can improve cloud-deployed applications' performance. We showed that our proposed approaches reduce the applications' tail latency significantly compared to the baseline scenarios where our dynamic solutions are not deployed and compared to other state-of-art approaches.

7.1 Future Work

Running background workloads next to primary workloads to improve data center utilization has been a hot research topic for many years and has been deployed in production by many IT companies. While in this dissertation we explored this topic from new perspectives, including the realistic black-box assumption for customer workloads, there are still interesting and promising perspectives that can be explored. Some of these new research directions that are beyond this thesis are as follows:

- It will be interesting to study the efficacy of Machine Learning (ML) techniques to predict the resource demand of customer workloads in the next time interval and increase or decrease background workloads' pressure accordingly. Besides, ML techniques can help us dynamically tune our solution parameters in response to, for example, workload variability. The primary requirement will be the design and integration of ML methods with Scavenger that can be easily implemented in production systems without introducing additional complexity.

- It will be promising to extend Scavenger, our proposed background workload manager, by taking new hardware technologies into account. For instance, in our current study we did not have access to physical servers equipped with Intel Cache Allocation Technology (CAT) [98]. CAT enables fine-grained LLC capacity management by providing a mechanism to assign cache ways to cores to enable cache isolation. In our current study, the goal was to propose a technique that can be deployed in different generations of servers. However, it will be interesting to leverage Scavenger's resource regulation algorithm for dynamic LLC allocation to background workloads in CAT-equipped servers. This can significantly outperform the static LLC allocation schemes in terms of LLC utilization, and thus background workloads' progress rate. In such a scenario, background workloads will be allowed to use idle cores in full capacity while their assigned LLC is dynamically regulated. We think that the CPU utilization improvement can be significant while the LLC regulation governs the trade-off between potential foreground workloads' performance degradation and the progress rate of the background workloads.

Bibliography

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

[2] Yasaman Amannejad, Diwakar Krishnamurthy, and Behrouz Far. Detecting Performance Interference in Cloud-based Web Services. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 423–431. IEEE, 2015.

[3] Amazon Inc. Amazon Auto Scaling. `http://aws.amazon.com/autoscaling`.

[4] Amazon Inc. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`.

[5] Amazon Inc. How Elastic Load Balancing Works. `http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/how-elb-works.html`.

[6] Amazon Web Services, Inc. Amazon EC2 Dedicated Hosts. `https://aws.amazon.com/ec2/dedicated-hosts/`.

[7] A Big Data and AI Benchmark Suite. `http://prof.ict.ac.cn/`.

[8] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 557–558. ACM, 2010.

[9] M.A.A. Boon, E.M.M. Winands, I.J.B.F. Adan, and A.C.C. van Wijk. Closed-form Waiting Time Approximations for Polling Systems. *Performance Evaluation*, 68(3):290 – 306, 2011.

[10] Xiangping Bu, Jia Rao, and Cheng-zhong Xu. Interference and Locality-Aware Task Scheduling for MapReduce Applications in Virtual Clusters. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 227–238, New York, NY, USA, 2013.

[11] Carl Brooks. Amazon does not oversubscribe. `http://searchcloudcomputing.techtarget.com/blog/The-Troposphere/Amazon-does-not-oversubscribe`.

[12] Giuliano Casale, Carlo Ragusa, and Panos Parpas. A Feasibility Study of Host-level Contention Detection by Guest Virtual Machines. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 152–157. IEEE, 2013.

[13] Shuang Chen, Christina Delimitrou, and José F. Martínez. 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*, ASPLOS '19, pages 107–120, Providence, RI, USA, 2019.

[14] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt. CloudScope: Diagnosing and Managing Performance Interference in Multi-tenant Clouds. In *IEEE 23rd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 164–173, Atlanta, GA, USA, 2015.

[15] R.C. Chiang and H.H. Huang. TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 1–12, Seattle, WA, USA, 2011.

[16] John Ciancutti. 5 Lessons We have Learned Using AWS. `http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html`, 2010.

[17] ClickHouse — Open Source Distributed Column-oriented DBMS. `https://clickhouse.yandex`.

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

[19] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, OSDI '04, San Francisco, CA, USA, 2004.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, Indianapolis, IN, USA, 2010.

[21] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data Placement in Bubba. In *SIGMOD'88*, pages 99–108, 1988.

[22] 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, Shanghai, China, 2017.

[23] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware Database Monitoring and Consolidation. In *SIGMOD'11*, pages 313–324, 2011.

[24] David Lo et al. Heracles: Improving Resource Efficiency at Scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

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

[26] D. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut. PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1742–1755, 2016.

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

[28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, Stevenson, Washington, USA, 2007.

[29] Frank Dehne, David Robillard, Andrew Rau-Chaplin, and Neil Burke. VOLAP: A Scalable Distributed System for Real-time OLAP with High Velocity Data. In *IEEE Cluster'16*, pages 354–363, 2016.

[30] Christina Delimitrou and Christos Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. *ACM Transactions on Computer Systems*, 31(4), 2013.

[31] 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, Salt Lake City, UT, USA, 2014.

[32] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: High Quality and Low Latency Scheduling in Large, Shared Clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, SOCC '15, pages 97–110, Kohala Coast, HI, USA, 2015.

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

[34] Distributing Data in Druid at Petabyte Scale. `https://metamarkets.com/2016/distributing-data-in-druid-at-petabyte-scale`.

[35] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 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*, pages 335–346, Pittsburgh, PA, USA, 2010.

[36] Aaron J Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng Yan. Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs. In *SIGMOD'13*, pages 517–528, 2013.

[37] AWS Case Study: Expedia. `https://aws.amazon.com/solutions/case-studies/expedia`.

[38] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D Bowers, and Michael M Swift. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 20. ACM, 2012.

[39] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 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*, ASPLOS '12, pages 37–48, London, UK, 2012.

[40] Pedro Furtado. Experimental Evidence on Partitioning in Parallel Data Warehouses. In *DOLAP'04*, pages 23–30, 2004.

[41] Anshul Gandhi and Justin Chan. Analyzing the Network for AWS Distributed Cloud Computing. *ACM SIGMETRICS Performance Evaluation Review*, 43(3):12–15, 2015.

[42] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Harsha Ellanti. The Unobservability Problem in Clouds. In *Proceedings of the 2015 IEEE International Conference on Cloud and Autonomic Computing*, Cambridge, MA, USA, 2015.

[43] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, Model-driven Autoscaling for Cloud Applications. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 57–64, 2014.

[44] Anshul Gandhi, Mor Harchol-Balter, and Michael A Kozuch. Are Sleep States Effective in Data Centers? In *2012 International Conference on Green Computing Conference (IGCC)*, pages 1–10. IEEE, 2012.

[45] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael Kozuch. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *Transactions on Computer Systems*, 30, 2012.

[46] Anshul Gandhi, Xi Zhang, and Naman Mittal. HALO: Heterogeneity-Aware Load Balancing. In *Proceedings of the 23rd IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '15, Atlanta, GA, USA, 2015.

[47] Wanling Gao, Jianfeng Zhan, Lei Wang, Chunjie Luo, Daoyi Zheng, Rui Ren, Chen Zheng, Gang Lu, Jingwei Li, Zheng Cao, Shujie Zhang, and Haoning Tang. Bigdatabench: A dwarf-based big data and AI benchmark suite. *CoRR*, abs/1802.08254, 2018.

[48] Mainak Ghosh, Le Xu, Xiaoyao Qian, Thomas Kao, Indranil Gupta, and Himanshu Gupta. Getafix: Workload-aware Distributed Interactive Analytics. Technical report, University of Illinois Urbana-Champaign, 2016.

[49] Devarshi Ghoshal, Richard Shane Canon, and Lavanya Ramakrishnan. I/O Performance of Virtualized Cloud Environments. In *Proceedings of the second international workshop on data intensive computing in the clouds*, pages 71–80. ACM, 2011.

[50] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling Data-parallel Jobs Against Live Production Systems. In *Proceedings of the 2015 Usenix Annual Technical Conference*, USENIX ATC '15, pages 459–471, Santa Clara, CA, USA, 2015.

[51] Google Cloud Platform. Auto Scaling on the Google Cloud Platform. `http://cloud.google.com/resources/articles/auto-scaling-on-the-google-cloud-platform`.

[52] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the*

2nd ACM Symposium on Cloud Computing, SOCC '11, pages 1–14, Cascais, Portugal, 2011.

[53] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 1–14, Oakland, CA, USA, 2015.

[54] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[55] The Reliable, High Performance TCP/HTTP Load Balancer. `http://www.haproxy.org`.

[56] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[57] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory Overbooking and Dynamic Control of Xen Virtual Machines in Consolidated Environments. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 630–637. IEEE, 2009.

[58] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22, 2011.

[59] Tibor Horvath and Kevin Skadron. Multi-mode Energy Management for Multi-tier Server Clusters. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 270–279, Toronto, ON, Canada, 2008.

[60] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 519–532, Boston, MA, USA, 2018.

[61] J. Mukherjee et al. Subscriber-driven Interference Detection for Cloud-based Web Services. *IEEE Transactions on Network and Service Management*, 14(1):48–62, 2017.

[62] Pawel Janus and Krzysztof Rzadca. SLO-aware Colocation of Data Center Tasks Based on Instantaneous Processor Requirements. In *Proceedings of the 8th ACM Symposium on Cloud Computing*, pages 256–268, Santa Clara, CA, USA, 2017.

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

[64] Seyyed Ahmad Javadi, Shalini Bhaskara, Rahul Doshi, Prashanth Soundarapandian, Muhammad Wajahat, and Anshul Gandhi. Application-Agnostic Batch Workload Management in Cloud Environments. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 504–504. ACM, 2018.

[65] Seyyed Ahmad Javadi, Harsh Gupta, Robin Manhas, Shweta Sahu, and Anshul Gandhi. EASY: Efficient Segment Assignment Strategy for Reducing Tail Latencies in Pinot. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1432–1437. IEEE, 2018.

[66] JF Im et al. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of SIGMOD 2018*, pages 583–594. ACM, 2018.

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

[68] Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Gumshoe: Diagnosing Performance Problems in Replicated File-Systems. In *Proceedings of the 2008 Symposium on Reliable Distributed Systems*, SRDS '08, pages 137–146, Naples, Italy, 2008.

[69] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *NSDI'16*, pages 485–500, 2016.

[70] Younggyun Koh, R. Knauerhase, P. Brett, M. Bowman, Zhihua Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems Software*, pages 200–209, 2007.

[71] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *IEEE International Symposium on Performance Analysis of Systems & Software*, pages 200–209. IEEE, 2007.

[72] Kleinrock L. *Queueing Systems, Volume 2*. Wiley-Interscience, New York, 1976.

[73] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, Amsterdam, The Netherlands, 2014.

[74] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, pages 1–14, Melbourne, Australia, 2010.

[75] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 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.

[76] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 95–106, Chicago, IL, USA, 2010.

[77] Who Moved My 99th Percentile Latency? `https://engineering.linkedin.com/performance/who-moved-my-99th-percentile-latency`.

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

[79] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. Performance and Energy Modeling for Live Migration of Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, pages 171–182, San Jose, CA, USA, 2011.

[80] Qixiao Liu and Zhibin Yu. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: a View from Alibaba Trace. In *Proceedings of 2018 ACM Symposium on Cloud Computing*, Carlsbad, CA, USA, 2018.

[81] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 213–224, Newport Beach, CA, USA, 2011.

[82] 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, Portland, OR, USA, 2015.

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

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

[85] Amiya K Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating Interference in Cloud Services by Middleware

Reconfiguration. In *Proceedings of the 15th International Middleware Conference*, pages 277–288. ACM, 2014.

[86] Amiya K. Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 277–288, Bordeaux, France, 2014.

[87] Alexandros Marinos and Gerard Briscoe. Community Cloud Computing. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 472–484, Beijing, China, 2009.

[88] 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 Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '11, pages 248–259, Porto Alegre, Brazil, 2011.

[89] Peter Mell, Tim Grance, et al. The NIST Definition of Cloud Computing. 2011.

[90] Jeffrey C Mogul and Ramana Rao Kompella. Inferring the Network Latency Requirements of Cloud Tenants. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[91] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. FlexDCP: A QoS Framework for CMP Architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, 2009.

[92] David Mosberger and Tai Jin. httperf—A Tool for Measuring Web Server Performance. *ACM Sigmetrics: Performance Evaluation Review*, 26:31–37, 1998.

[93] J. Mukherjee, D. Krishnamurthy, and J. Rolia. Resource Contention Detection in Virtualized Environments. *IEEE Transactions on Network and Service Management*, 12(2):217–231, 2015.

[94] Joydeep Mukherjee and Diwakar Krishnamurthy. Subscriber-Driven Cloud Interference Mitigation for Network Services. In *Proceedings of IWQoS 2018*, pages 1–6, 2018.

[95] N. Mishra et al. ESP: A machine learning approach to predicting application interference. In *Proceedings of ICAC 2017*, pages 125–134, 2017.

[96] Senthil Nathan, Umesh Bellur, and Purushottam Kulkarni. Towards a Comprehensive Performance Model of Virtual Machine Live Migration. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 288–301, Kohala Coast, HI, USA, 2015.

[97] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing Performance Interference Effects for QOS-Aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, Paris, France, 2010.

[98] Khang T. Nguyen. 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.

[99] Dejan Novakovi, Nedeljko Vasi, Stanko Novakovi, Dejan Kosti, and Ricardo Bianchini. Deepdive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of 2013 USENIX Annual Technical Conference*, ATC '13, pages 219–230, San Jose, CA, USA, 2013.

[100] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 219–230, 2013.

[101] Openstack.org. OpenStack Open Source Cloud Computing Software. `http://www.openstack.org`.

[102] PACELab/pinot. `https://github.com/PACELab/pinot`.

[103] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Symposium on Operating Systems Principles*, pages 69–84, Farminton, PA, USA, 2013.

[104] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-aware Storage Layout for Database Systems. In *SIGMOD'10*, pages 939–950, 2010.

[105] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review*, 28(4):303–314, 1998.

[106] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying Cloud Benchmarking. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 122–132. IEEE, 2016.

[107] How Performance Issues Impact Cloud Adoption. `https://www.rickscloud.com/how-performanceissues-impact-cloud-adoption`.

[108] Pinot — A Realtime Distributed OLAP Datastore. `https://github.com/linkedin/pinot`.

[109] Xing Pu, Ling Liu, Yiduo Mei, S. Sivathanu, Younggyun Koh, C. Pu, and Yuanda Cao. Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds. *IEEE Transactions on Services Computing*, 6(3):314–329, 2013.

[110] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding Performance Interference of i/o Workload in Virtualized Cloud Environments. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 51–58. IEEE, 2010.

[111] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. Who is Your Neighbor: Net i/o Performance Interference in Virtualized Clouds. *Services Computing, IEEE Transactions on*, 6(3):314–329, 2013.

[112] Moinuddin K. Qureshi and Yale N. Patt. 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, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[113] Martin I. Reiman and Burton Simon. An Interpolation Approximation for Queueing Systems with Poisson Input. *Operations Research*, 36(3):454–469, 1988.

[114] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, November 2011. Revised 2014-11-17 for version 2.1. Posted at `https://github.com/google/cluster-data`.

[115] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[116] Jan Schaffner, Dean Jacobs, Tim Kraska, and Hasso Plattner. The Multi-Tenant Data Placement Problem. In *DBKDA'12*, 2012.

[117] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management. *SIGMETRICS Perform. Eval. Rev.*, 39(1):13–24, 2011.

[118] E Shurman and J Brutlag. The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search, 2009.

[119] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10, Incline Village, NV, USA, 2010.

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

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

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

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

[124] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J Elmore, Michael Stonebraker, and David DeWitt. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *SOCC'16*, pages 388–400, 2016.

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

[126] An open source machine learning frameworks for everyone. `https://www.tensorflow.org`.

[127] Selome Kostentinos Tesfatsion, Eddie Wadbro, and Johan Tordsson. PerfGreen: Performance and Energy Aware Resource Provisioning for Heterogeneous Clouds. In *Proceedings of the 2018 IEEE International Conference on Autonomic Computing*, pages 81–90, Trento, Italy, 2018.

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

[129] The Apache Software Foundation. Apache Spark. `http://spark.apache.org`.

[130] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf. 2006.

[131] A. Tiwari and P. Kanungo. Dynamic Load Balancing Algorithm for Scalable Heterogeneous Web Server Cluster with Content Awareness. In *Proceedings of Trendz in Information Sciences and Computing*, pages 143–148, Chennai, India, 2010.

[132] Top Ten Challenges of Cloud Implementation. `http://www.peak10.com/top-ten-challenges-ofcloud-implementation`.

[133] Top 5 Service Performance Challenges in the Cloud. `http://www.apmdigest.com/top-5-serviceperformance-challenges-in-the-cloud`.

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

[135] Erik-Jan van Baaren. WikiBench: A distributed, Wikipedia based web application benchmark. Master's thesis, Vrije Univesiteit Amsterdam, the Netherlands, 2009.

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

[137] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-Freeing Attacks: Improve your Cloud Performance (at your Neighbor's Expense). In *Proceedings of the 19th ACM conference on Computer and communications security*, ACM '12, pages 281–292, Raleigh, NC, USA, 2012.

[138] Nedeljko Vasi, Dejan Novakovi, Svetozar Miu in, Dejan Kosti, and Ricardo Bianchini. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 423–436, London, UK, 2012.

[139] 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 4th Annual Symposium on Cloud Computing*, pages 5:1–5:16, Santa Clara, CA, USA, 2013.

[140] 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 SOCC 2013*, Santa Clara, CA, USA, 2017.

[141] 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 10th European Conference on Computer Systems*, Bordeaux, France, 2015.

[142] W Viraf. The Value of a Millisecond: Finding the Optimal speed of a Trading Infrastructure, 2008.

[143] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. In *Proceedings of the 1st International Conference on Cloud Computing*, pages 254–265, Beijing, China, 2009.

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

[145] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, Lydia Y. Chen, Robert Birke, and George Kesidis. Effective Capacity Modulation as an Explicit Control Knob for Public Cloud Profitability. In *Proceedings of the 13th IEEE International Conference on Autonomic Computing*, Wurzburg, Germany, 2016.

[146] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, Lydia Y Chen, Robert Birke, and George Kesidis. Effective Capacity Modulation as an Explicit Control Knob for Public Cloud Profitability. Technical report, Technical report, http://www.cse.psu.edu/research/publications/tech-reports/2016/CSE-16-001.pdf, 2016.

[147] Guohui Wang and TS Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[148] Hui Wang, Canturk Isci, Lavanya Subramanian, Jongmoo Choi, Depei Qian, and Onur Mutlu. A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters. *ACM SIGPLAN Notices*, 50(7):93–106, 2015.

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

[150] John Wilkes. More Google cluster data. Google research blog, November 2011. Posted at `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html`.

[151] E. Wong and R. H. Katz. Distributing a Database for Parallelism. In *SIGMOD'83*, pages 23–29, 1983.

[152] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Sandpiper: Black-box and Gray-box Resource Management for Virtual Machines. *Computer Networks*, 53(17):2923–2938, 2009.

[153] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service. In *Proceedings of the 13th EuroSys Conference*, EuroSys '18, pages 14:1–14:13, Porto, Portugal, 2018.

[154] F. Xu, F. Liu, and H. Jin. Heterogeneity and Interference-Aware Virtual Machine Provisioning for Predictable Performance in the Cloud. *IEEE Transactions on Computers*, 65(8):2470–2483, 2016.

[155] Fei Xu, Fangming Liu, Linghui Liu, Hai Jin, Bo Li, and Baochun Li. iAware: Making Live Migration of Virtual Machines Interference-Aware in the Cloud. *IEEE Transactions on Computers*, 63(12):3012–3025, 2014.

[156] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is Better: Avoiding Latency Traps in Virtualized Data Centers. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 7. ACM, 2013.

[157] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is Better: Avoiding Latency Traps in Virtualized Data Centers. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Santa Clara, CA, USA, 2013.

[158] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. TR-Spark: Transient Computing for Big Data Analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 484–496, Santa Clara, CA, USA, 2016.

[159] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A Real-time Analytical Data Store. In *SIGMOD'14*, pages 157–168, 2014.

[160] L. Youseff, M. Butrico, and D. Da Silva. Toward a Unified Ontology of Cloud Computing. In *Proceedings of the 2008 Grid Computing Environments Workshop*, pages 1–10, Austin, TX, USA, 2008.

[161] Y. Yuan, H. Wang, D. Wang, and J. Liu. On interference-aware provisioning for cloud-based big data processing. In *Proceedings of the 21st International Symposium on Quality of Service*, pages 1–6, Montreal, Canada, 2013.

[162] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. SPARK: Cluster Computing with Working Sets. *HotCloud*, 10:10–10, 2010.

[163] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '10, Boston, MA, USA, 2010.

[164] Qi Zhang, Ludmila Cherkasova, Ningfang Mi, and Evgenia Smirni. A Regression-based Analytic Model for Capacity Planning of Multi-tier Applications. *Cluster Computing*, 11(3):197–211, 2008.

[165] Wei Zhang, Sundaresan Rajasekaran, Timothy Wood, and Mingfa Zhu. MIMP: Deadline and Interference Aware Scheduling of Hadoop Virtual Machines. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 394–403. IEEE, 2014.

[166] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Hardware Execution Throttling for Multi-core Resource Management. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, San Diego, CA, USA, 2009.

[167] 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.

[168] 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*, EuroSys '13, pages 379–391, Prague, Czech Republic, 2013.

[169] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. 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*, pages 755–770, 2016.

[170] 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):62–71, 2015.

[171] Haishan Zhu and Mattan Erez. 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*, pages 33–47, Atlanta, GA, USA, 2016.