Architecting Data Centers for High Efficiency and Low Latency

by

Yunqi Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Yunqi Zhang

yunqi@umich.edu

ORCID iD: 0000-0003-3610-1965

To my parents, Yuhong Shi and Baoguo Zhang.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the guidance from a team of great mentors. My advisors, Jason and Lingjia, I can never thank you enough for how much impact you have had on my life. Lingjia, you taught me to reason with scientific rigour and defend my own work. Jason, your passion and perseverance gave me the courage to pursue what I truly believe regardless of what it takes. David, my mentor during my internships at Facebook, you taught me to care about the impact and enjoy what I am doing. I would also like to thank Ricardo and Íñigo, who mentored me during my internship at Microsoft Research and have supported me ever since. I thank my dissertation committee – Lingjia, Jason, Tom, Christos, Judy – for their guidance in constructing this dissertation.

I owe a great deal of gratitude to all the members of Clarity Lab: Mike, Johann, Yiping, Animesh, Chang-Hong, Parker, Shih-Chieh, Ram, Austin, Matt, Steve, Vini, Md, Hailong, Quan, Jeongseob, Moeiz, Yunsheng, Arjun and John. You have helped me grow as a scientist, and without you graduate school would have been nowhere near as much fun as I have had.

Yuhong Shi and Baoguo Zhang, my parents – thank you for always letting me make my own decisions, respecting my decisions, and unconditionally supporting me to chase my dreams. To my family, friends, and all those who wished me well, thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Modern data centers, housing remarkably powerful computational capacity, are built in massive scales and consume a huge amount of energy. The energy consumption of data centers has mushroomed from virtually nothing to about three percent of the global electricity supply in the last decade, and will continuously grow. Unfortunately, a significant fraction of this energy consumption is wasted due to the inefficiency of current data center architectures, and one of the key reasons behind this inefficiency is the stringent response latency requirements of the user-facing services hosted in these data centers such as web search and social networks. To deliver such low response latency, data center operators often have to overprovision resources to handle high peaks in user load and unexpected load spikes, resulting in low efficiency.

This dissertation investigates data center architecture designs that reconcile high system efficiency and low response latency. To increase the efficiency, we propose techniques that understand both microarchitectural-level resource sharing and system-level resource usage dynamics to enable highly efficient co-locations of latency-critical services and low-priority batch workloads. We investigate the resource sharing on real-system simultaneous multithreading (SMT) processors to enable SMT co-locations by precisely predicting the performance interference. We then leverage historical resource usage patterns to further optimize the task scheduling algorithm and data placement policy to improve the efficiency of workload co-locations. Moreover, we introduce methodologies to better manage the response latency by automatically attributing the source of tail latency to low-level architectural and system configurations in both offline load testing environment and online production environment. We design and develop a response latency evaluation framework at microsecond-level precision for data center applications, with which we construct statistical inference procedures to attribute the source of tail latency. Finally, we present an approach that proactively enacts carefully designed causal inference micro-experiments to diagnose the root causes of response latency anomalies, and automatically correct them to reduce the response latency.

CHAPTER I

Introduction

Modern data centers are built in massive scales to provide the computing resources to keep up with the fast growing demand in cloud computing. In the last decade, the electricity consumed by data centers globally has grown from virtually nothing to about three percent of the entire global energy supply [108]. While that is already 39% higher than the total consumption of the entire United Kingdom, this number is likely to triple in the next decade as noted by recent studies [37]. However, a significant amount of this energy is wasted due to the inefficiency of current data center architectures.

One of the key reasons for this inefficiency is that many of such large-scale data centers are built to host user-facing, interactive services such as web search and social networks, which often operate under stringent response latency requirements, especially tail latency (*i.e.*, high quantiles of the response latency distribution). This is because high response latency often results in poor user experience, and sometimes even service abandonment. Nevertheless, is is extremely challenging to keep the tail latency low as the size and complexity of the data center systems scale up, and it has been identified as one of the key challenges facing modern data center architecture design.

This dissertation investigates the inefficiency of modern data centers, and proposes a novel data center architecture that optimizes for high efficiency and low latency.

1.1 Motivation

This section motivates the need for architecting data centers for high efficiency and low latency in the context of the critical challenges facing modern data center architecture.

1.1.1 Resource Efficiency

The geometric growth of computation in the cloud drives rapidly increasing costs in building and operating large-scale data centers, such as those operated by Google, Facebook and Microsoft. Unfortunately, the utilization these data centers is often low, especially in clusters that host user-facing, interactive services [35, 64]. The reasons for this include: these services are often latency-critical (*i.e.*, require low tail response times); may exhibit high peaks in user load; and must reserve capacity for unexpected load spikes and failures. This low utilization results in low resource efficiency and high total cost of ownership (TCO).

An effective approach for improving data center efficiency is the co-location of useful batch workloads (*e.g.*, data analytics, machine learning) and the data they require on the same servers that perform other functions, including those that run latency-critical services. However, for co-location with these services to be acceptable, we must shield them from any non-trivial performance interference produced by the batch workloads or their storage accesses, even when unexpected events occur. If co-location starts to degrade response times, the scheduler must throttle or even kill (and re-start elsewhere) the culprit batch workloads. In either case, the performance of the batch workloads suffers. Nevertheless, co-location ultimately reduces TCO [164], as the batch workloads are not latency-critical and share the same infrastructure as the services, instead of needing their own.

Recent scheduling research has considered how to carefully select which batch workload to co-locate with each service to minimize the potential for interference (most commonly, last-level cache interference on chip multiprocess servers), *e.g.* [63, 64, 122, 179]. Based on this precise interference prediction, data center schedulers can identify "safe" co-locations that bound performance degradation while improving server efficiency. However, prior techniques only focus on predicting the interference caused by resource sharing across cores on a multicore processor. Despite the ubiquitous presence of simultaneous multithreading (SMT) processors [159, 158] in modern data centers, an approach to perform precise interference prediction on real-system SMT processors has been an open problem.

Realizing precise prediction for SMT co-locations in addition to CMP co-locations is a particularly challenging problem due to significantly more complex interactions between shared resources within the core and in the uncore. In addition to the last-level cache (LLC) and memory bandwidth, which are shared across CMP cores, SMT cores provide much finer granularity resource sharing on core, among hardware contexts. The additional shared resources include private cache(s), memory ports, as well as integer and floating-point functional units. This fine-grained sharing across a large number of resources leads to greater performance variability and unpredictability. In addition, there is diversity in how resources are shared, which further increases the difficulty of precise interference prediction. For example, a functional unit cannot be shared concurrently by multiple hardware contexts, however a cache's capacity can be shared simultaneously. In this work, we demonstrate that the difference between CMP and SMT resource sharing calls for a fundamental redesign in the way we model interference.

Moreover, these works either assume simple sequential batch applications or overlook the resource utilization dynamics of real services. Scheduling data-intensive workloads comprising many distributed tasks (*e.g.*, data analytics jobs) is challenging, as scheduling decisions must be made in tandem for collections of these tasks for best performance. The resource utilization dynamics make matters worse. For example, a long-running workload may have some of its tasks throttled or killed when the load on a co-located service increases.

Additionally, no prior study has explored in detail the co-location of services with data for batch workloads. Real services often leave large amounts of spare storage space (and bandwidth) that can be used to store the data needed by the batch workloads. However, co-locating storage raises even more challenges, as the management and utilization of the services may affect data durability and availability. For example, service engineers and the management system itself may reimage (reformat) disks, deleting all of their data. Reimaging typically results from persistent state management, service deployment, robustness testing, or disk failure. Co-location and reimaging may cause all replicas of a data block to be destroyed before they can be re-generated.

1.1.2 Response Latency

Managing response latency, especially tail latency (*i.e.*, high quantiles of the response latency distribution), is one of the most critical tasks for data center operators to provide quality of service (QoS) guarantees. This is because high latency can easily result in poor user experience, service abandonment and lost revenue, particularly for interactive services such as web search and social networks. These services are powered by clusters of machines wherein a single request is distributed among a large number of servers in a "fan-out" pattern. In such design, the overall performance of such systems depends on the slowest responding machine [59]. Recent work has sought to control and understand these tail requests both at the individual server and overall cluster level [110].

For data center operators, the capability of accurately measuring tail latency without disrupting the production system is important for a number of reasons. First, servers are typically acquired in large quantities (*e.g.*, 1000s at a time), so it is important to choose the best design possible and carefully provision resources. Evaluating hardware configurations requires extensive and accurate measurements against existing workloads. Second, it is necessary to be able to faithfully measure performance effects without disrupting production systems. The high frequency of software and hardware changes makes it extremely hard, or impossible, to evaluate these changes in production, because it can easily result in user-visible incidents. Instead, it is desirable to understand the impact of performance-critical decisions in a safe, but accurate load testing environment.

Building such load testing environment for accurate tail latency measurement is particularly challenging. This is primarily because there are significantly more systems and resources involved for large-scale Internet service workloads (*e.g.*, distributed server-side software, network connections, etc) than traditional single-server workloads (*e.g.*, SPEC CPU2006, PARSEC). Although there have been several prior works [53, 70, 74, 115, 166, 90, 89] trying to bridge this gap recently, they have several pitfalls in their load test design as we will show later in the dissertation. Unfortunately, these tools are commonly used in research publications for evaluation and the pitfalls may result in misleading conclusions. Similar to the academic community, there is also a lack of an accurate tail latency measurement test bed in industry, causing unnecessary resource over-provisioning [35] and unexplained performance regressions [71].

Furthermore, to be able to control the tail latency of these Internet services, a thorough and correct understanding of the source of tail latency is required. These Internet services interact with a wide range of systems and resources including operating system, network stack and server hardware thus the ability of quantitatively attributing the source of tail latency to individual resources is critical yet challenging.

Although a number of prior works [63, 64, 99, 102, 110, 120, 118, 176, 180, 96, 171, 151, 122, 149] have studied the impact of individual resources on the tail latency, many resources have complex interacting behaviors that cannot be captured in isolated studies. For example, Turbo Boost and DVFS governor may interact indirectly through competing for the thermal headroom. Note that the capability of identifying the source of tail latency relies on the first aforementioned challenge. In other words, without an accurate measurement of the tail latency we will not be able to correctly attribute it to various sources.

In addition, performance anomalies, manifested as significant unexplained QoS degradations, frequently occur in production environments, causing user-visible performance incidents. Architectural and low-level system factors including cache contention, non-uniform memory access (NUMA) issues, and suboptimal thread-to-core mappings have been shown to be common causes for such performance anomalies [60, 152, 122, 63, 61, 64, 171, 180, 72, 67, 162, 143, 111, 132, 38, 114, 118, 120, 124, 56, 167]. For example, suboptimal IRQ-to-core mappings have been shown to cause a 3× query latency degradation in Nginx web server, and mismanaged remote NUMA accesses can cause up to 5× performance degradation in Memcached [111].

However, it is extremely challenging to always optimally configure these factors. Firstly, misconfigurations may not manifest themselves until the system is under certain states (*e.g.*, suboptimal frequency boost configurations may remain hidden until the server is under heavy load [181]), and data center workloads are constantly changing [19, 3]. The heterogeneity of data centers (*i.e.*, composed of different hardware platforms [83, 121] and variety of network topologies [161]) further complicates the configurations. Identifying the optimal configuration in controlled offline experiments is extremely difficult because production environment is much more complex [161].

Unfortunately, in current data centers, due to the lack of effective performance diagnosis techniques to pinpoint the root causes of the observed anomaly, these anomalies often remain unaddressed, thereby resulting in QoS violations. Although techniques have been developed in the software community to analyze the performance impact of software configurations [25, 52, 165, 168, 49, 128, 134, 145, 33, 40, 146, 170, 34, 101, 173, 138, 127, 32, 137, 50, 29, 157, 30, 144], most of these techniques focus only on application-level software configuration issues, and many require instrumentation in the source code. These prior works are not applicable for handling QoS anomalies caused by hardware and low-level system issues, because such issues do not manifest themselves at the source code level (*e.g.*, suboptimal voltage frequency or remote NUMA access slows down all the code) and cannot be diagnosed by application-level investigation. Large data center operators like Facebook still rely on manual instrumentation and inspection to diagnose performance anomalies [161], which is extremely expensive in both time consumption and engineering effort. Therefore, new techniques that can reason about hardware-caused performance anomalies are needed to automatically diagnose and correct such performance anomalies.

Diagnosing QoS anomalies caused by low-level architectural factors is particularly challenging for several reasons.

- **Root Cause Identification:** Correlation analysis has been commonly used by prior work [52, 165, 168, 170, 101, 127] to "diagnose" the causes of performance anomalies. However, these techniques can generate misleading conclusions and fail to identify the *root causes* of anomalies. Correlation analysis cannot establish causal relationship and many symptoms can appear to be correlated due to their interactions with the actual root causes. For instance, when the CPU frequency is lowered, causing QoS degradation, the number of cache misses per second decreases since the memory access rate is lower. Correlation analysis may mistakenly identify cache miss rate reduction as one of the root causes. Therefore, a technique that infers causal relationship rather than correlation is needed to identify the actual root causes.

- **System Complexity:** There are a large number of hardware and system-level factors that could cause performance anomalies, as well as complex interactions among the factors, further complicating the diagnosis [181]. Designing a comprehensive system that achieves coverage over the wide spectrum of factors without sacrificing generality and efficiency is challenging.

- **Intolerance of Overhead:** Despite the high complexity of identifying the root causes, to design a system that can be left in place continuously in production, we need to carefully control its overhead to be no more than 1-2% [133].

1.2 Architecting for High Efficiency and Low Latency

This section summarizes the design of the proposed data center architecture for high efficiency and low latency.

1.2.1 SMiTe: Precise QoS Prediction to Enable SMT Co-locations

This dissertation first presents a real-system investigation to better understand how applications interfere on commodity SMT multicore processors. From the investigation, we have gained several insights that guide the design of an SMT interference prediction methodology.

- Firstly, across various shared resources (caches, functional units, memory ports, etc.), contention on each individual resource alone can cause significant performance degradation and the amounts of degradation exhibit high variability across applications and resources.

- Secondly, there is little correlation among application contention characteristics for different shared resources. For example, an application being sensitive to contention for data caches does not necessarily mean that it is less (or more) contentious (or sensitive to contention) for the floating-point functional unit.

These observations indicate that a holistic approach such as those used in prior work for interference prediction on CMP servers is not suitable for SMT. For example, Bubble-Up [122] relies on a single monotonic metric to quantify interference in all shared resources, which fails to capture the multidimensionality of the resource sharing behavior on SMT. We must redesign a methodology to model interference for SMT co-locations in a manner that the sharing behavior is decoupled along multiple dimensions of various types of resources.

Based on these observations, we design `SMiTe`, a methodology that enables precise performance prediction on real-system SMT processors. `SMiTe` leverages a carefully designed suite of software stressors, called `Rulers`, to characterize an application's contention nature for each shared resource. Each `Ruler` in the suite is designed to maximize the pressure on one specific resource while minimizing the pressure on all other resources. By co-locating one application with a `Ruler`, we measure the performance degradation of the application as its *sensitivity*, and the performance degradation of the `Ruler` as the application's *contentiousness* on the corresponding resource. A regression model is then established, using application's *sensitivity* and *contentiousness* for different resources to precisely predict the performance interference in SMT co-locations. Based on the precise prediction, we are able to steer the cluster-level job scheduler to make co-location decisions that improve the data center efficiency without violating QoS requirements.

1.2.2 Harvest: History-Based Resource Harvesting

We propose techniques for harvesting the spare compute cycles and storage space in data centers for distributed batch workloads. We refer to the original workloads of each server as its "primary tenant", and to any resource-harvesting workload (*i.e.*, batch compute tasks or their storage accesses) on the server as a "secondary tenant". We give priority over each server's resources to its primary tenant; secondary tenants may be killed (in case of tasks) or denied (in case of storage accesses) when the primary tenant needs the resources.

To reduce the number of task killings and improve data availability and durability, *we propose task scheduling and data placement techniques that rely on historical resource utilization and disk reimaging patterns.* We logically group primary tenants that exhibit similar patterns in these dimensions. Using the utilization groups, our scheduling technique schedules related batch tasks on servers that have *similar* patterns and enough resources for the tasks' expected durations, and thereby avoids creating stragglers due to a lack of resources. Using the utilization and reimaging groups, our data placement technique places data replicas in servers with *diverse* patterns, and thereby increases durability and availability despite the harvested nature of the storage resources.

To create the groups, we characterize the primary tenants' utilization and reimaging patterns in ten production data centers, including a popular search engine and its supporting services. Each data center hosts up to tens of thousands of servers. Our characterization shows that the common wisdom that data center workloads are periodic is inaccurate, since often most servers do not execute interactive services. We target *all* servers for harvesting.

1.2.3 Treadmill: Attributing the Source of Tail Latency

To better manage tail latency of modern data centers, we first survey existing performance evaluation methodologies for response latency used in the research community. We have identified a set of common pitfalls across these tools:

- **Query inter-arrival generation** - Load testing software is often written for software simplicity. We find commonly used programming paradigms create an implicit queueing model that approximates a closed-loop system and systematically underestimates the tail latency. Instead, we demonstrate a precisely-timed open-loop load tester is necessary to properly exercise the queueing behavior of the system.

- **Statistical aggregation** - Due to high request rates, sampling must be used to control the measurement overhead. Online aggregation of these latency samples must be performed carefully. We find that singular statistics (*e.g.*, a point estimate of the 95th- or 99th-percentile latency) fails to capture detailed information; static histograms used in other load testers also exhibit bias.

- **Client-side queueing bias** - Due to the high throughput rates (100k - 1M requests per second) in many commercial systems, we demonstrate that multiple client machines must be used to test even a single server. Without lightly utilized clients, latency measurements quickly begin to be impacted by client-side queueing, generating biased results.

- **Performance "hysteresis"** - We observe a phenomenon in which the estimated latency converges after collecting a sufficient amount of samples, but upon running the load test again, the test converges to a different value. This is caused by changes in underlying system states such as the mapping of logical memory, threads, and connections to physical resources. We refer to this as hysteresis because no reasonable amount of additional samples can make the two runs converge to the same point. Instead we find experiments must be restarted multiple times and the converged values should be aggregated.

Based on these insights, we propose a systematic procedure for accurate tail latency measurement, and details the design choices that allow us to overcome the pitfalls of existing methodologies. The proposed procedure leverages multiple lightly-utilized instances of `Treadmill`, a modular software load tester, to avoid client-side queueing bias. The software architecture of `Treadmill` preserves proper request inter-arrival timings, and allows easy addition of new workloads without complicated software changes. Importantly, it properly aggregates the distributions across clients, and performs multiple independent experiments to mitigate the effects of performance hysteresis.

The precise measurement achieved by `Treadmill` enables the capability of identifying "where" the latency is coming from. We build upon recent research in quantile regression [105], and attribute tail latency to various hardware features that cause the tail. This allows us to unveil the system "black box" and better understand the impact of tuning hardware configurations on tail latency. We perform this evaluation using Facebook production hardware running two critical Facebook workloads: the pervasive key-value server Memcached and a recently-disclosed software routing system mcrouter [113]. Using our tail latency attribution procedure, we are able to identify many counter-intuitive performance behaviors, including complex interactions among different hardware resources that cannot be captured by prior studies of individual hardware features in isolation.

1.2.4 TailSniping: Pinpointing Root Causes of QoS Anomalies

Hardware and low-level system misconfigurations often manifest themselves via system-level and microarchitectural performance indicators such as hardware performance counters. Moreover, collection of these indicators is common practice in modern production data centers, such as GWP [133, 98] at Google and Scuba [23, 31] at Facebook. A key motivating observation behind this work is that continuous online monitoring of these indicators in data center composed of tens of thousands of servers provides a massive amount of data, a previously untapped resource that can be utilized for continuous anomaly diagnosis and root cause identification.

Building on this observation, we propose `TailSniping`, a system that leverages the data provided by continuous monitoring infrastructure to automatically detect, diagnose and correct a wide range of architectural-caused performance anomalies in data centers. `TailSniping` is composed of two modules: `Nerve` and `Brain`. `Nerve` continuously monitors a rich set of runtime information, sampling and collecting a set of architectural and system events with less than 1% overhead. `Brain` then analyzes the data produced by `Nerve` to detect potential performance anomalies using robust statistical analysis. Once an anomaly has been detected, `Brain` first performs a correlation analysis to find all correlating factors as candidate causes. To pinpoint the root causes among the candidate causes, `Brain` proactively invokes a series of carefully-designed novel causal inference micro-experiments, reproducing each candidate cause at a very small scale (often node-level) online. To infer whether a candidate is the underlying root cause, the system constructs a null hypothesis: the application behaves similarly during the micro-experiment and during the detected anomaly. When the system fails to reject the null hypothesis, it concludes the corresponding candidate as the root cause. The analysis incurs little overhead and handles production issues such as load fluctuation and unknown causes not yet included in the online monitoring infrastructure. With the accurate diagnosis that `Brain` provides, the corresponding misconfigurations can be automatically modified to rescue the affected machines from the performance anomalies.

1.3 Summary of Contributions

This dissertation proposes a novel architecture to improve data center resource efficiency and reduce response latency. A summary of specific contributions is as follows:

- **Precise QoS Prediction to Enable SMT Co-locations -** We conduct an in-depth analysis of performance interference on real-system SMT processors, which demonstrates the low correlation among application contention characteristics for various shared resources. This motivates our design of a multidimensional modeling methodology for SMT co-locations. We propose `Rulers`, carefully designed software stressors to put maximum amount of pressure on each individual resource while incurring minimum pressure on other resources. These `Rulers` allow us to capture an application's *sensitivity* and *contentiousness* for each shared resource in a decoupled manner. Then we introduce `SMiTe`, a methodology to establish a prediction model using `Ruler` measurements to combine the *sensitivity* and *contentiousness* characteristics of each application to precisely predict performance interference on multicore SMT processors. Using the QoS interference prediction provided by `SMiTe` to steer cluster-level scheduling decisions, we demonstrate the effectiveness of the proposed methodology in enabling SMT co-locations to achieve higher data center efficiency.

- **History-Based Resource Harvesting -** We characterize the dynamics of how servers are used and manged in ten production data centers at Microsoft. Based on the insights we gained in the characterization, we propose techniques for improving task scheduling and data placement based on historical behavior of applications and how they are managed. We extend Hadoop stack to harvest the spare cycles and storage in data centers using our techniques. Evaluated on both real systems and simulations, the proposed techniques provide large improvements in batch job performance, data durability, and data availability. Finally, we recently deployed our proposed file system in large-scale production data centers, and we discuss our experience and lessons learned that can be useful for others.

- **Attributing the Source of Tail Latency -** We conduct a survey of existing methodologies in tail latency measurements, and present an empirical demonstration of their shortcomings. We classify these flaws into four major principles for future practitioners. We present the design of a robust experimental methodology, and a software load testing tool `Treadmill`, which we release as open-source software[1]. Both systems properly fulfill the requirements of our principles and are easily extensible for adoption. The high precision measurements achieved by our methodology enables the possibility of understanding the source of tail latency variance using quantile regression. We successfully attribute the majority of the variance to several advanced hardware features and the interactions among them. By carefully tuning the hardware configurations recommended by the attribution results, we significantly reduce the tail latency and its variance.

- **Pinpointing Root Causes of QoS Anomalies -** We present a novel methodology that proactively conducts causal inference micro-experiments to pinpoint the root causes of performance anomalies, whereas prior work using correlation analysis can only provide a list of correlating symptoms without providing any insight about the causal relationship. We design a scalable system, `TailSniping`, that continuously monitors, detects, diagnoses and corrects hardware and system-level misconfiguration-caused performance anomalies with less than 1% performance overhead. We prototype and evaluate `TailSniping` with popular data center applications including Memcached [74], Web-Search [74] and three deep learning-based application from DjiNN [89] on real hardware, and demonstrate its effectiveness by accurately diagnosing six hardware and low-level system performance anomalies.

[1]`https://github.com/facebook/treadmill`

CHAPTER II

Background and Related Work

In this Chapter, we survey the related literature and provide the background relevant to the topics covered in this dissertation. This includes prior works on resource sharing on SMT processors, resource harvesting in data centers, response latency measurement and management, and performance anomaly diagnosis.

2.1 Resource Sharing on Multicore SMT Processors

2.1.1 SMT Processor Scheduling

There has been a large amount of work on resource management for multicore SMT processors [73, 68, 47, 46, 143, 58, 163]. Feliu *et al.* [73] proposes a method to improve the overall throughput by balancing L1 bandwidth usage, however there is no performance guarantees. Eyerman *et al.* present probabilistic job symbiosis [68], which employs specialized performance accounting hardware to facilitate modeling the performance impact for SMT co-locations. Cazorla *et al.* [47, 46] propose a hardware mechanism to track and adjust shared resource usage, then leverage that mechanism to dynamically adjust the resources to meet application QoS targets.

As an alternative to predicting the impact of SMT co-locations, others have used competition heuristics to achieve efficient scheduling. Snavely and Tullsen [143] and De Vuyst *et al.* [58] use a sampling phase to discover the performance interference due to co-locations in order to schedule jobs on SMT processors. Vega *et al.* [163] present a competition heuristic to decide whether multiple threads should be consolidated to the SMT contexts on the same core for multithreaded workloads.

2.1.2 Resource Sharing on Multicore Processors

There are also a number of prior works on hardware or application characterization and modeling using micro-benchmarks [44, 107, 122, 39, 62]. Mars *et al.* [122] present a methodology that uses tunable memory micro-benchmarks to quantify the *sensitivity* and *contentiousness* of an application for the last level cache and memory bandwidth contention. However, their approach is designed only for uncore-level resource sharing. Bertran *et al.* [39] present an approach to automatically generate micro-benchmarks to study the energy-performance trade-offs for multicore SMT processors, in which they use the micro-benchmarks to obtain energy-related platform characterization. Delimitrou *et al.* [62] describe a micro-benchmark suite that can be used to detect resource contention for a number of shared resources in a CMP machine to facilitate intelligent cluster-level scheduling decisions.

Others have studied the performance interference on multicore processors without considering SMT co-locations. Delimitrou *et al.* [64] manage various co-location scenarios in order to improve the resource utilization without violating the QoS target. Tang *et al.* present a compiler [148] and a compiler-supported runtime framework [150] to control low-priority application's *contentiousness* and ensure the QoS of high-priority application, in order to improve the system throughput. Yang *et al.* [171] improve resource utilization by dynamically probing and controlling the execution of low-priority applications to guarantee the QoS of the high-priority applications.

2.2 Resource Harvesting in Data Centers

2.2.1 Resource Harvesting

Prior works have proposed to harvest resources for batch workloads in the absence of co-located latency-critical services, *e.g.* [116, 117]. Recent research has targeted two aspects of co-location: (1) performance isolation – ensuring that batch tasks do not interfere with services, after they have been co-located on the same server [102, 106, 109, 119, 130, 148, 150, 171, 179]; or (2) scheduling – selecting which tasks to co-locate with each service to minimize interference or improve packing quality [63, 64, 79, 122, 164, 180]. Borg addresses both aspects in Google's data centers, using Linux cgroup-based containers, special treatment for latency-critical tasks, and resource harvesting from containers [164].

With respect to resource usage dynamics, a related paper is [45], which derives Service-Level Objectives (SLOs) for resource availability from historical utilization data.

2.2.2 Data-processing Frameworks and Co-locations

Researchers have proposed improvements to the Hadoop stack in the absence of co-location, *e.g.* [26, 54, 80, 81, 82, 100, 174]. Others considered Hadoop (version 1) in co-location scenarios using virtual machines, but ran HDFS on dedicated servers [51, 140, 178]. Lin *et al.* [116] stored data on dedicated and volunteered computers (idle desktops), but in the absence of primary tenants. We are not aware of studies of Mesos [94] in co-location scenarios. Bistro [79] relies on static resource reservations for services, and schedules batch jobs on the leftover resources.

2.3 Response Latency Measurement and Management

2.3.1 Statistically Sound Performance Measurements

There has been large amount of work on developing statistically sound performance evaluation methodology. Mytkowicz *et al.* [126] show the significance of measurement bias commonly existing in computer system research, and present a list of experimental techniques to avoid the bias. Oliveira *et al.* [57] present a study on two Linux Schedulers using statistical methods, which demonstrates that ANOVA can sometimes be insufficient especially for non-normally distributed data whereas quantile regression can provide more conclusive insights. Curtsinger and Berger propose STABILIZER [55], which randomizes the layouts of code, stack and heap objects at runtime to eliminate the measurement bias caused by layout effects in performance evaluation. Alameldeen and Wood [27] leverage confidence interval and hypothesis testing to compensate the variability they discover in architectural simulations for multi-threaded workloads. Tsafrir *et al.* [155, 156] develop input shaking technique to address the environmental sensitivities they observe in parallel job scheduling simulations. Georges *et al.* [78] point out a list of pitfalls in existing Java performance evaluation methodologies, and propose JavaStats to perform rigorous Java performance analysis. Breughe and Eeckhout [41] point out benchmark inputs are critical for rigorous evaluation on microprocessor designs.

There are also a number of prior works reducing the variance of query latency, and improving the tail latency. Shen [141] models the request-level behavior variation caused by resource contention, and proposes a contention-aware OS scheduling algorithm to reduce the tail latency.

2.3.2 Performance Benchmarking for Data Center Workloads

Others have developed benchmark suites that capture the representative workloads in modern data centers. Cooper *et al.* [53] present the Yahoo! Cloud Serving Benchmark (YCSB) framework for benchmarking large-scale distributed data serving applications. Fan *et al.* [70] present and characterize 3 types of representative workload in Google data centers, including web search, web mail and MapReduce. In addition, Lim *et al.* [115] further characterize the video streaming workloads at Google and benchmark them to evaluate new server architectures. Ferdman *et al.* [74] introduce the CloudSuite benchmark suite, which represents the emerging scale-out workloads running in modern data centers. Wang *et al.* [166] enrich the data center workload benchmark by presenting BigDataBench, which covers diverse cloud applications together with representative input data sets. Hauswald *et al.* [90, 89] introduce benchmarks of emerging machine learning data center applications. Meisner *et al.* [125] present a data center simulation infrastructure, BigHouse, that can be used to model data center workloads.

2.4 Performance Anomaly Diagnosis

There has been a large mount of work on diagnosing performance issues using causal and regression analysis [25, 52, 165, 168, 49, 128, 134, 145, 33, 40, 146, 170, 34, 101, 173, 138, 127, 32, 137, 50]. However, most of them focus on software configurations, and none of the prior works applies to hardware configurations in modern data centers, which are often not explicitly managed.

Taint analysis is a widely used class of techniques for performance and security debugging, which dynamically tracks the control and data flow at very fine granularity and attributes the contribution to the observed execution back to configuration and user input [128, 134, 145, 33, 146, 34, 173, 32, 137]. These systems track the data and control flow originated or arithmetically derived from the sources like configuration and user input, and quantify their contribution to the current execution. In addition, there are also prior works that use the dependency and path information at coarse granularity in large-scale distributed systems [25, 49, 138, 50]. These systems trace the interactions among different components in the distributed systems, and infer dependency, convolution, ordering and other relationships from large amount of traces. When the application experiences performance issues, the systems verify if these inferred relationships still hold to diagnose possible root causes. Furthermore, there is also a class of approaches that simply look for correlations between explanatory factors and system performance, in order to help narrow down the possible root causes [52, 165, 168, 170, 101, 127]. These approaches do not explicitly infer causal relationships, instead they leverage statistical techniques (*e.g.*, bayesian networks, decision tree, etc.) to find correlations and require further hypothesis testing to diagnose the root causes. Moreover, there are also systems that leverage user-specified rules to diagnose the causes of performance anomalies [40].

It has been demonstrated that a data center wide continuous performance monitoring infrastructure (*e.g.*, Google-Wide Profiling [133]) can provide useful insights on performance bottlenecks and issues [98]. The tradeoff between performance overhead and powerfulness of the monitoring has been identified as one of the key challenges, that extremely fined-grained monitoring provides higher resolution at performance issues while introducing higher overhead [172].

CHAPTER III

SMiTe: Precise QoS Prediction for SMT Co-locations

One of the key challenges for improving efficiency in data centers is to improve server utilization while guaranteeing the quality of service (QoS) of latency-sensitive applications. To this end, prior work has proposed techniques to precisely predict performance and QoS interference to identify 'safe' application co-locations. However, such techniques are only applicable to resources shared across cores. Achieving such precise interference prediction on real-system simultaneous multithreading (SMT) architectures has been a significantly challenging open problem due to the complexity introduced by sharing resources within a core.

In this Chapter, we demonstrate through a real-system investigation that the fundamental difference between resource sharing behaviors on CMP and SMT architectures calls for a redesign of the way we model interference. For SMT servers, the interference on different shared resources, including private caches, memory ports, as well as integer and floating-point functional units, do not correlate with each other. This insight suggests the necessity of decoupling interference into multiple resource sharing dimensions. In this work, we propose `SMiTe`, a methodology that enables precise performance prediction for SMT co-location on real-system commodity processors. With a set of `Rulers`, which are carefully designed software stressors that apply pressure to a multidimensional space of shared resources, we quantify application *sensitivity* and *contentiousness* in a decoupled manner. We then establish a regression model to combine the *sensitivity* and *contentiousness* in different dimensions to predict performance interference. Using this methodology, we are able to precisely predict the performance interference in SMT co-location with an average error of 2.80% on SPEC CPU2006 and 1.79% on CloudSuite. Our evaluation shows that `SMiTe` allows us to improve the utilization of data centers by up to 42.57% while enforcing an application's QoS requirements.

3.1 Real-System Investigation

In contrast to CMP co-locations where only last-level cache (LLC) and memory bandwidth are shared among different cores, hardware contexts co-located on the same SMT core share a much wider range of resources including both functional units and the memory subsystem. In this section, we present an investigation to better understand application sharing behavior on these various resources and the resulting performance interference on commodity multicore SMT processors.

3.1.1 Experimental Methodology

One main difference between CMP co-locations and SMT co-locations is whether on-core resources are shared. In an Intel Sandy Bridge [8] processor, these resources are implemented as an execution cluster composed of 6 ports that perform sets of different operations. As illustrated in Figure 3.1, ports 0, 1 and 5 are used for functional units, whereas ports 2, 3 and 4 (not shown) are for memory accesses. Note that, in this design, there are many port-specific operations that can only execute on specific port(s). For example, `FP_MUL` can only execute on port 0, `FP_ADD` on port 1, `FP_SHF` on port 5 and `INT_ADD` on ports 0, 1 and 5.

Taking advantage of these port-specific operations, we carefully designed a set of

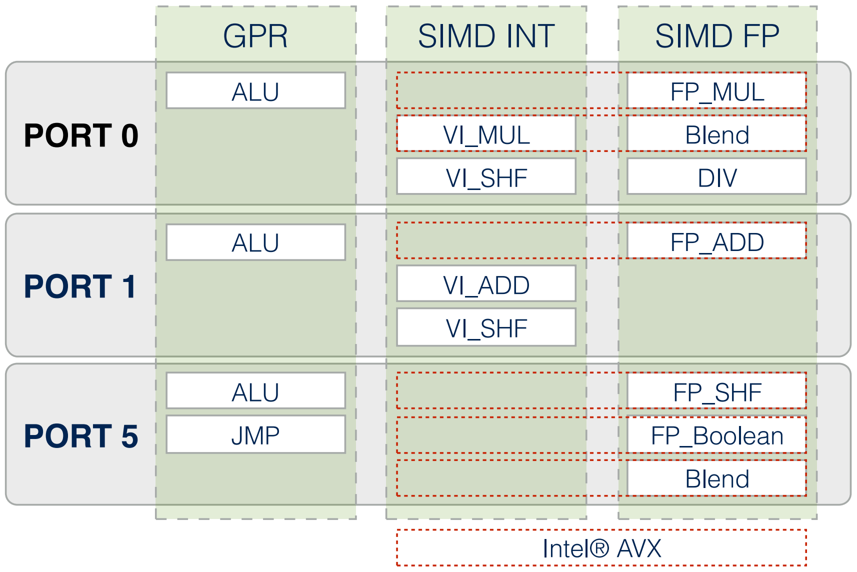


Figure 3.1: Execution cluster of Intel Sandy Bridge microarchitecture. Multiple operations are port-specific (*e.g.*, `FP_MUL` can only execute on port0).

stressors to study application *sensitivity* and *contentiousness* on various functional units for SMT co-locations. In addition to functional units, we also designed a set of memory stressors to study the interfering behavior on various levels of cache. The design principles and details behind these stressors (`Rulers`) are presented in Section 3.2.2.1.

3.1.2 Contention for Functional Units

Modern microarchitectures often include a large number of functional units to exploit instruction-level parallelism (ILP). As illustrated in Figure 3.1, each functional unit is usually designed to only execute certain types of operations. When investigating the interference due to sharing functional units between multiple hardware contexts co-located on an SMT core, we aim to answer the following questions:

- What is the amount of performance degradation caused by contention for each type of functional unit?
- Are applications' *sensitivity* and *contentiousness* for the same resource cor-

related? The answer to this question will indicate whether they need to be modeled separately.

- What is the variability of an application's contention characteristics across different functional units? The answer to this would indicate whether we need to characterize each shared resource separately or one single unified metric is sufficient for all resources.

- Do emerging data center workloads (*e.g.*, CloudSuite) behave differently from traditional workloads (*e.g.*, SPEC CPU2006) in terms of functional unit contention?

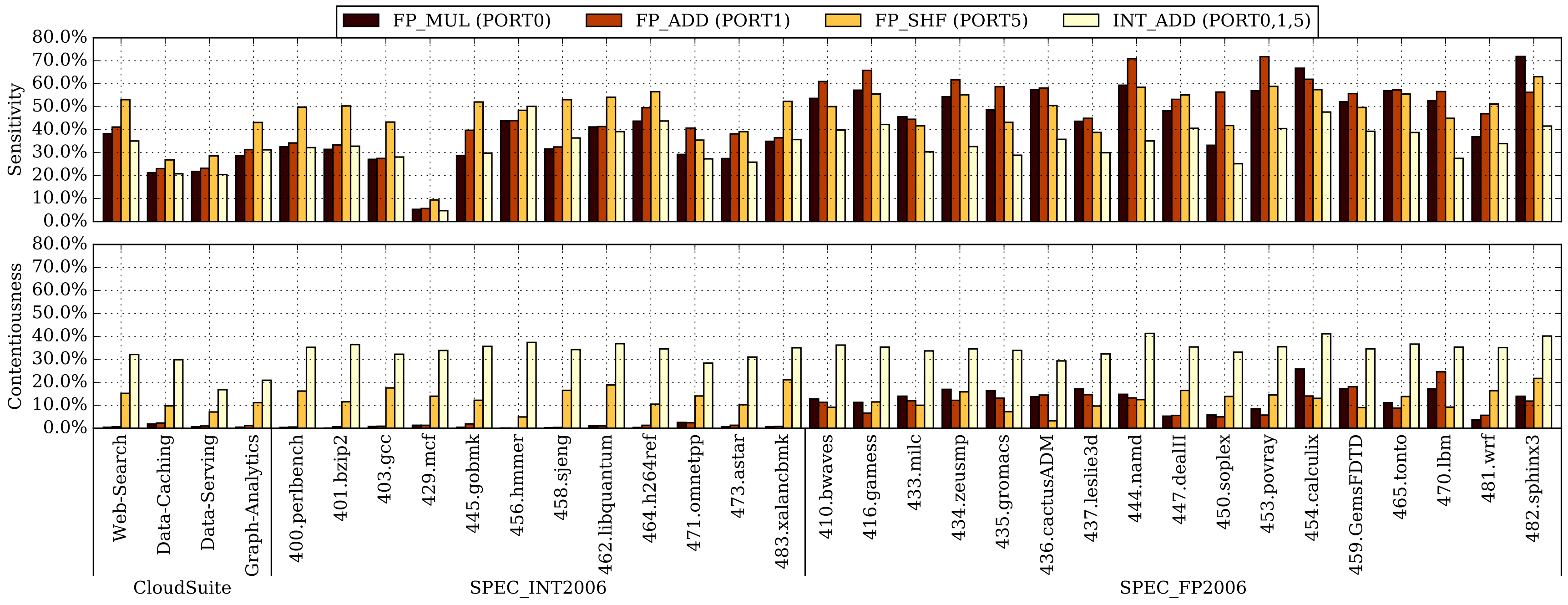


Figure 3.2: The *sensitivity* and *contentiousness* of different workloads on functional unit resources.

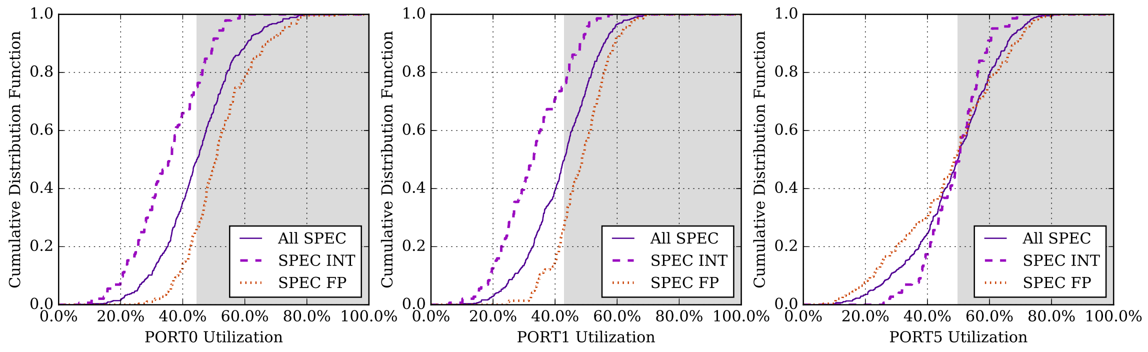


Figure 3.3: Aggregated functional unit utilization distributions across all the co-location pairs in SPEC CPU2006 for ports 0, 1 and 5.

Functional Unit Contentiousness and Sensitivity - Figure 3.2 shows applications' *sensitivity* and *contentiousness* measured by a set of `Rulers`, each `Ruler` maximizing the pressure in one specific functional unit resource, including `FP_MUL` at port 0, `FP_ADD` at port 1, `FP_SHF` at port 5 and `INT_ADD` spreading across ports 0, 1, 5. We quantify an application's *sensitivity* as the degradation it suffers from co-locating with `Rulers`, while *contentiousness* is defined as the degradation it causes to the `Rulers`. Our findings are as follows:

- *Finding 1. Applications in general are sensitive to functional unit contention.* As shown in Figure 3.2, applications suffer 5% - 70% performance degradation when contending for only one type of functional unit.
- *Finding 2. The level of sensitivity to contention for each functional unit varies across applications.* For example, `429.mcf` suffers 6% performance degradation due to port 1 contention, while `444.namd` suffers as high as 71% degradation.
- *Finding 3. Sensitivity and contentiousness of each application for each shared resource do not correlate with each other, and thus need to be captured separately.*
- *Finding 4. Each application has various levels of sensitivity and contentiousness for different functional units.* For example, `454.calculix` is more contentious

to port 0 while `470.lbm` is more contentious to port 1. *This suggests the need to capture the contention characteristics for each functional unit separately.*

- *Finding 5. Emerging data center workloads' contention behaviors for functional units are similar to SPEC_INT benchmarks.*

Due to the variability across applications and across each type of functional unit, we conclude that an ideal interference model needs to capture application *contentiousness* and *sensitivity* separately along each resource sharing dimension.

Functional Unit Utilization - In addition to the *sensitivity* and *contentiousness*, we also profile the utilization of various functional units when applications co-locate on an SMT core using hardware performance monitoring units (PMUs). Figure 3.3 presents the cumulative distribution function (CDF) for the utilization of ports 0, 1, and 5 respectively, across all pairs of co-located applications. In Figure 3.3, utilization is measured as the aggregated utilization of two co-located applications on an SMT core, where the shaded area illustrates the percentage of the co-located pairs that have higher utilization than the median. As shown in the figure, SPEC_FP benchmarks tend to have higher utilization for ports 0 and 1 than SPEC_INT. On the contrary, for port 5, SPEC_INT has higher utilization, and this is due to the higher branch instruction counts in SPEC_INT, which are executed on port 5.

- *Finding 6. Ports 0 and 1 have similar utilization distributions, which are distinctly different from the utilization distribution of port 5.* This also indicates that applications' contention behaviors at different functional units need to be measured separately to capture the variability across ports.

3.1.3 Interference in Memory Subsystem

In addition to functional units, private caches (L1 and L2), shared LLC and memory bandwidth are also shared among SMT contexts. Compared to CMP co-locations, sharing private caches adds additional complexity to the SMT co-locations.

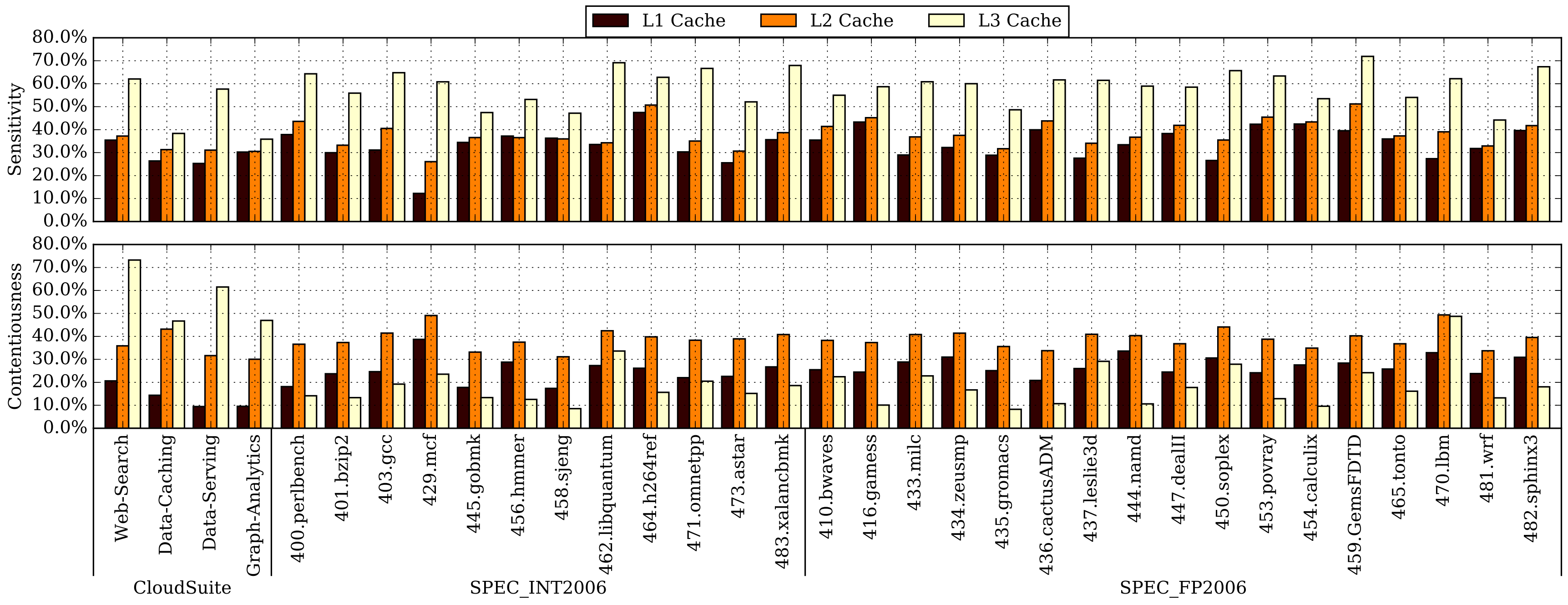


Figure 3.4: The *sensitivity* and *contentiousness* of different workloads on memory subsystem resources.

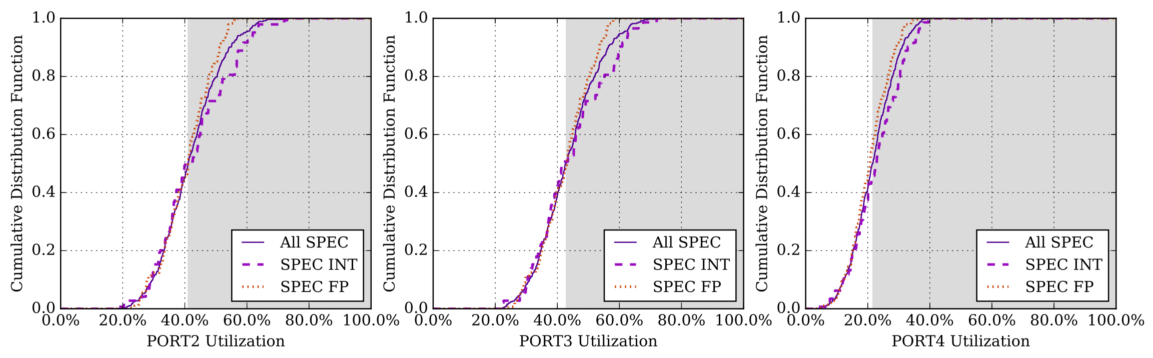


Figure 3.5: Aggregated memory port utilization distributions across all the co-location pairs in SPEC CPU2006.

Memory Subsystem Contentiousness and Sensitivity - Figure 3.4 presents application *sensitivity* and *contentiousness* measured using a set of memory `Rulers` co-located with the applications. We design our L1 and L2 cache `Rulers` to be the same binary with different working set sizes. As we increase the working set size, there is a monotonic increase in the `Ruler`'s performance impact.

- *Finding 7. Contention behaviors in the memory subsystem are more monolithic than functional units, demonstrating the basis for prior work to quantify the memory subsystem pressure using a unified metric. In addition, there is noticeable variability across applications.* Some applications' performance heavily relies on one specific cache level. For example, applications such as `454.calculix` have very similar *sensitivity* to contention in L1 and L2 caches, which indicates their high reliance on the L1 cache and low utilization of the L2 cache.

- *Finding 8. CloudSuite workloads are much more contentious at the L3 cache than SPEC applications, although they exhibit very similar levels of sensitivity across three levels of cache.*

Memory Port Utilization - Similarly, we also profile the aggregated utilization for memory ports across all the co-location pairs in SPEC as shown in Figure 3.5, in which port 2 and port 3 are used for memory loads and port 4 for memory stores. In

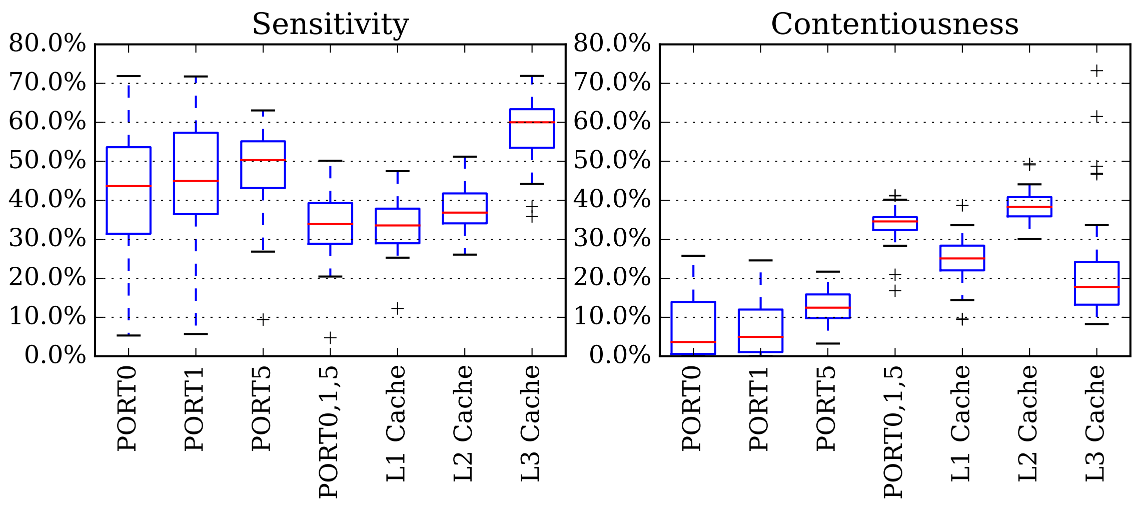


Figure 3.6: The *sensitivity* and *contentiousness* of all SPEC CPU2006 and CloudSuite applications. Applications' contention characteristics have a large variance both for the same resource and across different resources.

the figure, we find that memory store port (port 4) is heavily underutilized, compared to the load ports. This is supplementary to our finding 6 that applications' behaviors across ports vary, and thus need to be captured separately.

3.1.4 Correlation Among Sharing Dimensions

We summarize our measurement of application *sensitivity* and *contentiousness* for different resources in Figure 3.6. As illustrated in the figure, *there is a large variance in sensitivity and contentiousness in each sharing dimension across applications.* For example, application *sensitivity* to port 0 or port 1 ranges from negligible to above 70%. However, *on average, contention in each dimension can cause significant interference and thus all these dimensions need to be considered in our interference model.* In addition, *there is also a significant difference in terms of contention behaviors across different sharing dimensions.* For example, applications are more sensitive to contention for port 5 because branch instructions, which can only be executed on port 5, are critical for performance. In addition, most applications tend to generate more pressure on L2 than on L1 and L3 caches.

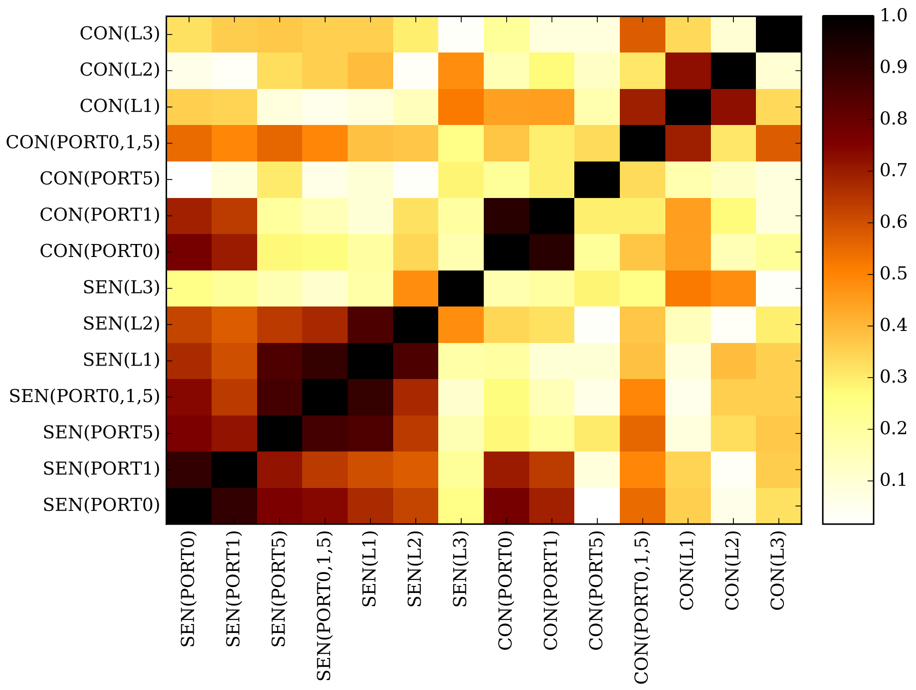


Figure 3.7: The absolute values of Pearson correlation coefficient among all the *sensitivity* and *contentiousness* dimensions. 97.96% of the pairs have a correlation coefficient lower than 0.80 and the majority of the pairs, lower than 0.50.

Furthermore, we quantify the correlation among different sharing dimensions using Pearson correlation coefficients. The absolute Pearson correlation coefficients of the *contentiousness* and *sensitivity* across all benchmarks in each of the 7 shared resources are presented in Figure 3.7. The absolute value of Pearson correlation coefficient ranges from 0 to 1, where 1 indicates the perfect correlation (both positive and negative) and 0 indicates no correlations.

- *Finding 9. As demonstrated in this figure, there is little correlation for application sensitivity and contentiousness among different sharing dimensions.* For example, as shown in the figure, an application being sensitive to contention for caches does not necessarily mean that it is less (or more) contentious or sensitive to contention for the floating-point functional unit. In fact, 97.96% of the pairs of sharing dimensions have a correlation coefficient lower than 0.80, and for the majority of the pairs the coefficient is lower than 0.50. These low

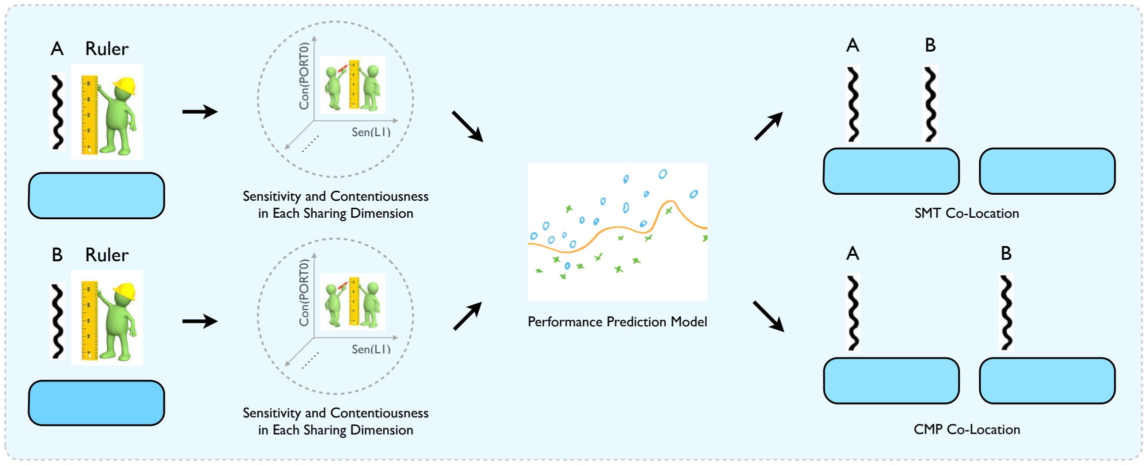

A
Ruler
Con(PORT0)
Sen(L1)
Sensitivity and Contentiousness in Each Sharing Dimension
B
Ruler
Con(PORT0)
Sen(L1)
Sensitivity and Contentiousness in Each Sharing Dimension
Performance Prediction Model
A
B
SMT Co-Location
A
B
CMP Co-Location

Figure 3.8: Overview of `SMiTe` methodology. Based on our insight that there is little correlation among applications' contention characteristics across multiple resource sharing dimensions, we design a set of `Rulers` to quantify an application's *sensitivity* and *contentiousness* in a decoupled manner (*e.g.*, in each sharing dimension). A regression-based prediction model is then established to use an application's *sensitivity* and *contentiousness* characterizations to make performance prediction for SMT and CMP co-locations.

correlations further suggest the necessity to decouple and measure each sharing dimension separately.

3.2 SMiTe Methodology

In this section, we present `SMiTe` methodology. Designed based on the findings summarized in Section 3.1, `SMiTe` enables precise performance prediction for SMT co-locations on real-system multicore processors.

3.2.1 Overview

The overview of the `SMiTe` methodology is presented in Figure 6.1, which consists of three main steps.

1. **Characterizing Sensitivity and Contentiousness (Section 3.2.2)** – For each application, we quantify its contention characteristics for shared SMT resources. A set of `Rulers` is designed to sense an application's *sensitivity* and

contentiousness along various sharing dimensions, including functional units and the memory subsystem. By co-locating the application of interest with a `Ruler`, we measure the application's performance degradation as its *sensitivity* to contention in the corresponding sharing dimension, and the degradation of the `Rulers` as the application's *contentiousness* in the same dimension.

2. **Performance Prediction Model (Section 3.2.3)** – To predict the performance interference between applications when they co-locate on an SMT core or CMP cores, we establish a regression-based prediction model, which combines each application's multidimensional characteristics quantified by the `Rulers` to precisely predict the performance degradation in both CMP and SMT co-location scenarios.

3. **Steering towards Safe Co-locations (Section 3.2.4)** – `SMiTe` allows us to quickly profile an application, and precisely predict the level of performance degradation that applications may suffer from the co-location. With this prediction ability, a cluster scheduler in a data center can identify 'safe' job co-locations that would not violate applications' QoS requirements, achieving high server utilization.

3.2.2 Quantifying Sensitivity and Contentiousness

In order to make precise performance predictions, we first characterize an application's *sensitivity* and *contentiousness.* Several key factors determine the characterization quality, including our `Rulers` design and the methodology to quantify the *sensitivity* and *contentiousness.*

3.2.2.1 Ruler Design

We design a set of `Rulers` to sense an application's interfering behavior in each sharing dimension in a decoupled manner. A good `Ruler` design needs to maximize the measurement accuracy while minimizing the profiling overhead. Here are two key principles that guide our `Ruler` design:

- **Each `Ruler` needs to maximize the pressure in the targeted sharing dimension while minimizing the impact in all other dimensions.** For example, a `Ruler` that targets port 0 needs to achieve maximum pressure on that port while minimizing its pressure on other functional units and the memory subsystem. As demonstrated in Figure 3.7, there is little correlation across all sharing dimensions. Therefore, minimizing the overlapping resources that each `Ruler` stresses helps decouple the interfering behaviors into independent dimensions.

- **A linear relationship between the intensity of the `Ruler` and the amount of interference it causes on the corresponding resource is desirable.** To characterize an application's *sensitivity* to contention for a given resource, we need to measure its performance degradation under a range of pressure intensities generated by the `Ruler`. Having a linear relationship between the intensity and the resulting interference is highly useful for reducing the profiling overhead. Instead of profiling the entire *sensitivity* curve by sampling the degradation under various intensity points, a linear relationship requires only two samples at both end points of the *sensitivity* curve.

It is very challenging to achieve these principles on real-system SMT processors due to the complexity in a commodity processor. Here we present our carefully designed `Rulers`.

```
loop:
        mulps       %xmm0, %xmm0
        ......
        mulps       %xmm7, %xmm7
        ......
        jmp loop
```

(a) FP_MUL (PORT0)

```
loop:
        addps       %xmm0, %xmm0
        ......
        addps       %xmm7, %xmm7
        ......
        jmp loop
```

(b) FP_ADD (PORT1)

```
loop:
        shufps      %xmm0, %xmm0
        ......
        shufps      %xmm7, %xmm7
        ......
        jmp loop
```

(c) FP_SHF (PORT5)

```
loop:
        addl        %eax, %eax
        ......
        addl        %edx, %edx
        ......
        jmp loop
```

(d) INT_ADD (PORT0,1,5)

```
#define MASK 0xd0000001u
#define RAND (lfsr = (lfsr >> 1) ^ (unsigned int)(0 - (lfsr & 1u) & MASK))
......
        while (1) {
                data_chunk[RAND % FOOTPRINT]++;
                ......
                data_chunk[RAND % FOOTPRINT]++;
        }
```

(e) MEM (L1, L2 Cache)

```
......
        first_chunk = data_chunk;
        second_chunk = data_chunk + FOOTPRINT / 2;
        while (1) {
                for (i = 0; i < FOOTPRINT / 2; i += 64) {
                        first_chunk[i] = second_chunk[i] + 1;
                }
                for (i = 0; i < FOOTPRINT / 2; i += 64) {
                        second_chunk[i] = first_chunk[i] + 1;
                }
        }
```

(f) MEM (L3 Cache)

Figure 3.9: Implementation of `Rulers`.

Functional Unit `Rulers` - As presented in Figure 3.9(a-d), in order to design decoupled `Rulers` that stress each resource independently, we design our functional unit `Rulers` using port-specific instructions [8] (see Figure 3.1). In addition, we remove all data dependencies between consecutive instructions and unroll the loops to maximize the functional unit utilization. By doing so, we achieve higher than 99.99% utilization for the targeted resource, validated using the hardware performance counters `UOPS_DISPATCHED_PORT:PORT0,1,5`. In addition, this design allows us to achieve the desirable linear relationship between the `Ruler` intensity and the interference it causes, because the intensity of our functional unit `Ruler` directly translates to the port utilization. Note that because specialized functional units are commonly used in modern processors, the design principle of the port-specific functional unit `Ruler` can be applicable to other microarchitectures such as IBM Power7 [142].

Memory Subsystem `Rulers` - Compared to functional unit `Rulers`, it is more difficult to completely decouple the interference in the memory subsystem because multiple levels of caches can be inclusive. In addition, to issue memory accesses, a certain amount of computation is unavoidable. Thus, we design our `Rulers` to maximize the pressure on the targeted cache level as an approximation, and rely on the regression-based prediction model to decouple the overlapping impact.

As shown in Figure 3.9(e), the L1 and L2 cache `Rulers` randomly access a chunk of data using a lightweight random number generator: linear-feedback shift register (LFSR). For the L3 cache `Ruler` as shown in Figure 3.9(f), we use stride access with a 64-byte offset, the size of the cache line, to maximize the amount of pressure. For both designs, we also unroll the loops to minimize the number of branch instructions. The intensity of our memory subsystem `Ruler` is defined as the working set size of each `Ruler`. We measure the average Pearson correlation coefficient between the working set size of our `Ruler` at each cache level and the performance degradation of all SPEC applications when co-located with the `Ruler`, and we observe strong linear correlations. The Pearson coefficients are 0.92 for L1, 0.89 for L2 and 0.95 for L3 cache. This linear relationship significantly reduces our profiling overhead, because the entire *sensitivity* curve for all working set sizes can be accurately approximated by interpolating between 3 `Rulers` whose working set sizes being the L1, L2 and L3 cache sizes.

3.2.2.2 Characterizing Contentiousness and Sensitivity

To quantify an application's *sensitivity* and *contentiousness*, we co-locate the application with the `Rulers` on the neighboring hardware context on an SMT core.

For each resource i, we measure the application A's performance degradation as its *sensitivity* Sen_i^A via the following equation:

$$Sen_i^A = \frac{IPC_{solo}^A - IPC_{co-location/Ruler_i}^A}{IPC_{solo}^A} \tag{3.1}$$

Similarly, we define application A's *contentiousness* Con_i^A as the corresponding `Ruler`'s performance degradation.

$$Con_i^A = \frac{IPC_{solo}^{Ruler_i} - IPC_{co-location/A}^{Ruler_i}}{IPC_{solo}^{Ruler_i}} \tag{3.2}$$

3.2.3 Performance Prediction Model

3.2.3.1 Prediction Model

After characterizing each application, to predict the performance degradation of application A when co-located with application B on an SMT core, we combine both A's *sensitivity* and B's *contentiousness* on each sharing dimension i, using a linear model. The prediction model is shown in Equation 3.3.

$$Deg_{co-locate/B}^A = \sum_i^N (c_i \times Sen_i^A \times Con_i^B) + c_0 \tag{3.3}$$

In this model, the degradation for A in each dimension is proportional to measured application A's *sensitivity* and the co-located application B's *contentiousness* on that dimension. The linear model reflects the assumption that an application's performance degradation from each shared dimension is additive. The amount (weight) that each sharing dimension contributes to the total performance degradation is captured by the coefficient c_i. The constant term c_0 is introduced to approximate the performance interference caused by other resources not captured in the model. A constant is used because the impact of other resources should have a small variance across applications, based our assumption that functional units and memory subsystem are the main contributors for the degradation.

3.2.3.2 Manage Prediction Error

There are two main sources of potential prediction errors. Firstly, the model can only capture the interference in a limited number of dimensions. Other shared resources such as the branch predictor might also cause performance interference, which are approximated by the constant c_0 in our model. Secondly, in order to reduce the profiling overhead, we take advantage of the approximately linear relationship between the intensity of a `Ruler` and the performance interference. This approximation might introduce errors in performance prediction. However, as we will show in our evaluation (Section 6.3), our model achieves high precision, demonstrating that the model has captured the significant resource dimensions.

3.2.3.3 Predicting Tail Latency

In addition to the average performance, many modern web service workloads in data centers have certain requirements on the percentile latency, often the tail latency [59]. For example, QoS requirements can be specified as 90% of the queries need to achieve under-100ms latency. In addition to service time, the time a query waits in the job queue before it gets processed also contributes to the latency. Thus, the percentile latency does not linearly correlate with the average performance due to this queueing effect, and needs to be modeled differently on top of the average performance prediction calculated using Equation 3.3.

To address this, we model the web service workload using a simple first-come first-served (FCFS) `M/M/1` queueing system [88], which has a closed-form solution. We use the `M/M/1` model based on two observations:

- Both the service time distribution and the inter-arrival distribution usually have small coefficients of variance in practice. This indicates that we can approximate these distributions using the exponential distribution and Poisson distribution, respectively, without losing much precision [86].

- The queueing and the processing usually happen at the same level (*e.g.*, a per thread queueing strategy often implies that each job in the queue is handled by one thread), which indicates that we can model the system with a single-server model [110]. For example, rather than having a global queue for all the worker threads, each thread has its own processing queue in `Memcached`. This allows us to model the response time distribution using the single-server model, because we are essentially just instantiating multiple copies of a single-server queueing system, one copy per worker thread.

In FCFS `M/M/1` queueing model, the response time probability density function (PDF), $f(t)$, can be modeled as shown in Equation 3.4, in which λ is the mean value of the arrival rate distribution $Poisson(\lambda)$, and μ is the average rate of the servicing time distribution $Exp(\mu)$.

$$f(t) = (\mu - \lambda)e^{-(\mu-\lambda)t} \tag{3.4}$$

Based on the average performance degradation (Deg) in Equation 3.3, we can extrapolate the degraded average service rate μ'.

$$\mu' = (1 - Deg)\mu \tag{3.5}$$

Taking the integral of the PDF in Equation 3.4, we can calculate the cumulative distribution function (CDF) of the response time. Using the inversion of the CDF and combining it with the degraded service time estimation in Equation 3.5, we estimate the p-th percentile latency t_p with Equation 3.6.

Table 3.1: Machine specifications in our experimental setup.

Processor	**Microarchitecture**	**Kernel**
Intel Xeon E5-2420 @ 1.90GHz	Sandy Bridge-EN	3.8.0
Intel i7-3770 @ 3.40GHz	Ivy Bridge	3.8.0

$$t_p = -\frac{ln(1-p)}{(1-Deg)\mu - \lambda} \tag{3.6}$$

3.2.4 SMiTe in Action

With the ability to precisely predict the average performance and percentile latency interference, `SMiTe` can identify 'safe' co-locations of applications so that the QoS interference for latency-sensitive applications due to co-locations is under a given threshold. The advantages of `SMiTe` over exhaustive pairwise offline profiling are twofold: 1) `SMiTe` characterizes each application individually once and uses the characterization for performance prediction. This allows the data center operators to avoid the complexity of cross-product characterization for all possible co-locating applications. Many latency-sensitive applications in data centers are long running and well suited for the type of profiling [122]. 2) In addition to much more efficient offline profiling, `SMiTe` has carefully controlled profiling complexity so that each application's characterization can be completed in the order of seconds. This allows us to conduct quick online profiling for any new application when it arrives at the cluster-level scheduler before getting scheduled to a suitable server.

3.3 Evaluation

3.3.1 Experimental Setup

We evaluate our `SMiTe` methodology on two commodity multicore SMT processors summarized in Table 3.1. The Linux `perf` tool is used to measure the hardware performance monitoring units (PMUs). We use CloudSuite [74] and SPEC CPU2006 [91] with `ref` inputs as our workloads. To construct the training and testing sets for our prediction model, we divide 29 SPEC benchmarks into 2 sets based on their even/odd numbering. Four applications from CloudSuite, including `Web-Search`, `Data-Caching`, `Data-Serving` and `Graph-Analytics`, are used to represent latency-sensitive workloads in modern data centers.

Throughout this section, the performance degradation caused by the co-location $Deg_{co-location}$ is defined as in Equation 3.7, where IPC_{solo} is the instructions per cycle (IPC) measurement when the application is running alone and $IPC_{co-location}$ is the IPC when co-located with other applications.

$$Deg_{co-location} = \frac{IPC_{solo} - IPC_{co-location}}{IPC_{solo}} \tag{3.7}$$

The performance prediction error is reported as the absolute error between the measured performance degradation and the predicted degradation as shown in Equation 3.8.

$$Error = \left| Deg^{predicted}_{co-location} - Deg^{actual}_{co-location} \right| \tag{3.8}$$

3.3.2 Performance Interference Prediction

3.3.2.1 SMiTe Prediction for General Purpose Workloads

We first investigate the prediction accuracy of `SMiTe` using SPEC benchmarks on an Intel Ivy Bridge server. All even-numbered benchmarks are used as training set for the performance prediction model, and odd-numbered benchmarks as testing set. The model is trained using the *sensitivity* and *contentiousness* measurements of each application as well as the performance degradation profiling of each co-locating pairs in the training set. Then the trained prediction model takes application's *sensitivity* and *contentiousness* from the testing set to predict the performance degradation of co-locating pairs.

In this experiment, we also compare our `Ruler` based prediction model against PMU based models. PMU based models have been commonly used for scheduling optimization [73] and power modeling [39] on SMT processors. Since there is no prior work providing techniques for precise performance prediction on real-system SMT processors, we carefully designed several PMU based model for predicting performance and selected the best one as our baseline to evaluate the viability of a PMU based model. Specifically, after experimenting with a number of PMUs and various regression strategies including linear regression, decision tree, higher order polynomial regression, we found the best performing model to be a linear regression model using 11 PMU measurements: *instructions/cycle*, *iTLB-misses/cycle*, *dTLB-load-misses/cycle*, *dTLB-store-misses/cycle*, *i-cache-misses/cycle*, *L1D-hits/cycle*, *L2-hits/cycle*, *L2-misses/cycle*, *L3-hits/cycle*, *MEM-hits/cycle*, *branch-mispredictions/cycle*. The linear regression is established using Equation 3.9 to predict the performance degradation on SMT and CMP co-locations.

$$Deg^{A}_{co-locate/B} = \sum_{i}^{N}(c_i^A PMU_i^A + c_i^B PMU_i^B) + c_0 \qquad (3.9)$$

The prediction accuracy of `SMiTe` and the PMU based model is reported in Figure 3.10 for SMT co-locations and Figure 3.11 for CMP co-locations. As shown in the figure, the average measured performance degradations of each benchmark when co-located span a wide range, from 11.74% to 53.14%. In the figure, the bars labeled as PMU Prediction Error present the average prediction error of the PMU based

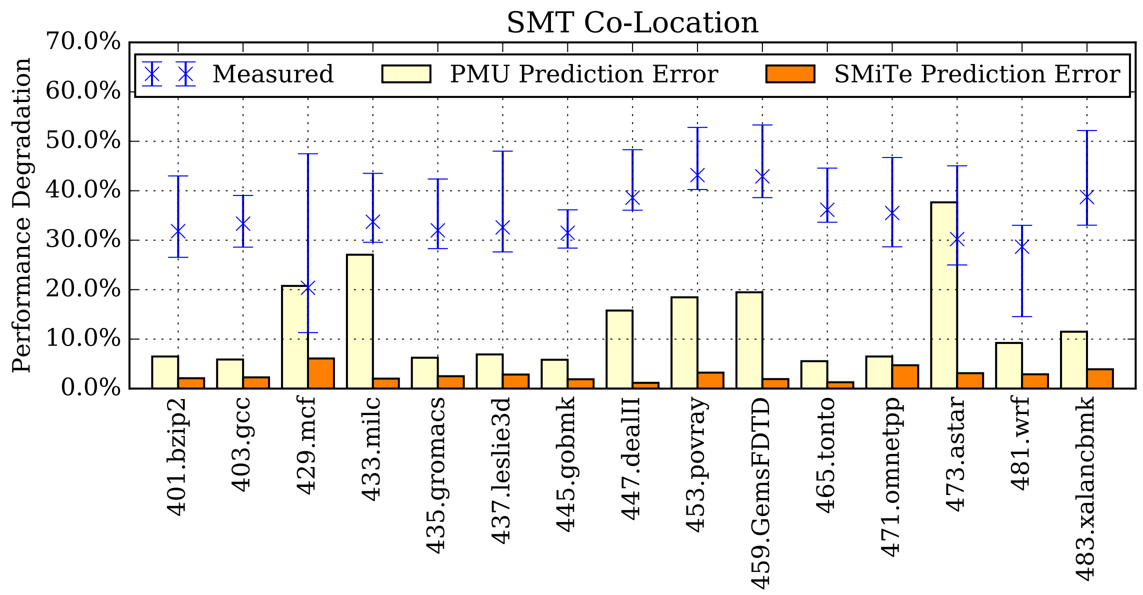


Figure 3.10: Performance prediction accuracy for SMT co-location on SPEC CPU2006 benchmarks, where the average prediction error of PMU based approach is 13.55% and `SMiTe` is 2.80%.

prediction model when each benchmark co-locates with all the other benchmarks in the testing set. The average error for the PMU based model is 13.55% for SMT co-locations and 9.43% for CMP co-locations. Compared to PMU based approach, `SMiTe` provides significantly higher precision, predicting both SMT and CMP co-locations with an average error of 2.80%.

3.3.2.2 SMiTe Prediction for Cloud Workloads

In this section, we evaluate our methodology on CloudSuite benchmarks, which are used to represent latency-sensitive applications running in modern data centers.

In contrast to the SPEC workloads, CloudSuite applications are usually multi-threaded and span more than one core. Thus, we set up this experiment differently on our Sandy Bridge-EN machine, which has 6 cores with 12 SMT hardware contexts on each socket. To half load the server as a baseline, we configure the cloud applications to run with 6 threads for SMT co-location experiment such that each core has one SMT context busy and the other one idle. Similarly, 3 threads are used for the

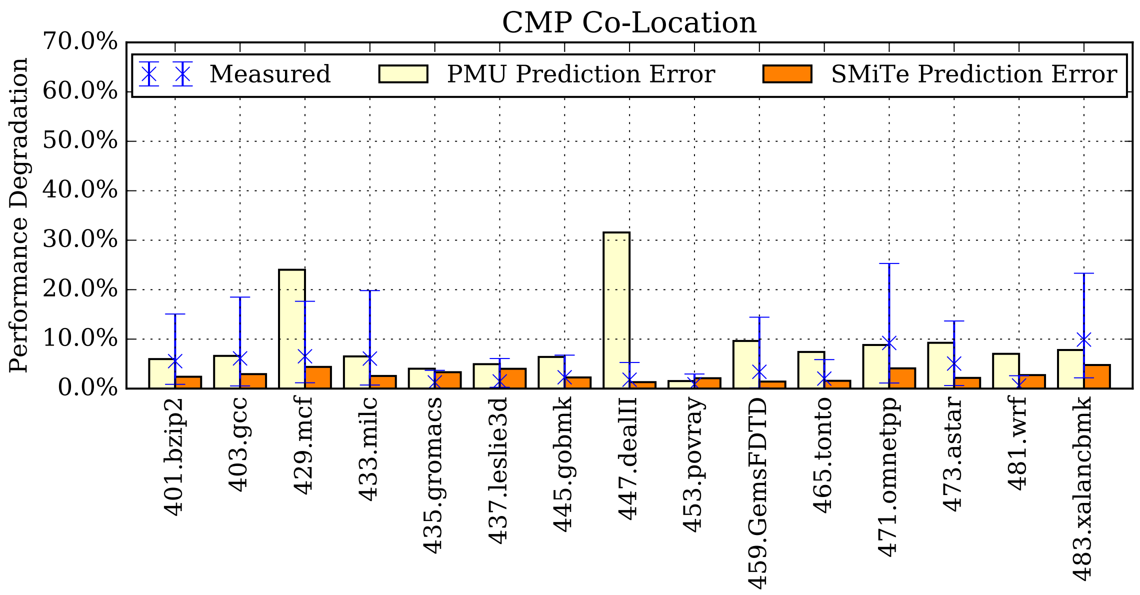


Figure 3.11: Performance prediction accuracy for CMP co-location on SPEC CPU2006 benchmarks, where the average prediction error of PMU based approach is 9.43% and `SMiTe` is 2.80%.

CMP co-location experiment with 3 out of 6 cores are left completely idle as the baseline. Accordingly, we use 6 instances of the same `Ruler` for SMT experiment and 3 instances for CMP experiment when measuring the *sensitivity* and *contentiousness* of the cloud applications. We use odd-numbered benchmarks from SPEC as the training set and even-numbered benchmarks as the testing set. Both PMU based and `SMiTe` prediction models are trained using the SPEC training set, and tested on co-locations between CloudSuite applications as latency-sensitive applications and SPEC testing set as batch applications.

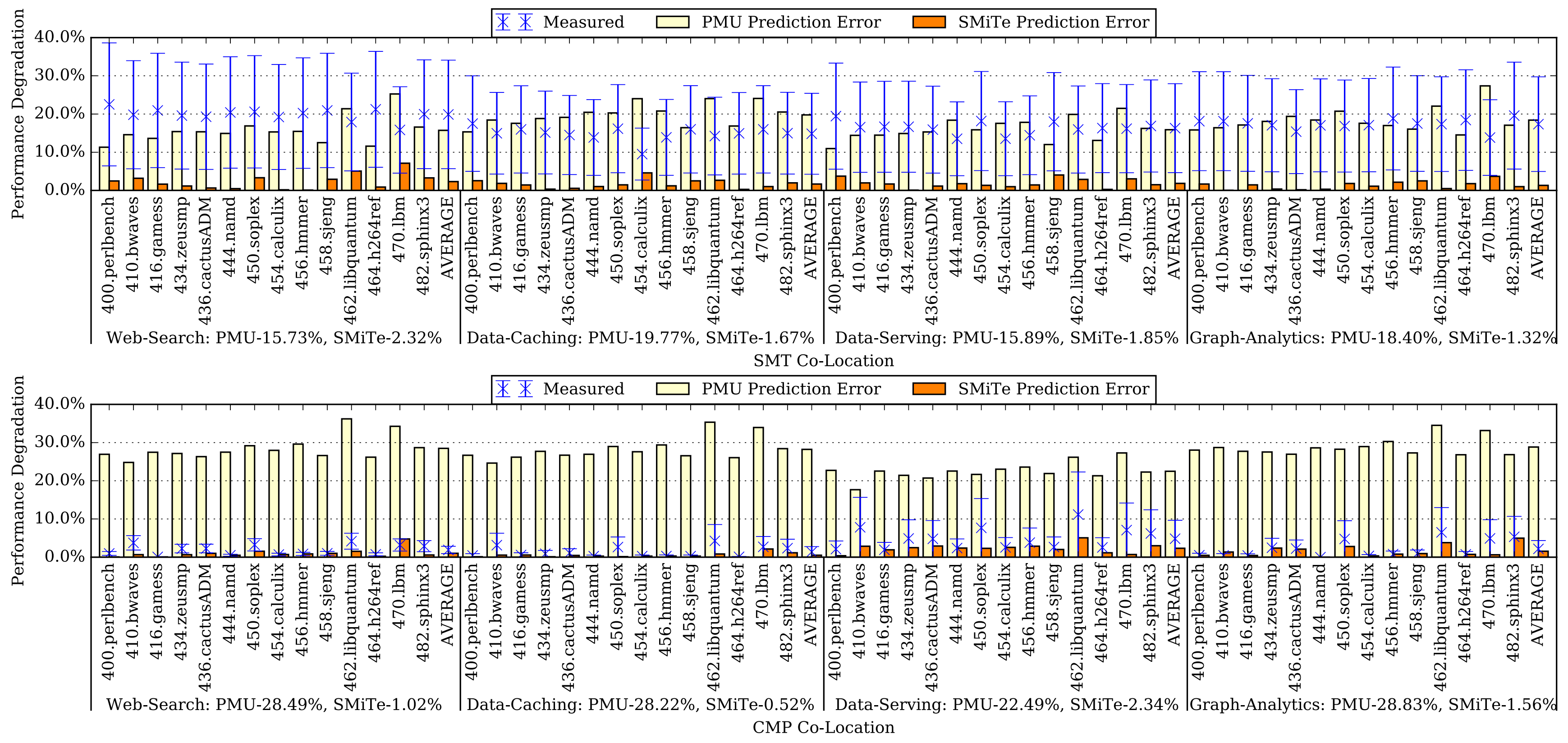


Figure 3.12: Performance prediction accuracy for SMT and CMP co-location on CloudSuite benchmarks (`Web-Search`, `Data-Caching`, `Data-Serving` and `Graph-Analytics`), where the average prediction error of PMU based approach is 17.45% for SMT co-location, 27.01% for CMP co-location and `SMiTe` is 1.79% and 1.36% respectively.

The prediction errors for both `SMiTe` and PMU based approaches are presented in Figure 3.12. The bars labeled as Measured present the maximum, average and minimum measured performance degradation, ranging from co-locating with 1 instance to 6 instances of the batch applications (x-axis) for SMT co-locations, and 1 to 3 instances for CMP co-locations. As shown in the figure, the PMU based model has an average prediction error of 17.45% for SMT co-locations and 27.01% for CMP co-locations, while `SMiTe` can precisely predict the performance degradation with 1.79% and 1.36% average errors respectively.

As demonstrated by our experiments, the PMU based prediction model performs poorly on both SPEC and CloudSuite applications. We observed a few possible sources that may contribute to the inaccuracy:

- Some PMUs are designed to be core counters, *e.g.*, `UOPS_EXECUTED.PORT2_CORE`, and there are no counters available to measure the corresponding events at per SMT context granularity [85].
- Some PMUs are known to contain bugs and may report inaccurate measurements [66].
- There are limited numbers of PMUs available on the real system, and they may not fully expose the resource usage information that is needed for the precise prediction.

3.3.2.3 SMiTe's Prediction Accuracy for Tail Latency

We evaluate our prediction model for 90th percentile latency using `Web-Search` and `Data-Caching` (`Data-Serving` and `Graph-Analytics` do not report percentile latency statistics). We use the profiled performance degradation and 90th percentile latency of CloudSuite application when co-located with `Rulers` to train our latency prediction model using Equation 3.6. In Figure 3.13, the measured performance degrada-

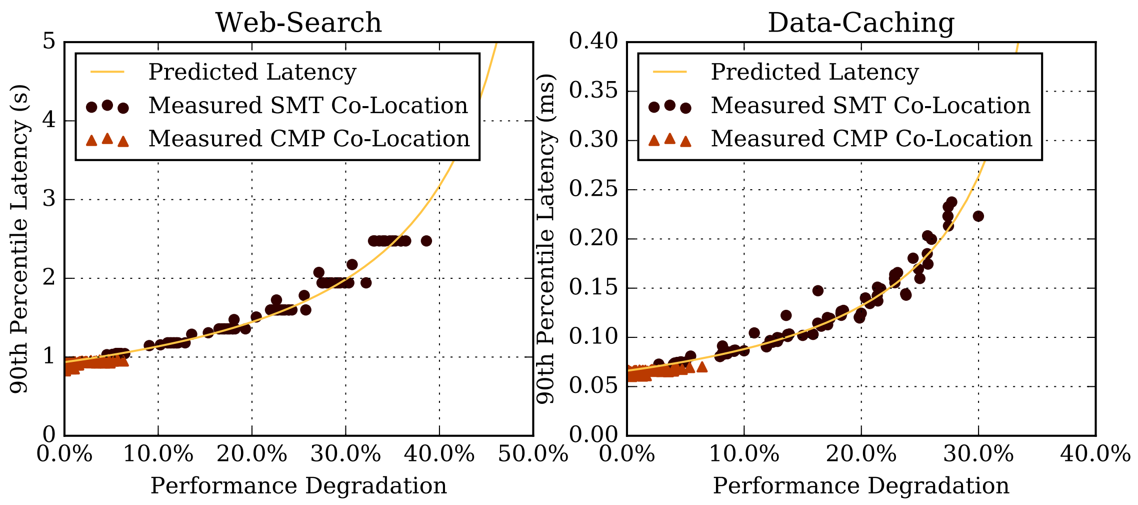


Figure 3.13: Prediction accuracy for 90th percentile latency when latency-sensitive application represented by CloudSuite co-locate with batch applications. The average absolute prediction error on `Web-Search` is 4.61% and 6.17% for `Data-Caching`.

tion and 90th percentile latency are measured when `Web-Search` and `Data-Caching` are co-located with applications from the SPEC testing set. The figure demonstrates that our queueing model is able to capture the correlation between the performance degradation and the 90th percentile latency. The average prediction error of our model is 4.61% for `Web-Search` and 6.17% for `Data-Caching`.

3.3.3 Scale-out Study: Improving Utilization while Guaranteeing QoS

With `SMiTe`'s precise prediction, we can enable 'safe' co-locations in order to improve utilization without violating the QoS requirement. In this experiment, we assume a cluster composed of 4,000 servers and each 1,000 of them are running one of the four latency-sensitive applications from CloudSuite (`Web-Search`, `Data-Caching`, `Data-Serving` and `Graph-Analytics`). We use our performance prediction model to guide the cluster-level scheduler to co-locate latency-sensitive applications with batch SPEC applications. Our evaluation baseline disallows SMT co-locations, which is the state-of-the-art approach to guarantee QoS in modern data centers without a precise prediction mechanism, leaving one out of the two SMT contexts on each core idle.

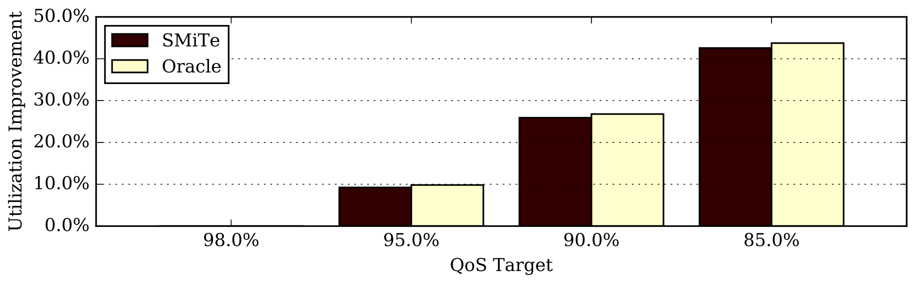


Figure 3.14: Utilization improvement when we allow SMT co-location under different QoS targets defined as average performance. `SMiTe` improves the utilization by 9.24%, 25.90% and 42.97%, at 95%, 90% and 85% QoS target respectively, which is very close to the Oracle co-location policy as 9.82%, 26.78% and 43.75%.

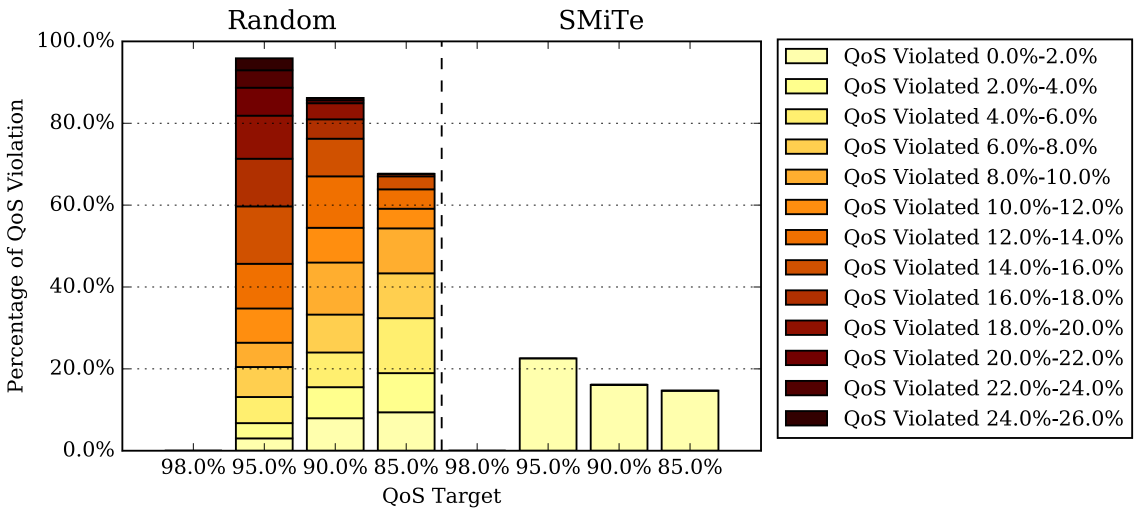


Figure 3.15: Percentage of QoS violation in all scheduled co-locations under `SMiTe` co-location policy and Random co-location policy when QoS is defined as average performance. In order to achieve same amount of utilization gain, Random co-location policy violates up to 26% QoS requirement while the largest violation from `SMiTe` is only 1.67%.

Thus, we have 6 latency-sensitive application threads running on 6 cores, and we could potentially co-locate from 0 to 6 instances of batch applications on each server. In addition to `SMiTe`, we measure application's actual performance degradation and use these measurements to construct an Oracle co-location policy for comparison.

Figure 3.14 shows the utilization improvement when applying different co-location policies. `SMiTe` achieves 9.24%, 25.90% and 42.97% utilization improvement at 95%, 90% and 85% QoS targets respectively. Compared to the Oracle policy, which im-

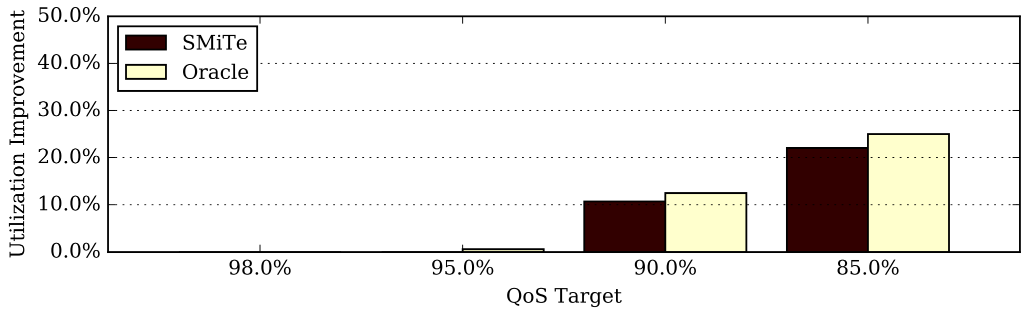


Figure 3.16: Utilization improvement when we allow SMT co-location under different QoS targets defined as 90th percentile latency. `SMiTe` improves the utilization by 0%, 10.72% and 22.03% at 95%, 90% and 85% QoS target respectively, which is relatively close to the Oracle co-location policy as 0.59%, 12.50% and 24.99%.

proves the utilization by 9.82%, 26.78% and 43.75%, `SMiTe` is very efficient and achieves utilization that is very close to the Oracle.

Due to the potential inaccuracy the prediction model has, in rare cases, the co-location decisions made by `SMiTe` might slightly violate the targeted QoS requirement. We quantify the violations compare it against an interference-oblivious policy that achieves exactly the same amount of utilization gain through randomly co-locating applications. In this experiment, the percentage of QoS violations is defined as the number of violations divided by the number of co-locations ($\frac{server_{violated}}{server_{co-located}}$), and the amount of violations is defined as the normalized violation ($\frac{QoS_{target}-QoS_{actual}}{QoS_{target}}$).

The QoS violations are shown in Figure 3.15. To achieve the same amount of utilization gain as `SMiTe` at each QoS target, the Random policy suffers from up to 26% QoS violation while the biggest violation using `SMiTe` is only 1.67%. In addition, as shown in the figure, `SMiTe` reduces 78.57% QoS violations on average compared to the Random policy.

3.3.4 Scale-out Study: Tail Latency

In this section, we evaluate the utilization improvement and QoS when the cluster-level scheduler uses `SMiTe`'s prediction for tail latency to steer scheduling. Similar

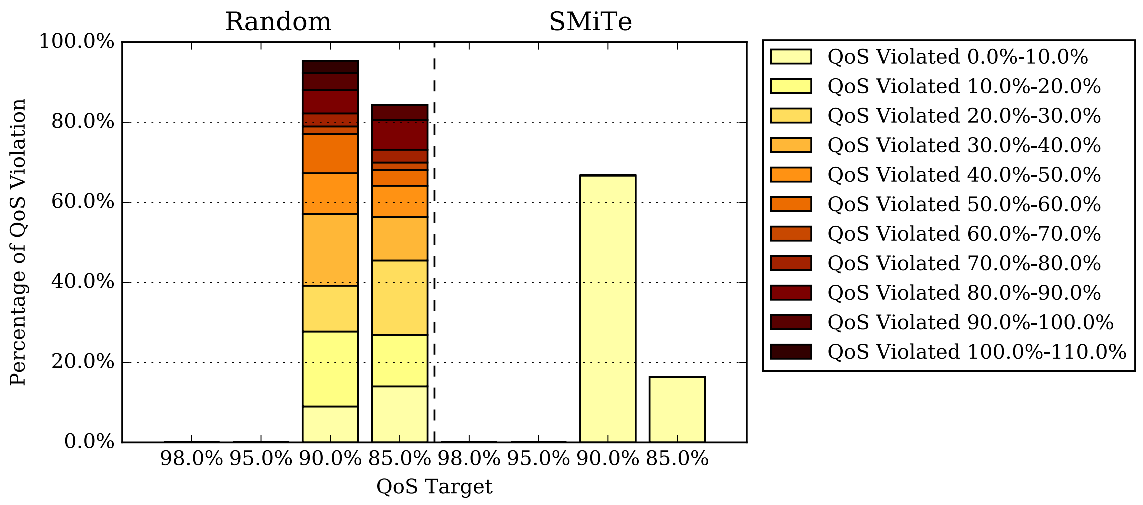


Figure 3.17: Percentage of QoS violation in all scheduled co-locations under `SMiTe` and Random co-location policy when QoS is defined as 90th percentile latency. To improve the same amount of server utilization, Random policy suffers from up to 110% QoS violation while the largest violation from `SMiTe` is only 0.96%.

to the previous scale-out experiment, we assume a cluster composed of 4,000 machines with half-loaded latency-sensitive workloads composed of `Web-Search` and `Data-Caching`. The QoS requirement in this experiment is defined as the 90th percentile latency, which is more challenging to meet than the average performance. This is because the tail latency grows super-linearly with the average performance degradation due to the queueing effect. However, `SMiTe` is able to achieve 10.72% utilization improvement at 90% QoS requirement and 22.03% at 85% QoS requirement (90th percentile query latency is affected by 10% and 15% respectively), which is relatively close to the Oracle policy of 12.50% and 24.99% improvement as shown in Figure 3.16. In addition, compared to the Random policy shown in Figure 3.17, which suffers up to 110% QoS violations, the most serious violation `SMiTe` experiences is only 0.96%.

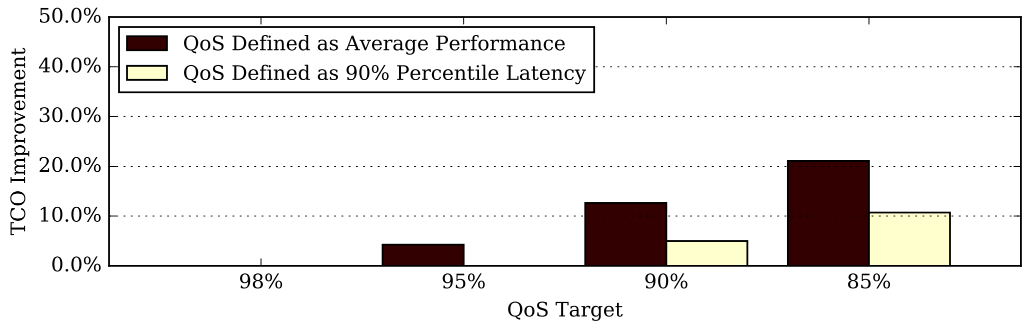


Figure 3.18: Total cost of ownership (TCO) improvement under different QoS requirements normalized by disallowing SMT co-location. `SMiTe` can save up to 21.05% cost under average performance requirement and up to 10.70% under 90th percentile latency requirement.

3.3.5 TCO Analysis

By improving the server utilization through co-locations, we improve the energy efficiency and also reduce the total cost of ownership (TCO) for building and operating the data centers. Because we can provide the same amount of computation with fewer servers through co-locations, we reduce the number of servers needed, the required power provisioning, the data center area and the maintenance expenses consequently. In this section, we quantify the TCO saving by applying `SMiTe` methodology in data centers under various QoS requirements.

In the baseline configuration, we assume the data center has half of the machines running latency-sensitive applications and the other half running batch applications. By applying `SMiTe` methodology, we can co-locate batch applications together with latency-sensitive applications on the same server if the QoS requirement can be met based on the prediction. The analytical methodology introduced in [35] is applied to study the impact of `SMiTe` on the 3-year TCO. We use the latest PUE statistics published by Google [6] as part of the input to the TCO model.

Figure 3.18 presents the results of our TCO analysis. `SMiTe` improves the TCO by up to 21.05% when targeting the average performance QoS requirement. Although 90th percentile QoS requirement is more challenging because the tail latency grows super-linearly due to queueing effect, `SMiTe` still achieves up to 10.70% improvement.

3.4 Summary

In this Chapter, we present `SMiTe` methodology, which enables precise performance interference prediction on multicore SMT processors. Based on our observation that there is very little correlation for an application's contention characteristics across different shared resources, we design a set of `Rulers` to quantify application's *sensitivity* and *contentiousness* in a decoupled manner. We then establish a regression model that combines the *sensitivity* and *contentiousness* measurements to predict the performance interference under various co-location scenarios. With `SMiTe`, we are able to predict SMT co-locations with 2.80% average error for SPEC CPU2006 benchmarks and 1.79% average error on CloudSuite. Based on the precise performance prediction our methodology provides, we can improve the server utilization by up to 42.57% through co-locations while enforcing the QoS requirement.

CHAPTER IV

Harvest: History-Based Resource Harvesting in Data Centers

An effective way to increase utilization and reduce costs in data centers is to co-locate their latency-critical services and batch workloads. In this Chapter, we describe systems that harvest spare compute cycles and storage space for co-location purposes. The main challenge is minimizing the performance impact on the services, while accounting for their utilization and management patterns. To overcome this challenge, we propose techniques for giving the services priority over the resources, and leveraging historical information about them. Based on this information, we schedule related batch tasks on servers that exhibit *similar* patterns and will likely have enough available resources for the tasks' durations, and place data replicas at servers that exhibit *diverse* patterns. We characterize the dynamics of how services are utilized and managed in ten large-scale production data centers. Using real experiments and simulations, we show that our techniques eliminate data loss and unavailability in many scenarios, while protecting the co-located services and improving batch job execution time.

4.1 Characterizing Behavior Patterns

We now characterize the primary tenants in ten production data centers. In later sections, we use the characterization for our co-location techniques and results.

4.1.1 Data sources and terminology

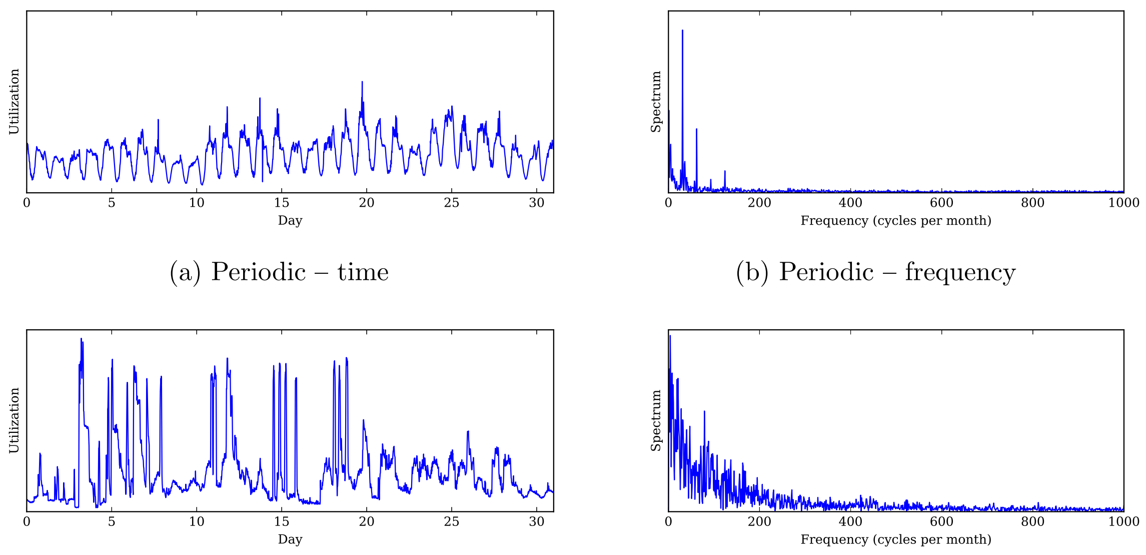
We leverage data collected by AutoPilot [97], the primary tenant management and deployment system used in the data centers. Under AutoPilot, each server is part of an *environment* (a collection of servers that are logically related, *e.g.* indexing servers of a search engine) and executes a *machine function* (a specific functionality, *e.g.* result ranking). Environments can be used for production, development, or testing. In our terminology, each primary tenant is equivalent to an <environment, machine function> pair. Primary tenants run on physical hardware, *without* virtualization. Each data center has between a few hundred to a few thousand primary tenants.

Though our study focuses on AutoPilot-managed data centers, our characterization and techniques should be easily applicable to other management systems as well. In fact, similar telemetry is commonly collected in other production data centers, *e.g.* GWP [133] at Google and Scuba [24] at Facebook.

4.1.2 Resource utilization

AutoPilot records the primary tenant utilization per server for all hardware resources, but for simplicity we focus on the CPU in this paper. It records the CPU utilization every two minutes. As the load is not always evenly balanced across all servers of a primary tenant, we compute the average of their utilizations in each time slot, and use the utilization of this *"average"* server for one month to represent the primary tenant.

We then identify trends in the tenants' utilizations, using signal processing. Specifically, we use the Fast Fourier Transform (FFT) on the data from each primary tenant



(a) Periodic – time

(b) Periodic – frequency

(c) Unpredictable – time

(d) Unpredictable – frequency

Figure 4.1: Sample periodic and unpredictable one-month traces in the time and frequency domains.

individually. The FFT transforms the utilization time series into the frequency domain, making it easy to identify any periodicity (and its strength) in the series.

We identify three main classes of primary tenants: *periodic*, *unpredictable*, and (roughly) *constant*. Figure 4.1 shows the CPU utilization trends of a periodic and an unpredictable primary tenant in the time and frequency domains. Figure 4.1b shows a strong signal at frequency 31, because there are 31 days (load peaks and valleys) in that month. In contrast, Figure 4.1d shows a decreasing trend in signal strength as the frequency increases, as the majority of the signal derives from events that rarely happen (*i.e.*, exhibit lower frequency).

As one would expect, user-facing primary tenants often exhibit periodic utilization (*e.g.*, high during the day and low at night), whereas non-user-facing (*e.g.*, Web crawling, batch data analytics) or non-production (*e.g.*, development, testing) primary tenants often do not. For example, a Web crawling or data scrubber tenant may exhibit (roughly) constant utilization, whereas a testing tenant often exhibits unpredictable utilization behavior.

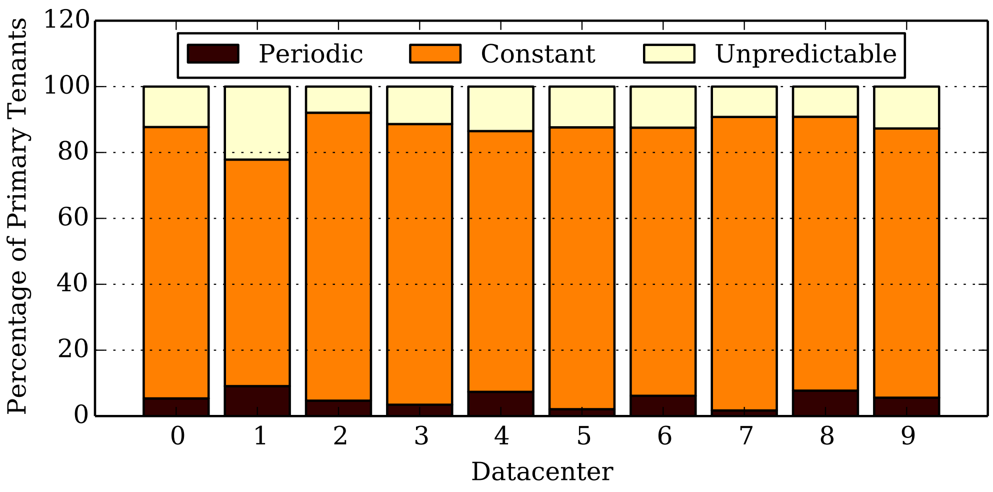


Figure 4.2: Percentages of primary tenants per class.

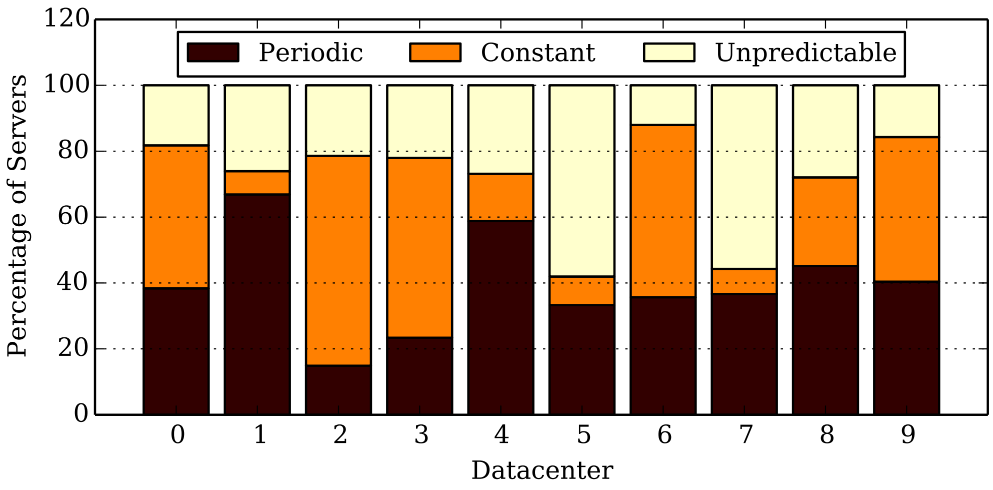


Figure 4.3: Percentages of servers per class.

More interestingly, Figure 4.2 shows that user-facing (periodic) primary tenants are actually a small minority. The vast majority of primary tenants exhibit roughly constant CPU utilization. Nevertheless, Figure 4.3 shows that the periodic primary tenants represent a large percentage (~40% on average) of the servers in each data center. Still, the non-periodic primary tenants account for more than half of the tenants and servers.

Most importantly, the vast majority of servers (~75%) run primary tenants (periodic and constant) for which the historical utilization data is a good predictor of future behaviors (the utilizations repeat periodically or all the time). Thus, leveraging this data should improve the quality of both our task scheduling and data placement.

4.1.3 Disk reimaging

Disk reimages are relatively frequent for some primary tenants, which by itself potentially threatens data durability under co-location. Even worse, disk reimages are often correlated, *i.e.* many servers might be reimaged at the same time (*e.g.*, when servers are repurposed from one primary tenant to another). Thus, it is critical for data durability to account for reimages and correlations.

AutoPilot collects disk reimaging (reformatting) data per server. This data includes reimages of multiple types: (1) those initiated manually by developers or service operators intending to re-deploy their environments (primary tenants) or restart them from scratch; (2) those initiated by AutoPilot to test the resilience of production services; and (3) those initiated by AutoPilot when disks have undergone maintenance (*e.g.*, tested for failure).

We now study the reimaging patterns using three years of data from AutoPilot. As an example of the reimaging frequencies we observe, Figure 4.4 shows the Cumulative Distribution Function (CDF) of the average number of reimages per month for each server in three years in five representative data centers in our sample. Figure 4.5 shows the CDF of the average number of reimages per server per month for each primary tenant for the same years and data centers. The discontinuities in this figure are due to short-lived primary tenants.

We make three observations from these figures. First and most importantly, there is a good amount of diversity in average reimaging frequency across primary tenants in each data center (Figure 4.5 does not show nearly vertical lines). Second, the reimaging frequencies per month are fairly low in all data centers. For example, at least 90% of servers are reimaged once or fewer times per month on average, whereas

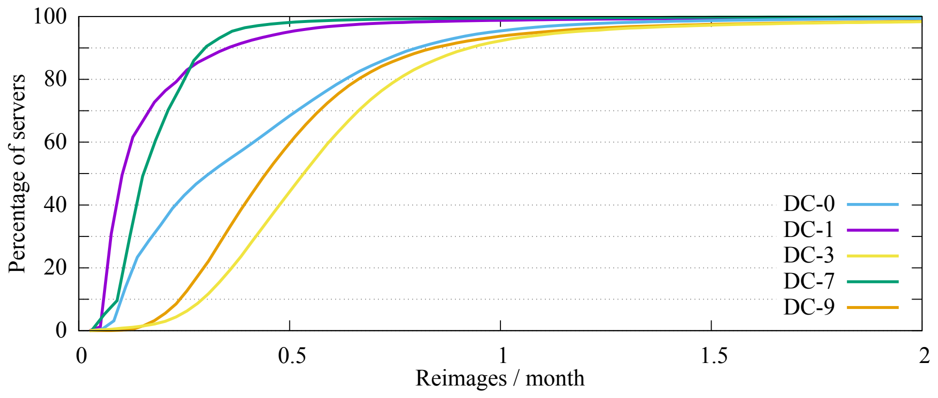


Figure 4.4: Per-server number of reimages in three years.

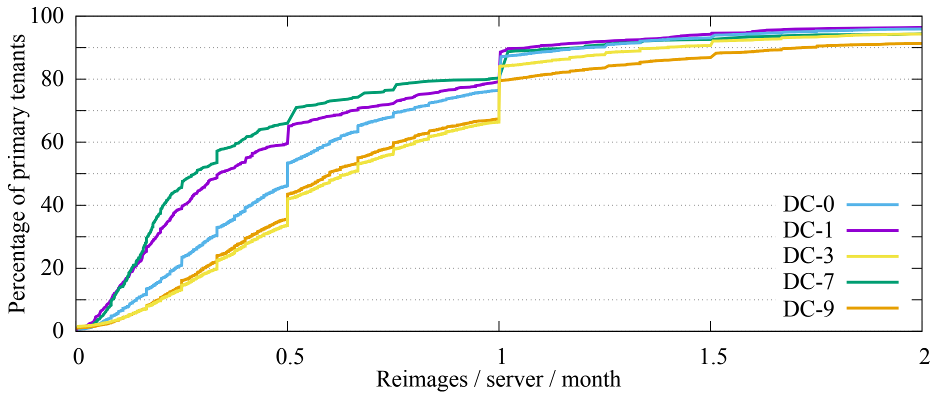


Figure 4.5: Per-tenant number of reimages in three years.

at least 80% of primary tenants are reimaged once or fewer times per server per month on average. This shows that reimaging by primary tenant engineers and AutoPilot is not overly aggressive on average, but there is a significant tail of servers (10%) and primary tenants (20%) that are reimaged relatively frequently. Third, the primary tenant reimaging behaviors are fairly consistent across data centers, though three data centers show substantially lower reimaging rates per server (we show two of those data centers in Figure 4.4).

The remaining question is whether each primary tenant exhibits roughly the same frequencies month after month. In this respect, we find that there is substantial variation, as frequencies sometimes change substantially.

Nevertheless, when compared to each other, primary tenants tend to rank consistently in the same part of the spectrum. In other words, primary tenants that

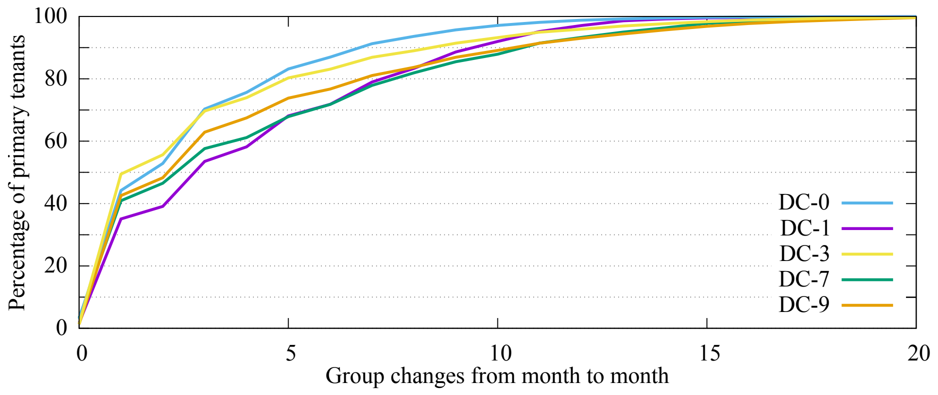


Figure 4.6: Number of times a primary tenant changed reimage frequency groups in three years.

experience a relatively small (large) number of reimages in a month tend to experience a relatively small (large) number of reimages in the following month. To verify this trend, we split the primary tenants of a data center into three frequency groups, each with the same number of tenants: *infrequent*, *intermediate*, and *frequent*. Then, we track the movement of the primary tenants across these groups over time. Figure 4.6 plots the CDF of the number of times a primary tenant changed groups from one month to the next. At least 80% of primary tenants changed groups only 8 or fewer times out of the possible 35 changes in three years. This behavior is also consistent across data centers.

Again, these figures show that historical reimaging data should provide meaningful information about the future. Using this data should improve data placement.

4.2 Smart Co-location Techniques

In this section, we describe our techniques for smart task scheduling and data placement, which leverage the primary tenants' historical behavior patterns.

4.2.1 Smart task scheduling

We seek to schedule batch tasks (secondary tenants) to harvest spare cycles from servers that natively run interactive services and their supporting workloads (primary tenants). Modern cluster schedulers achieve high job performance and/or fairness, so they are good candidates for this use. However, their designs typically assume dedicated servers, *i.e.* there are no primary tenants running on the same servers. Thus, we must (1) modify them to become aware of the primary tenants and the primary tenants' priority over the servers' resources; and (2) endow them with scheduling algorithms that reduce the number of task killings resulting from the co-located primary tenants' need for resources. The first requirement is fairly easy to accomplish, so we describe our implementation in Section 4.3. Here, we focus on the second requirement, *i.e.* smart task scheduling, and use historical primary tenant utilization data to select servers that will most likely have the required resources available throughout the tasks' entire executions.

Due to the sheer number of primary tenants, it would be impractical to treat them independently during task scheduling. Thus, our scheduling technique first clusters together primary tenants that have similar utilization patterns into the same utilization *class*, and then select a class for the tasks of a job. Next, we discuss our clustering and class selection algorithms in turn.

The **clustering** algorithm periodically (*e.g.*, once per day) takes the most recent time series of CPU utilizations from the average server of each primary tenant, runs the FFT algorithm on the series, groups the tenants into the three patterns described in Section 4.1 (periodic, constant, unpredictable) based on their frequency profiles, and then uses the K-Means algorithm to cluster the profiles in each pattern into classes. Clustering tags each class with the utilization pattern, its average utilization, and its peak utilization. It also maintains a mapping between the classes and their primary tenants.

Algorithm 1 Class selection algorithm.

```
 1: Given: Classes $C$, Headroom($type$,$c$), Ranking Weights $W$
 2: function SCHEDULE(Batch job $J$)
 3:     $J$.type = Length (short, medium, or long) from its last run
 4:     $J$.req = Max amount of concurrent resources from DAG
 5:     for each $c \in C$ do
 6:         $c$.weightedroom=Headroom($J$.type,$c$) $\times$ $W[J.\text{type},c.\text{class}]$
 7:     end for
 8:     $F = \{\forall c \in C |$ Headroom($J$.type,$c$) $\geq J$.req$\}$
 9:     if $F \neq \emptyset$ then
10:         Pick 1 class $c \in F$ probabilistically $\propto c$.weightedroom
11:         return $\{c\}$
12:     else if Job $J$ can fit in multiple classes combined then
13:         Pick $\{c_0, \ldots, c_k\} \subseteq C$ probabilistically $\propto c$.weightedroom
14:         return $\{c_0, \ldots, c_k\}$
15:     else
16:         Do not pick classes
17:         return $\{\emptyset\}$
18:     end if
19: end function
```

As we detail in Algorithm 1, our **class selection** algorithm relies on the classes defined by the clustering algorithm. When we need to allocate resources for a job's tasks, the algorithm selects a class (or classes) according to the expected job length (line 3) and a pre-determined ranking of classes for the length. We represent the desired ranking using weights (line 6); higher weight means higher ranking. For a long job, we give priority to constant classes first, then periodic classes, and finally unpredictable classes. We prioritize the constant classes in this case because constant-utilization primary tenants with enough available resources are unlikely to take resources away from the job during its execution. At the other extreme, a short job does not require an assurance of resource availability long into the future; knowing the current utilization is enough. Thus, for a short job, we rank the classes unpredictable first, then periodic, and finally constant. For a medium job, the ranking is periodic first, then constant, and finally unpredictable.

We categorize a job as short, medium, or long by comparing the duration of its last execution to two pre-defined thresholds (line 3). We set the thresholds based on the historical distribution of job lengths and the current computational capacity of

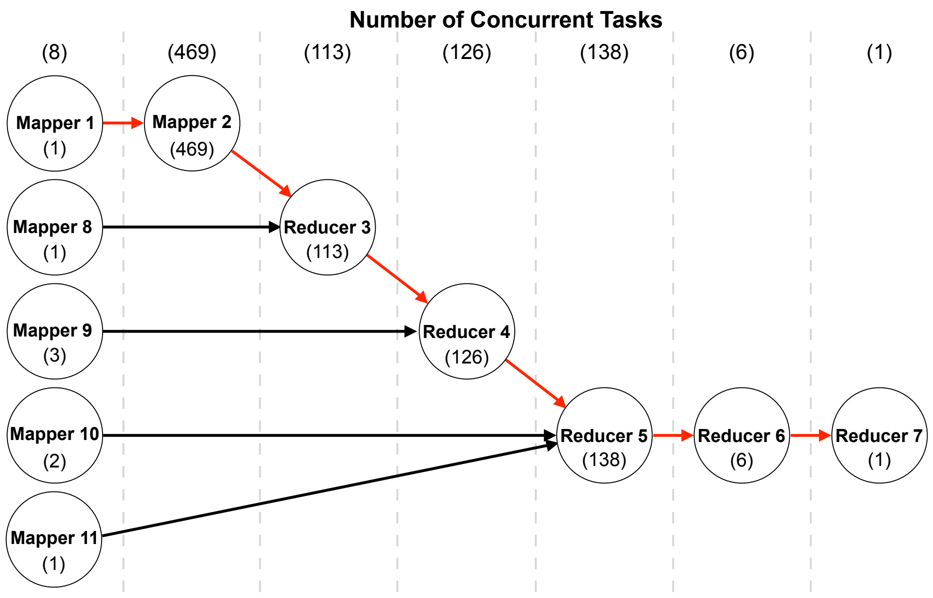


Figure 4.7: Example job execution DAG.

each preferred tenant class (*e.g.*, the total computation required by long jobs should be proportional to the computational capacity of constant primary tenants). Importantly, the last duration need *not* be an accurate execution time estimate. Our goal is much easier: to categorize jobs into three rough types. We assume that a job that has not executed before is a medium job. After a possible error in this first guess, we find that a job consistently falls into the same type.

We estimate the maximum amount of concurrent resources that the job will need (line 4) using a breadth-first traversal of the job's directed acyclic graph (DAG), which is a common representation of execution flows in many frameworks [22, 136, 175]. We find this estimate to be accurate for our workloads. Figure 4.7 shows an example job DAG (query 19 from TPC-DS [154]), for which we estimate a maximum of 469 concurrent containers.

Whether a job "fits" in a class (line 8) depends on the amount of available resources (or the amount of *headroom*) that the servers in the class currently exhibit, as we define below. When multiple classes could host the job, the algorithm selects one with probability proportional to its weighted headroom (lines 9 and 10). If multiple classes are necessary, it selects as many classes as needed, again probabilistically (lines 12 and 13). If there are not enough resources available in any combination of classes, it does not select any class (line 16).

The headroom depends on the job type. For a short job, we define it as 1 minus the current average CPU utilization of the servers in the class. For a medium job, we use 1 minus Max(average CPU utilization, current CPU utilization). For a long job, we use 1 minus Max(peak CPU utilization, current CPU utilization).

4.2.2 Smart data placement

Modern distributed file systems achieve high data access performance, availability, and durability, so there is a strong incentive for using them in our harvesting scenario. However, like cluster schedulers, they assume dedicated servers without primary tenants running and storing data on the same servers, and without primary tenant owners deliberately reimaging disks. Thus, we must (1) modify them to become co-location-aware; and (2) endow them with replica placement algorithms that improve data availability and durability in the face of primary tenants and how they are managed. Again, the first requirement is fairly easy to accomplish, so we discuss our implementation in Section 4.3. Here, we focus on the second requirement, *i.e.* smart replica placement.

The challenge is that the primary tenants and the management system may hurt data availability and durability for any block: (1) if the replicas of a block are stored in primary tenants that load-spike at the same time, the block may become unavailable; (2) if developers or the management system reimage the disks containing all the replicas of a block in a short time span, the block will be lost. A replica placement algorithm must then account for primary tenant and management system activity.

An intuitive best-first approach would be to try to find primary tenants that reimage their disks the least, and from these primary tenants select the ones that have

Algorithm 2 Replica placement algorithm.

```
1: Given: Storage space available in each server, Primary reimaging stats,
2:        Primary peak CPU util stats, Desired replication $R$
3: function PLACE REPLICAS(Block $B$)
4:     Cluster primary tenants wrt reimaging and peak CPU util
5:       into 9 classes, each with the same total space
6:     Select the class of the server creating the block
7:     Select the server creating the block for one replica
8:     for $r = 2$; $r \leq R$; $r = r + 1$ do
9:         Select the next class randomly under two constraints:
10:          No class in the same row has been picked
11:          No class in the same column has been picked
12:        Pick a random primary tenant of this class as long as
13:          its environment has not received a replica
14:        Pick a server in this primary tenant for the next replica
15:        if ($r$ mod 3) == 0 then
16:            Forget rows and columns that have been selected so far
17:        end if
18:    end for
19: end function
```

lowest CPU utilizations. However, this greedy approach has two serious flaws. First, it treats durability and availability independently, one after the other, ignoring their interactions. Second, after the space at all the "good" primary tenants is exhausted, new replicas would have to be created at locations that would likely lead to poor durability, poor availability, or both.

We prefer to make decisions that promote durability and availability at the same time, while consistently spreading the replicas around as evenly as possible across all types of primary tenants. Thus, our replica placement algorithm (Algorithm 2) creates a two-dimensional clustering scheme, where one dimension corresponds to durability (disk reimages) and the other to availability (peak CPU utilization). It splits the two-dimensional space into 3×3 classes (infrequent, intermediate, and frequent reimages versus low, medium, and high peak utilizations), each of which has the same amount of available storage for harvesting $S/9$, where S is the total amount of currently available storage (lines 4 and 5). This idea can be applied to splits other than 3×3, as long as they provide enough primary tenant diversity.

The above approach tries to balance the available space across classes. However, perfect balancing may be impossible when primary tenants have widely different amounts of available space, and the file system starts to become full. The reason is that balancing space perfectly could require splitting a large primary tenant across two or more classes. We prevent this situation by selecting a single class for each tenant, to avoid hurting placement diversity. The side effect is that small primary tenants get filled more quickly, causing larger primary tenants to eventually become the only possible targets for the replicas. This effect can be eliminated by not filling the file system to the point that less than three primary tenants remain as possible targets for replicas. In essence, there is a tradeoff between space utilization and diversity. We discuss this tradeoff further in Section 4.5.

When a client creates a new block, our algorithm selects one class for each replica. The first class is that of the server creating the block; the algorithm places a replica at this server to promote locality (lines 6 and 7). If the desired replication is greater than 1, it repeatedly selects classes randomly, in such a way that no row or column of the two-dimensional space has two selections (lines 9, 10, and 11). It places a replica in (a randomly selected server of) a randomly selected primary tenant in this class, while ensuring that no two primary tenants in the same environment receive a replica (lines 12, 13, and 14). Finally, for a desired replication level larger than 3, it does extra rounds of selections. At the beginning of each round, it forgets the history of row and column selections from the previous round (lines 15, 16, and 17).

The environment constraint is the only aspect of our techniques that is AutoPilot-specific. However, the constraints generalize to any management system: avoid placing multiple replicas in any logical (*e.g.*, environment) or physical (*e.g.*, rack) server grouping that induces correlations in resource usage, reimaging, or failures.

Figure 4.8 shows an example of our clustering scheme and primary tenant selection, assuming all primary tenants have the same amount of available storage. The rows defining the peak utilization classes do not align, as we ensure that the available

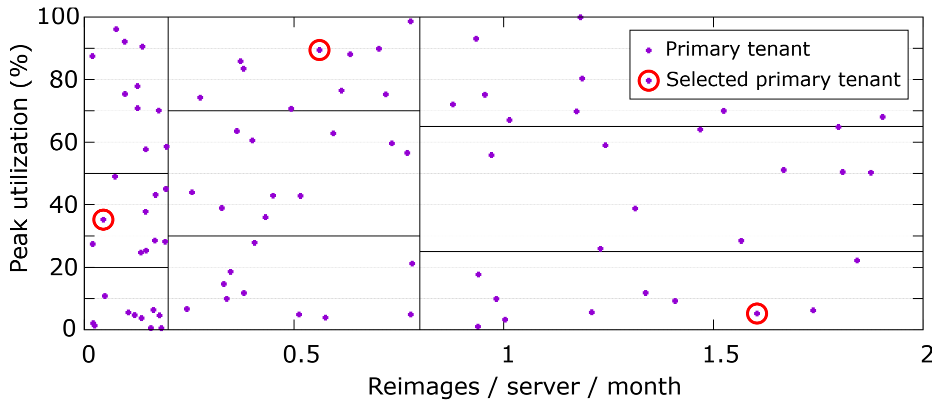


Figure 4.8: Two-dimensional clustering scheme.

storage is the same in all classes.

4.3 System Implementations

We implement our techniques into YARN, Tez, and HDFS. Next, we overview these systems. Then, we describe our implementation guidelines and systems, called YARN-H, Tez-H, and HDFS-H ("-H" refers to history).

4.3.1 Background

YARN [160] comprises a global Resource Manager (RM) running on a dedicated server, a Node Manager (NM) per server, and a per-job Application Master (AM) running on one of the servers. The RM arbitrates the use of resources (currently, cores and memory) across the cluster. The (primary) RM is often backed up by a secondary RM in case of failure. Each AM requests containers from the RM for running the tasks of its job. Each container request specifies the desired core and memory allocations for it, and optionally a "node label". The RM selects a destination server for each container that has the requested resources available and the same label. The AM decides which tasks it should execute in each container. The AM also tracks the tasks' execution, sequencing them appropriately, and re-starting any killed tasks. Each NM creates containers and reports the amount of locally available resources to

Table 4.1: Our main extensions to YARN, Tez, and HDFS.

System	Main extensions
YARN	Report primary tenant utilization to the RM Kill containers due to primary tenant needs Maintain resource reserve for primary tenant Probabilistically balance load
Tez	Leverage information on the observed job lengths Estimate max concurrent resource requirements Track primary tenant utilization patterns Schedule tasks on servers unlikely to kill them Schedule tasks on servers with similar primaries
HDFS	Track primary tenant utilization, deny accesses Report primary tenant status to the NN Exclude busy servers from info given to clients Track primary disk reimaging, peak utilizations Place replicas at servers with diverse patterns
General	Create dedicated environment for main components

the RM in periodic "heartbeats". The NM kills any container that tries to utilize more memory than its allocation.

Tez [136] is a popular framework upon which MapReduce, Hive, Pig, and other applications can be built. Tez provides an AM that executes complex jobs as DAGs.

HDFS [76] comprises a global Name Node (NN) running on a dedicated server, and a Data Node (DN) per server. The NN manages the namespace and the mapping of file blocks to DNs. The (primary) NN is typically backed up by a secondary NN. By default, the NN replicates each block (256 MBytes) three times: one replica in the server that created the block, one in another server of the same rack, and one in a remote rack. Upon a block access, the NN informs the client about the servers that store the block's replicas. The client then contacts the DN on any of these servers directly to complete the access. The DNs heartbeat to the NN; after a few missing heartbeats from a DN, the NN starts to re-create the corresponding replicas in other servers without overloading the network (30 blocks/hour/server).

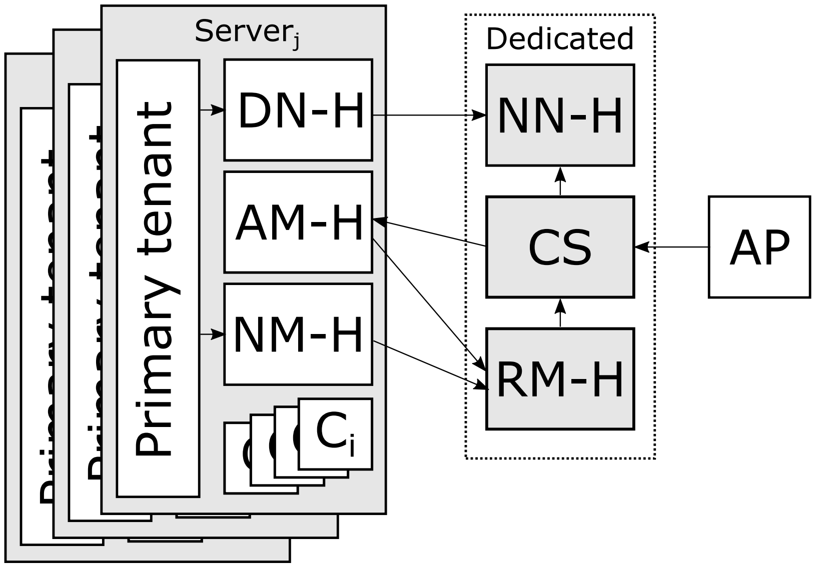


Figure 4.9: Overview of YARN-H (RM-H and NM-H), Tez-H (AM-H), and HDFS-H (NN-H and DN-H) in a co-location scenario. Our new clustering service (CS) interacts with all three systems. The arrows represent information flow. C_i = Container i; AP = AutoPilot.

4.3.2 Implementation guidelines

We first must modify the systems to become aware of the primary tenants and their priority over the servers' resources. Because of this priority, we must ensure that the key components of these systems (RMs and NNs) do not share their servers with any primary tenants. Second, we want to integrate our history-based task scheduling and data placement algorithms into these systems.

Figure 4.9 overviews our systems. The arrows in the figure represent information flow. Each shared server receives one instance of our systems; other workloads are considered primary tenants. Table 4.1 overviews our main extensions. The next sections describe our systems.

4.3.3 YARN-H and Tez-H

Design goals: (G1) ensure that the primary tenant always gets the cores and memory it desires; (G2) ensure that there is always a reserve of resources for the primary tenant to spike into; and (G3) schedule the tasks on servers where they are less likely to be killed due to the resource needs of the corresponding primary tenants.

Primary tenant awareness. We implement goals G1 and G2 in YARN-H by modifying the NM to (1) track the primary tenant's core and memory *utilizations*; (2) round them up to the next integer number of cores and the next integer MB of memory; and (3) report the sum of these rounded values and the secondary tenants' core and memory *allocations* in its heartbeat to RM-H. If NM-H detects that there is no longer enough reserved resources, it replenishes the reserve back to the pre-defined amount by killing enough containers from youngest to oldest.

Smart task scheduling. We implement goal G3 by implementing a service that performs our clustering algorithm, and integrating our class selection algorithm into Tez-H. We described both algorithms in Section 4.2.1.

Tez-H requests the estimated maximum number of concurrent containers from RM-H. When Tez-H selects one class, the request names the node label for the class. When Tez-H selects multiple classes, it uses a disjunction expression naming the labels. RM-H schedules a container to a heartbeating server of the correct class with a probability proportional to the server's available resources. If Tez-H does not name a label, RM-H selects destination servers using its default policy.

Overheads. Our modifications introduce negligible overheads. For primary tenant awareness, we add a few system calls to the NM to get the resource utilizations, perform a few arithmetic operations, and piggyback the results to RM-H using the existing heartbeat. The clustering service works off the critical path of job execution, computes headrooms using a few arithmetic operations, and imposes very little load on RM-H. In comparison to its querying of RM-H once per minute, *every* server heartbeats to RM-H *every 3 seconds*. Tez-H requires a single interaction with the clustering service per job.

4.3.4 HDFS-H

Design goals: (G1) ensure that we never use more space at a server than allowed by its primary tenant; (G2) ensure that HDFS-H data accesses do not interfere with the primary tenant when it needs the server resources; and (G3) place the replicas of each block so that it will be as durable and available as possible, given the resource usage of the primary tenants and how they are managed.

Note that full data durability cannot be *guaranteed* when using harvested storage. For example, service engineers or the management system may reimage a large number of disks at the same time, destroying multiple replicas of a block. Obviously, one can increase durability by using more replicas. We explore this in Section 5.4.

Primary tenant awareness. For goal G1, we use an existing mechanism in HDFS: the primary tenants declare how much storage HDFS-H can use in each server.

Implementing goal G2 is more difficult. To make our changes seamless to clients, we modify the DN to deny data accesses when its replica is unavailable (*i.e.*, when allowing the access would consume some of the resource reserve), causing the client to try another replica. (If all replicas of a desired block are busy, the block becomes unavailable and Tez will fail the corresponding task.) In addition, DN-H reports being "busy" or available to NN-H in its heartbeats. If DN-H says that it is busy, NN-H stops listing it as a potential source for replicas (and stops using it as a destination for new replicas as well). When the CPU utilization goes below the reserve threshold, NN-H will again list the server as a source for replicas (and use it as a destination for new ones).

Smart replica placement. For goal G3, we integrate our replica placement algorithm (Section 4.2.2) into NN-H.

Overheads. Our extensions to HDFS impose negligible overheads. For primary tenant awareness, we add a few system calls to the DN to get the primary tenant CPU utilization, and piggyback the results to NN-H in the heartbeat. Denying a request under heavy load adds two network transfers, but this overhead is minimal compared to that of disk accesses. For smart replica placement, our modifications add the clustering algorithm to the NN, and the extra communication needed for it to receive the algorithm inputs. The clustering and data structure updates happen in the background, off the critical path.

4.4 Evaluation

4.4.1 Methodology

Experimental testbed. Our testbed is a 102-server setup, where each server has 12 cores and 32GB of memory. We reserve 4 cores (33%) and 10GB (31%) of memory for primary tenants to burst into based on empirical measurements of interference. (Recall that performance isolation technology at each server would enable smaller resource reserves.) To mimic realistic primary tenants, each server runs a copy of the Apache Lucene search engine [123], and uses more threads (up to 12) with higher load. We direct traffic to the servers to reproduce the CPU utilization of 21 primary tenants (13 periodic, 3 constant, and 5 unpredictable) from data center DC-9. We also reproduce the disk reimaging statistics of these primary tenants. For the batch workloads, we run 52 different Hive [153] queries (which translate into DAGs of relational processing tasks) from the TPC-DS benchmark [154]. We assume Poisson inter-arrival times (mean 300 seconds) for the queries.

We use multiple baselines. When studying scheduling, the first baseline is stock YARN and Tez. We call it *"YARN-Stock"*. The second baseline combines primary-tenant-aware YARN with stock Tez, but does not implement smart task scheduling. We call it *"YARN-PT"*. We call our full system *"YARN-H/Tez-H"*. Given the workload above, we set the thresholds for distinguishing task length types to 173 and 433 seconds. Jobs shorter than 173 seconds are short, and longer than 433 seconds are long. These values produce resource requirements for the jobs of each type that roughly correspond to the amount of available capacity in the preferred primary tenant class for the type. We use HDFS-Stock with YARN-Stock, and HDFS-PT with the other YARN versions. The latter combination isolates the impact of primary tenant awareness in YARN from that in HDFS.

When studying data placement and access, the first baseline is *"HDFS-Stock"*, *i.e.* stock HDFS unaware of primary tenants. The second baseline is *"HDFS-PT"*, which brings primary tenant awareness to data accesses but does not implement smart data placement. We call our full system *"HDFS-H"*. We use YARN and Tez with HDFS-Stock, and YARN-PT and Tez with the other HDFS versions. Again, we seek to isolate the impact of primary tenant awareness in HDFS and YARN.

Simulator. Because we cannot experiment with entire data centers and need to capture long-term behaviors (*e.g.*, months to years), we also built a simulator that reproduces the CPU utilization and reimaging behavior of all the primary tenants (thousands of servers) in the data centers we study. We simulate servers of the same size and resource reserve as in our real experiments. To study a spectrum of utilizations, we also experiment with higher and lower traffic levels, each time multiplying the CPU utilization time series by a constant factor and saturating at 100%. Because of the inaccuracy introduced by saturation, we also study a method in which we scale the CPU utilizations using n^{th}-root functions (*e.g.*, square root, cube root). These functions make the higher utilizations change less than the lower ones when we scale them, reducing the chance of saturations.

When studying task scheduling and data availability, we simulate each data center for one month. When studying data durability, we simulate each data center for one year. We use the same set of Hive queries to drive our simulator, but multiply their lengths and container usage by a scaling factor to generate enough load for our large data centers (many thousands of servers) while limiting the simulation time.

In the simulator, we use the same code that implements clustering, task scheduling, and data placement in our real systems. The simulator also reproduces key behaviors from the real systems, *e.g.* it reconstructs lost replicas at the same rate as our real HDFS systems. However, it does not model the primary tenants' response times. We compare our systems to the second baseline (YARN-PT) in task scheduling, and the first baseline (HDFS-Stock) in data placement and access.

4.4.2 Performance microbenchmarks

The most expensive operations in our systems are the clustering and class selection in task scheduling and data placement. For task scheduling, clustering takes on average 2 minutes for the primary tenants of DC-9, when running single-threaded. (Recall that this clustering happens in the clustering service once per day, off the critical scheduling path.) The clustering produces 23 classes (13 periodic, 5 constant, and 5 unpredictable) for DC-9. For this data center, class selection takes less than 1 msec on average. For data placement, clustering and class selection take on average 2.55 msecs per new block (0.81 msecs in HDFS-Stock) for DC-9. (Clustering here can be done off the critical data placement path as well.)

4.4.3 Experimental results

Task scheduling comparisons. We start by investigating the impact of harvesting spare compute cycles on the performance of the primary tenant. Figure 4.10 shows the average of the servers' 99th-percentile response times (in ms) every minute during a five-hour experiment. The curve labeled "No Harvesting" depicts the tail latencies when we run Lucene in isolation. The other curves depict the Lucene tail latencies under different systems, when TPC-DS jobs harvest spare cycles across the cluster. The figure shows that YARN-Stock hurts tail latency significantly, as it disregards the primary tenant. In contrast, YARN-PT keeps tail latencies significantly

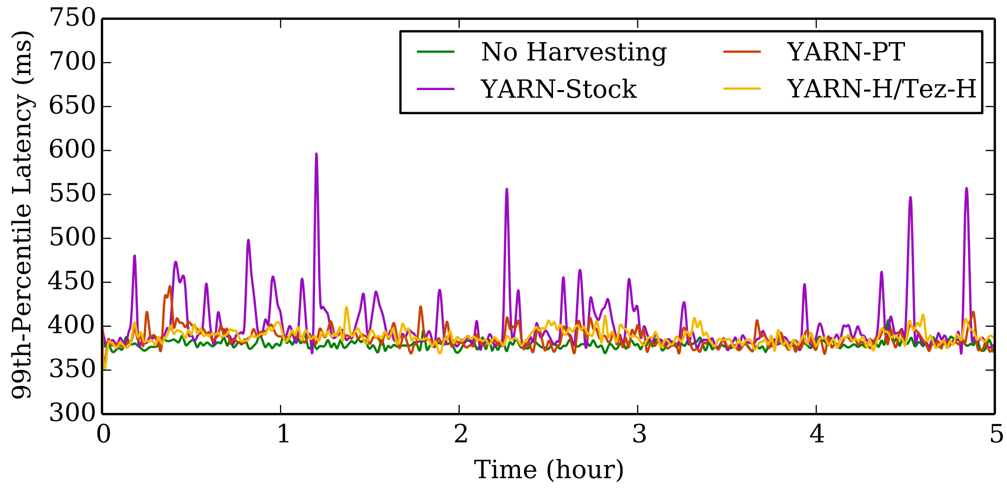


Figure 4.10: Primary tenant's tail latency in the real testbed for versions of YARN and Tez.

lower and more consistent. The main reason is that YARN-PT actually kills tasks to ensure that the primary tenant's load can burst up without a latency penalty. Finally, YARN-H/Tez-H exhibits tail latencies that nearly match those of the No-Harvesting execution. The maximum tail latency difference is only 44 ms, which is commensurate with the amount of variance in the No-Harvesting execution (average tail latencies ranging from 369 to 406 ms). The improved tail latencies come from the more balanced utilization of the cluster capacity in YARN-H.

Another key characteristic of YARN-H/Tez-H is its smart scheduling of tasks to servers where they are less likely to be killed. Figure 4.11 shows the execution times of all jobs in TPC-DS for YARN-Stock, YARN-PT, and YARN-H/Tez-H. As one would expect, YARN-Stock exhibits the lowest execution times. Unfortunately, this performance comes at the cost of ruining that of the primary tenant, which is unacceptable. Because YARN-PT must kill (and re-run) tasks when the primary tenant's load bursts, it exhibits substantially higher execution times, 1181 seconds on average. YARN-H/Tez-H lowers these times significantly to 938 seconds on average.

In these experiments, YARN-H/Tez-H improves the average CPU utilization from 33% to 54%, which is a significant improvement given that we reserve 33% of the CPU

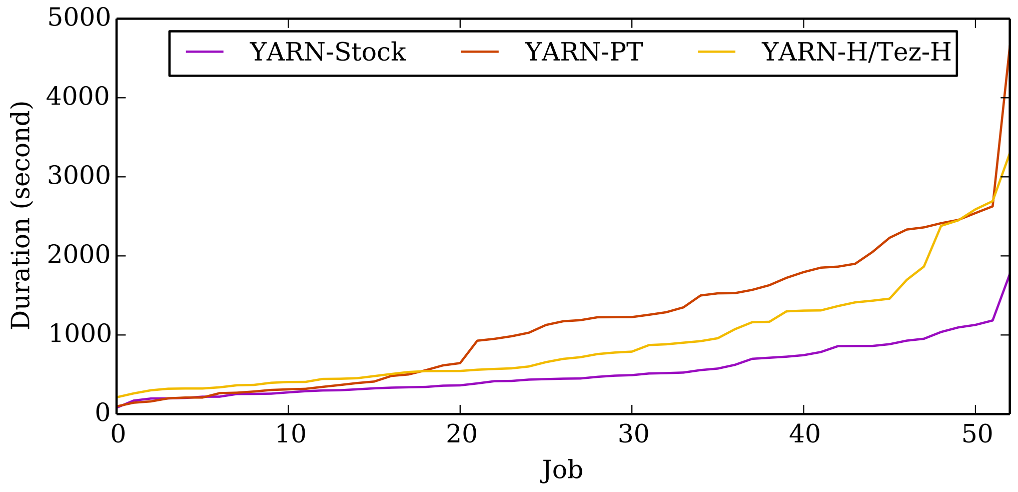


Figure 4.11: Secondary tenants' run times in the real testbed for versions of YARN and Tez.

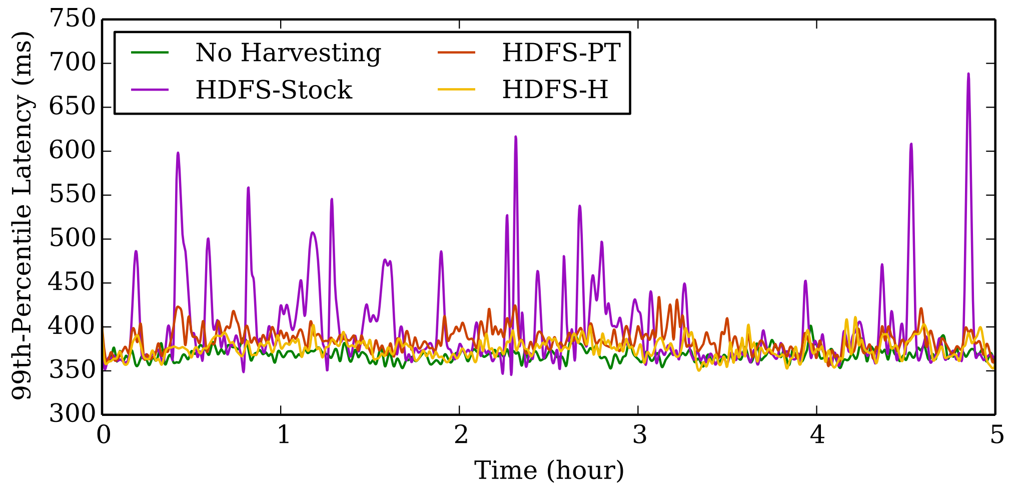


Figure 4.12: Primary tenant's tail latency in the real testbed for versions of HDFS.

for primary tenant bursts. The utilization improvement depends on the utilization of the primary tenants (the lower their utilization, the more resources we can harvest), the resource demand coming from secondary tenants (the higher the demand, the more tasks we can schedule), and the resource reserve (the smaller the reserve, the more resources we can harvest).

Overall, these results clearly show that YARN-H/Tez-H is capable of both protecting primary tenant performance and increasing the performance of batch jobs.

Data placement and access comparisons. We now investigate whether HDFS-

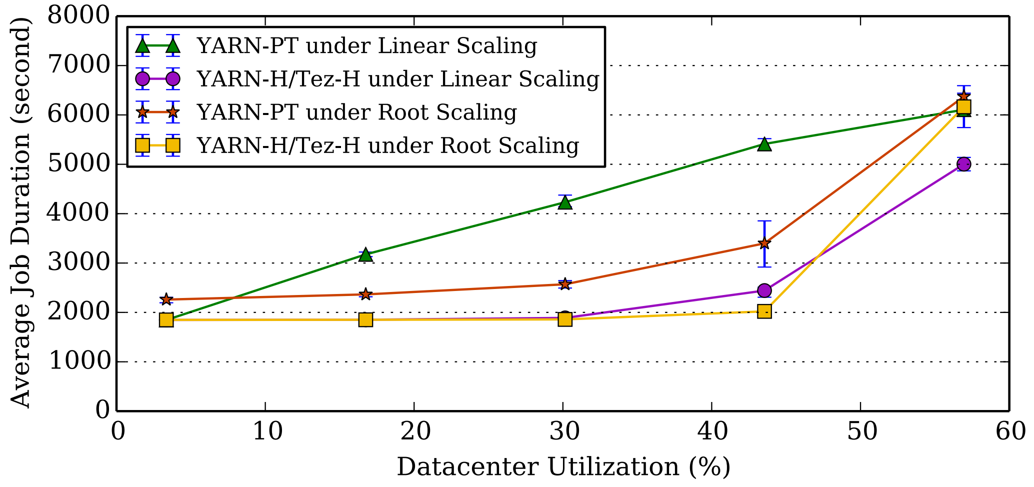


Figure 4.13: Secondary tenants' run time improvements in DC-9 under YARN-H/Tez-H for root and linear scalings.

H is able to protect the performance of the primary tenant and provide higher data availability than its counterparts. Figure 4.12 depicts the average of the servers' 99th-percentile response times (in ms) every minute during another five-hour experiment. As expected, the figure shows that HDFS-Stock degrades tail latency significantly. HDFS-PT and HDFS-H reduce the degradation to at most 47 ms. The reason is that these versions avoid accessing/creating data at busy servers. However, HDFS-PT actually led to 47 failed accesses, *i.e.* these blocks could not be accessed as all of their replicas were busy. By using our smart data placement algorithm, HDFS-H eliminated all failed accesses.

4.4.4 Simulation results

Task scheduling comparisons. We start our simulation study by considering the full spectrum of CPU utilizations, assuming the size and behavior of our real production data centers. Recall that we use two methods to scale utilizations (up and down) from the real utilizations: linear and root scalings. To isolate the benefit of our use of historical primary tenant utilizations, we compare YARN-H/Tez-H to YARN-PT. Figure 4.13 depicts the average batch job execution time in DC-9 under both systems and scalings, as a function of utilization. Each point along the curves shows the average of five runs, whereas the intervals range from the minimum average to the maximum average across the runs. As one would expect, high utilization causes higher queuing delays and longer execution times. (Recall that we reserve 33% of the resources for primary tenants to burst into, so queues are already long when we approach 60% utilization.) However, YARN-PT under linear scaling behaves differently; the average execution times start to increase significantly at lower utilizations. The reason is that linear scaling produces greater temporal variation in the CPU utilizations of each primary tenant than root scaling. Higher utilization variation means that YARN-PT is more likely to have to kill tasks, as it does not know the historical utilization patterns of the primary tenants. For example, at 45% utilization, YARN-PT under linear scaling kills 4$\times$ more tasks than the other system-scaling combinations.

Because YARN-H/Tez-H uses our clustering and smart task scheduling, it improves job performance significantly across most of the utilization spectrum. Under linear scaling, the average execution time reduction ranges from 0% to 55%, whereas under root scaling it ranges between 3% and 41%. The YARN-H/Tez-H advantage is larger under linear scaling, since the utilization pattern of each primary tenant varies more over time.

To see the impact of primary tenants with different characteristics than in DC-9, Figure 4.14 depicts the minimum, average, and maximum job execution time improvements from YARN-H/Tez-H across the utilization spectrum for each data center (five runs for each utilization level). The average improvements range from 12% to 56% under linear scaling, and 5% to 45% under root scaling. The lowest average improvements are for DC-0 and DC-2, which exhibit the least amount of primary tenant utilization variation over time. At the other extreme, the largest average improvements come for DC-1 and DC-4, as many of their primary tenants exhibit significant tem-

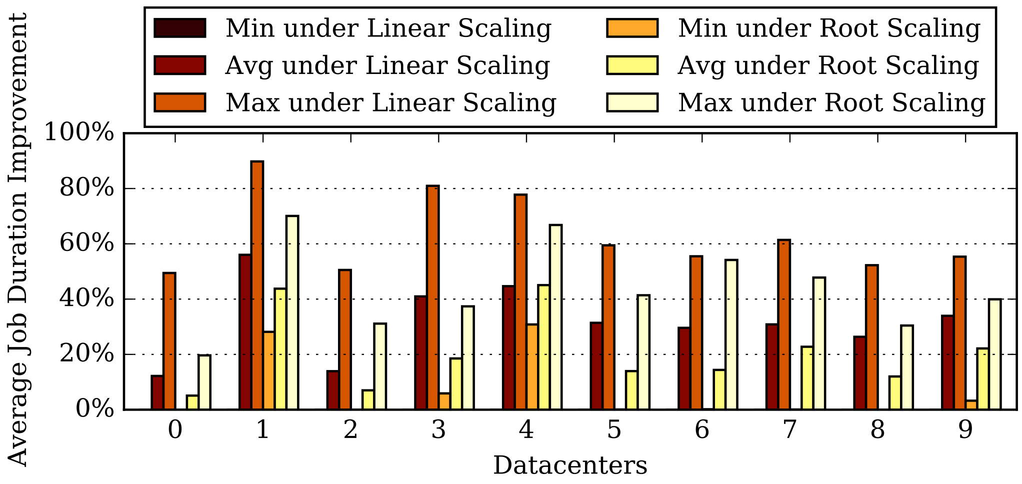


Figure 4.14: Secondary tenants' run time improvements from YARN-H/Tez-H for root and linear scalings.

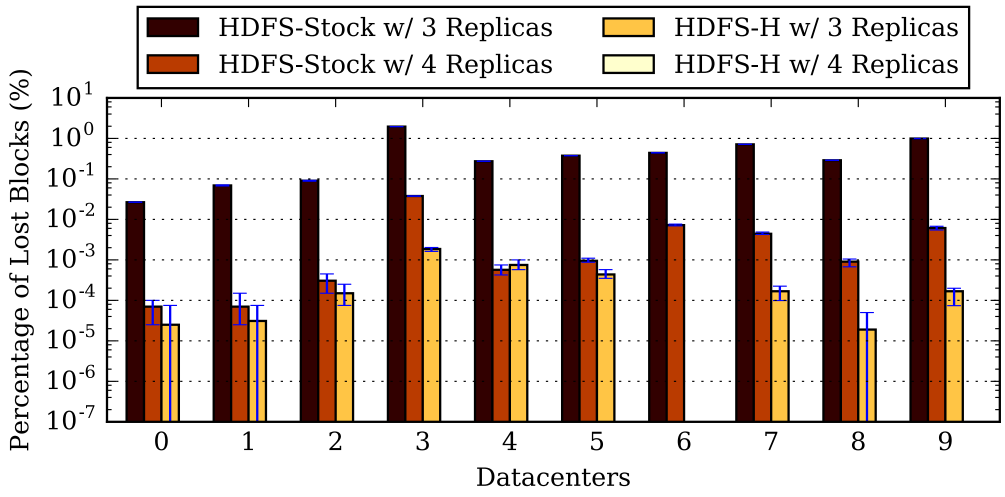


Figure 4.15: Lost blocks for two replication levels.

poral utilization variations. The largest maximum improvements (~90% and ~70% under linear and root scaling, respectively) also come from these two data centers, regardless of scaling type.

Data placement and access comparisons. We now consider the data durability in HDFS-H. Figure 4.15 shows the percentage of lost blocks under two replication levels (three and four replicas per block), as we simulate one year of reimages and 4M blocks. Each bar depicts the average of five runs, and the intervals range from the minimum to the maximum percent data loss in those simulations. The missing bars

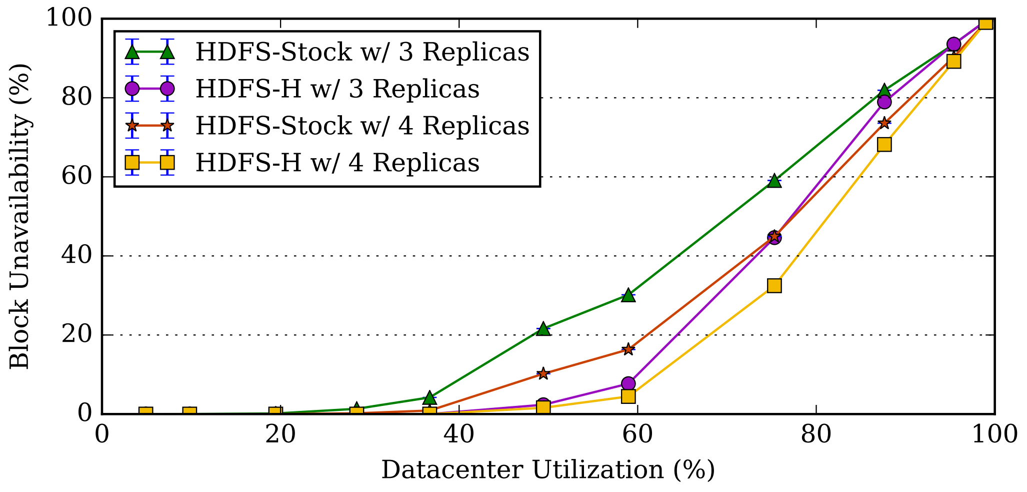


Figure 4.16: Failed accesses under linear scaling.

mean that there is no data loss in any of the corresponding five simulations. Note that a single lost block represents a 10^{-5} ($< 100 \times 1/4M$) percentage of lost blocks, *i.e.* 6 nines of durability.

The figure shows that HDFS-H reduces data loss more than two orders of magnitude under three-way replication, compared to HDFS-Stock. Moreover, for one of the data centers, HDFS-H eliminates all data loss under three-way replication. The maximum number of losses of HDFS-H in any data center was only 81 blocks (DC-3). Under four-way replication, HDFS-H completely eliminates data loss for all data centers, whereas HDFS-Stock still exhibits losses across the board. These results show that our data placement algorithm provides significant improvements in durability, despite the harvested nature of the disk space and the relatively high reimage rate for many primary tenants. In fact, the losses with HDFS-H and three-way replication are lower than those with HDFS-Stock and four-way replication for all but one data center; *i.e.* our algorithm almost always achieves higher durability at a lower space overhead than HDFS-Stock.

Our data availability results are also positive. Figure 4.16 depicts the percentage of failed accesses under the two replication levels and linear scaling, as a function of the average utilization. The figure includes range bars from five runs, but they are all too small to see. The figure shows that HDFS-H exhibits no data unavailability up to higher utilizations (~40%) than HDFS-Stock, and low unavailability for even higher utilization (50%), under both replication levels. At 50% utilization, HDFS-Stock already exhibits relatively high unavailability under both replication levels. Around 66% utilization, unavailability starts to increase faster (accesses cannot proceed if CPU utilization is higher than 66%). More interestingly, our smart data placement under three-way replication achieves lower unavailability than HDFS-Stock under four-way replication below 75% utilization. The trends are similar under root scaling, except that HDFS-H exhibits no unavailability up to a higher utilization (50%) than with linear scaling. Regardless of the scaling type, HDFS-H can achieve higher availability at a lower space overhead than HDFS-Stock for most utilizations.

4.5 Experiences in Production

As a first rollout stage, we deployed HDFS-H to a production cluster with thousands of servers eleven months ago. Since then, we have been enabling/adding features as our deployment grows. For example, we extended the set of placement constraints beyond environments to include machine functions and physical racks. In addition, we initially configured the system to treat the replica placement constraints as "soft", *e.g.* the placement algorithm would allow multiple replicas in the same environment, to prevent the block creation from failing when the available space was becoming scarce. This initial decision promoted space utilization over diversity. Section 4.2.2 discusses this tradeoff.

Since its production deployment, our system has eliminated all data losses, except for a small number of losses due to corner-case bugs or promoting space over diversity. Due to the latter losses, we started promoting diversity over space utilization more than nine months ago. Since then, we have not lost blocks. For comparison, when the stock HDFS policy was activated by mistake in this cluster for just three days during this period, dozens of blocks were lost.

We also deployed YARN-H's primary tenant awareness code to production fourteen months ago, and have not experienced any issues with it (other than needing to fix a few small bugs). We are now productizing our scheduling algorithm and will deploy it to production.

In the process of devising, productizing, deploying, and operating our systems, we learned many lessons.

1. Even well-tested open-source systems require additional hardening in production. We had to create watchdogs that monitor key components of our systems to detect unavailability and failures. Because of the non-trivial probability of concurrent failures, we increased the number of RMs and NNs to four instead of two. Finally, we introduced extensive telemetry to simplify debugging and operation. For example, we collect extensive information about HDFS-H blocks to estimate its placement quality.

2. Synchronous operations and unavailability. Synchronous operations are inadequate when resources or other systems become unavailable. For example, our production deployments interact with a performance isolation manager (similar to [119]). This interaction was unexpectedly harmful to HDFS-H. The reason is that the manager throttles the secondary tenants' disk activity when the primary tenant performs substantial disk I/O. This caused the DN heartbeats on these servers to stop flowing, as the heartbeat thread does synchronous I/O to get the status of modified blocks and free space. As a result, the NN started a replication storm for data that it thought was lost. We then changed the heartbeat thread to become asynchronous and report the status that it most recently found.

3. Data durability is king. As we mention above, our initial HDFS-H deployment favored space over diversity, which caused blocks to be lost and the affected users to become quite exercised. By default, we now monitor the quality of placements and stop consuming more space when diversity becomes low. To recover some space, we still favor space usage over diversity for those files that do not have strict durability requirements.

4. Complexity is your enemy. As others have suggested [48], simplicity, modularity, and maintainability are highly valued in large production systems, especially as engineering teams change and systems evolve. For example, our initial task scheduling technique was more complex than described in Section 4.2.1. We had to simplify it, while retaining most of the expected gains.

5. Scaling resource harvesting to massive data centers requires additional infrastructure. Stock YARN and HDFS are typically used in relatively small clusters (less than 4k servers), due to their centralized structure and the need to process heartbeats from all servers. Our goal is to deploy our systems to much larger installations, so we are now in the process of creating an implementation of HDFS-H that federates multiple smaller clusters and automatically moves files/folders across them based on primary tenant behaviors, and our algorithm's ability to provide high data availability and durability.

6. Contributing to the open-source community. Though our techniques are general, some of the code we introduced in our systems was tied to our deployments. This posed challenges when contributing changes to and staying in-sync with their open-source versions. For example, some of the YARN-H primary tenant awareness changes we made to Hadoop version 2.6 were difficult to port to version 2.7. Based on this experience, we refactored our code to isolate the most basic and general functionality, which we could then contribute back; some of these changes will appear in version 2.8.

4.6 Summary

In this Chapter, we first characterized all servers of ten large-scale data centers. Then, we introduced techniques and systems that effectively harvest spare compute cycles and storage space from data centers for batch workloads. Our systems embody knowledge of the existing primary workloads, and leverage historical utilization and management information about them. Our results from an experimental testbed and from simulations of the ten data centers showed that our systems eliminate data loss and unavailability in many scenarios, while protecting primary workloads and significantly improving batch job performance. Based on these results, we conclude that our systems in general, and our task scheduling and data placement policies in particular, should enable data center operators to increase utilization and reduce TCO.

CHAPTER V

Treadmill: Attributing the Source of Tail Latency

Managing tail latency of requests has become one of the primary challenges for large-scale Internet services. Data centers are quickly evolving and service operators frequently desire to make changes to the deployed software and production hardware configurations. Such changes demand a confident understanding of the impact on one's service, in particular its effect on tail latency (*e.g.*, 95th- or 99th-percentile response latency of the service). Evaluating the impact on the tail is challenging because of its inherent variability. Existing tools and methodologies for measuring these effects suffer from a number of deficiencies including poor load tester design, statistically inaccurate aggregation, and improper attribution of effects. As shown in this Chapter, these pitfalls can often result in misleading conclusions.

In this Chapter, we develop a methodology for statistically rigorous performance evaluation and performance factor attribution for server workloads. First, we find that careful design of the server load tester can ensure high quality performance evaluation, and empirically demonstrate the inaccuracy of load testers in previous work. Learning from the design flaws in prior work, we design and develop a modular load tester platform, `Treadmill`, that overcomes pitfalls of existing tools. Next, utilizing `Treadmill`, we construct measurement and analysis procedures that can properly attribute performance factors. We rely on statistically-sound performance evaluation and quantile regression, extending it to accommodate the idiosyncrasies of server systems. Finally, we use our augmented methodology to evaluate the impact of common server hardware features with Facebook production workloads on production hardware. We decompose the effects of these features on request tail latency and demonstrate that our evaluation methodology provides superior results, particularly in capturing complicated and counter-intuitive performance behaviors. By tuning the hardware features as suggested by the attribution, we reduce the 99th-percentile latency by 43% and its variance by 93%.

5.1 Pitfalls in the State-of-the-Art Methodologies

To understand the requirements of an evaluation test bed, we first survey existing methodologies and tools in prior work. Many tools are available to study the performance of server-side software, including YCSB [53], Faban [4], Mutilate [109] and CloudSuite [74]. These tools have been widely used in standard benchmark suites, including SPEC2010 jEnterprise [20], SPEC2011 SIP-Infrastructure [21], CloudSuite [74] and BigDataBench [166], thereby many recently published research projects.

Through studying these existing tools, we found several common pitfalls. We categorize them into four major themes provided below.

Table 5.1: Summary of load tester features.

	YCSB	Faban	CloudSuite	Mutilate	Treadmill
Query Interarrival Generation			✓		✓
Statistical Aggregation				✓	✓
Client-side Queueing Bias		✓		✓	✓
Performance Hysteresis					✓
Generality	✓	✓			✓

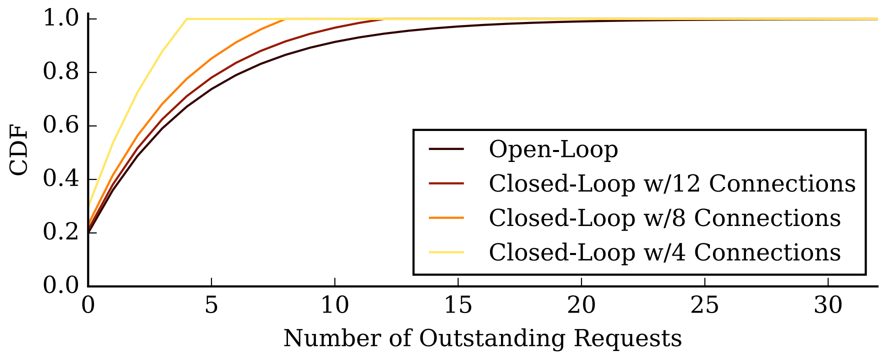


Figure 5.1: Comparison of the number of outstanding requests between closed-loop and open-loop controllers. The "Open-Loop" line shows the cumulative-distribution of the number of outstanding requests in an open-loop controlled system when running at 80% utilization. The "Closed-Loop" lines show the distribution of the number of outstanding requests in a closed-loop controlled system with 4, 8 and 12 concurrent connections respectively. The closed-loop controller significantly underestimates the number of outstanding requests in the system and therefore queueing latency, which creates bias in tail latency measurement.

5.1.1 Query Inter-arrival Generation

A performance evaluation test bed requires a *load tester*, a piece of software that issues requests to the server in a controlled manner. A client machine will run the load tester which periodically constructs, sends and receives requests to achieve a desired throughput. There are two types of control loops that are often employed to create these timings: open-loop control and closed-loop control [139]. The closed-loop controller has a feedback loop, where it only tries to send another request after the response to the previous request has already been received. In contrast, the open-loop controller sends requests at defined times regardless of the status of the responses. Almost all the modern data center server-side softwares are built to handle open-loop setup, so that each server thread does not reject requests from clients while busy processing previous ones.

However, many load testers are implemented as closed-loop controller because of software simplicity, including Faban, YCSB and Mutilate as we shown in Table 5.1. Often, load is generated by using worker threads that block when issuing network

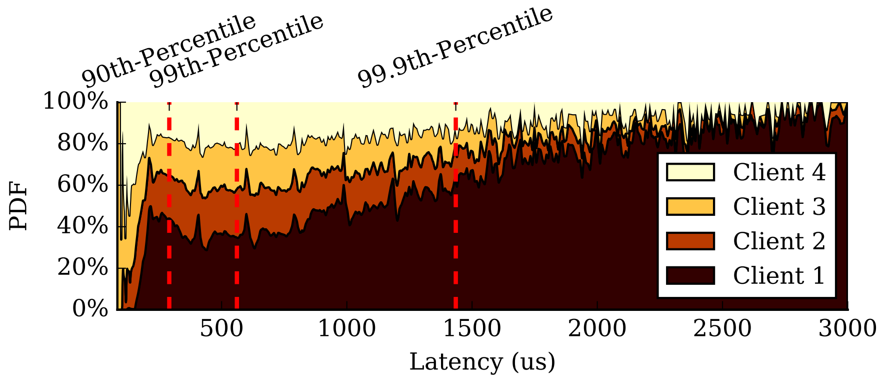


Figure 5.2: Different latency distributions measured from multiple clients in the a multi-client performance evaluation procedure, where the y-axis shows the decomposition among clients as what percent of samples is contributed by each client. We can see from the figure that Client 1 dominates the high quantiles of the combined distribution thus bias the measurement, because it resides on a different rack than the other clients and the server.

requests. The amount of load can then be controlled by increasing or decreasing the amount of threads in the load generator. Unfortunately, this pattern exactly resembles a closed-loop controller; each thread represents exactly one potentially outstanding request.

Figure 5.1 demonstrates the impact of closed-loop and open-loop design. For an open-loop design, the number of outstanding requests varies over time and follows the shown distribution. The high-quantiles of the distribution exhibit far more outstanding requests (and therefore queueing latency) than a closed-loop design. Using a closed-loop design can significantly underestimate the request latency especially at high quantiles. Therefore, we conclude that a open-loop design is required to properly exercise the queueing behavior of the system.

5.1.2 Statistical Aggregation

Due to high request rates, load tester software needs to perform at least some statistical aggregations of latency samples to avoid the overhead of keeping large number of samples. We find that care must be taken in this process and two types of errors can occur.

First, it is important for load testers to keep an internal histogram of latency that adapts over time. Those load testers that do maintain a histogram often make the mistake of statically setting the histogram buckets. Non-adaptive histogram binning will break when the server is highly utilized, because the latency will keep increasing before reaching the steady state thus exceeds the upper bound of the histogram.

Moreover, if the requests have distinct characteristics (*e.g.*, different request types, sent by different machines, etc.), we observe that bias can occur due to improper statistical aggregation. For example, in Figure 5.2 we demonstrate a scenario where four clients are used to send requests to the same Memcached server, and "Client 1" is on a different rack than the other clients and the server. At each latency point on the x-axis, each shaded region represents the proportion of samples that come from one of the four clients. As the quantile gets higher, one can clearly see that most of the samples are coming from "Client 1". This bias is problematic because the performance estimate of the system becomes a function of one single client. Instead, one should extract the interested metrics (*e.g.*, 99th-percentile latency) at each client individually, and aggregate them properly.

5.1.3 Client-side Queueing Bias

While operating a load tester, it is important to purely measure the effects of server-side latency. For workloads with long service time (*e.g.*, complex MySQL queries), clients do not have to issue many requests to saturate the server. However, for workloads like Memcached the request rates on the clients and network themselves are quite high. This can lead to queueing effects in the clients and network themselves, thereby bias the latency measurements. YCSB and CloudSuite suffer from such bias due to their single client configuration as shown in Table 5.1.

Figure 5.3 shows an example of how client and network utilizations can bias latency

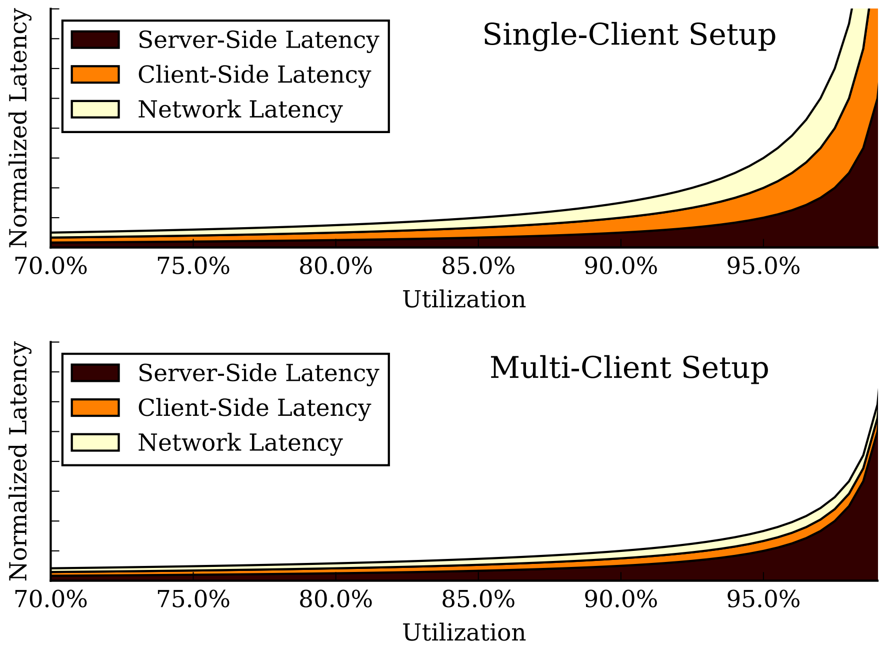


Figure 5.3: Comparison between single-client and multi-client setups for measuring request latency. In the single-client setup, the network and the client have the same utilization as the server, which results in increasing queueing delay when the server utilization increases. This will bias the latency measurement, whereas in the multi-client setup the utilizations of the network and the client are kept low enough that they only add an approximately constant latency.

measurements. In "Single-Client Setup", the client and the network have the same utilization as the server. As one can see, the client-side latency and the network latency grow as the server utilization increases, and represent a significant fraction of the end-to-end latency.

We find that it is extremely challenging, if not impossible, to design a single-client load tester that can fully saturate the server without incurring significant client-side queueing for modern data center workloads that operate at microsecond-level latency. Instead, it is necessary to have a multi-client setup and have a sufficient number of machines such that the client-side and network latency is kept low. In "Multi-Client Setup", we increase the number of client machines to minimize these biases. After this adjustment the majority of measured latency comes from the server.

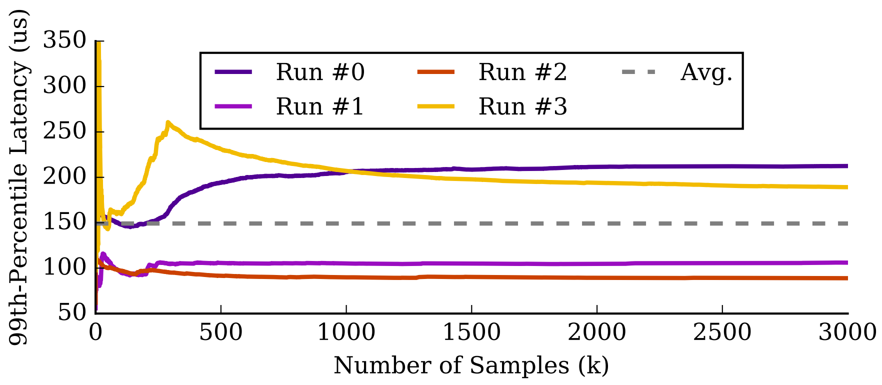


Figure 5.4: Variance exists regardless of a single run's sample size. A single run exhibits a large variance for a small sample size (*i.e.*, early in the run). With a sufficiently large sample size the estimate of 99th-percentile latency converges. However, empirically we find that each run can converge to a *different* value. Although the testing procedure of each run would yield a tight confidence interval, the result of each run clearly varies significantly (15-67% variation from the average).

5.1.4 Performance "Hysteresis"

Through experimentation we find an usual behavior in how estimates converge that we refer to as *performance "hysteresis"*. Figure 5.4 demonstrates that as more samples are collected, the estimate of 99th-percentile latency begins to converge to a singular value. However, if the server is restarted and another run is performed, the estimate can converge to a different value.

In this case, the estimates have a large sample size and we would have expected the "confidence" in each estimate to be high, but clearly there still exists variance *across* runs. In fact, the difference between these estimates and the average is as high as 67%. This phenomenon means that one cannot achieve higher statistical accuracy simply by "running for longer" and is similar to effects observed in STABILIZER [55]. Instead, it is necessary to restart the entire procedure many times and aggregate the results. However, none of the existing load testers is robust enough to handle this scenario as shown in Table 5.1.

5.2 Methodology

To overcome the four common pitfalls we find in existing methodologies, we design and develop `Treadmill`, a modular software load tester (Section 5.2.1), and a robust procedure for tail latency measurement (Section 5.2.2). To demonstrate the effectiveness of our methodology, we then evaluate it on real hardware (Section 5.2.3).

5.2.1 Treadmill

Given the existing pitfalls in state-of-the-art load tester tools, we decide to build our own load tester `Treadmill`. Specifically, the problems in existing tools are addressed by the following design decisions.

- **Query inter-arrival generation:** To guarantee the query inter-arrival generation can properly exercise the queueing behavior of the system, we implement an open-loop controller. The control loop is precisely timed to generate requests at an exponentially distributed inter-arrival rate, which is consistent with the measurements obtained from Google production clusters [124].

- **Statistical aggregation:** To provide high precision statistical aggregation, `Treadmill` goes through three phases during one execution: warm-up, calibration and measurement. During the warm-up phase, all measured samples are discarded. Next, we determine the lower and upper bounds of the sample histogram bins in the calibration phase. The calibration phase is useful to reduce the amount of information lost from transforming detailed latency samples into a histogram. Finally, `Treadmill` begins to collect samples until the end of execution. Histograms are used to reduce the storage and performance overhead, and are re-binned when sufficient amount of values exceed the histogram limits.

- **Client-side queueing bias:** To avoid client-side queueing bias, we use wangle [77], which provides inline execution of the callback function, to ensure that the response callback logic is executed immediately when the response is available. In addition, we highly optimize for performance (*e.g.*, lock-free implementation), which indirectly reduces the client-side queueing bias by keeping the clients at low utilization.

Furthermore, we also optimize for generality, making it easy to extend `Treadmill` to new workloads. Moreover, `Treadmill` is also able to reproduce configurable workload characteristics.

- **Generality:** We also optimize for generality, to minimizes the amount of effort required to extend `Treadmill` to new workloads. So far, we have successfully integrated `Treadmill` with several services including Memcached [129] and mcrouter [113]. Each integration takes less than 200 lines of code.
- **Configurable workload:** It has been demonstrated in prior work [31] that workload characteristics (*e.g.*, the ratio between GET and SET requests in Memcached) can have a big impact on the system performance. Therefore, being able to evaluate the system against various workload characteristics can improve the accuracy of measurement. To do so, a JSON formatted configuration file can be used to describe the workload characteristics (*e.g.*, request size distribution) and fed into `Treadmill`.

5.2.2 Tail Latency Measurement Procedure

`Treadmill` provides highly accurate measurement even at high quantiles. However, as we illustrated in Figure 5.3, multiple clients are needed to avoid client-side queueing bias for testing high throughput workloads like Memcached. Therefore, we develop a methodology leverages multiple `Treadmill` instances to perform load testing for such workloads.

To measure the tail latency, multiple instances of `Treadmill` are used to send requests to the same server, where each instance sends a fraction of the desired throughput. Then the same experiment is repeated multiple times, and the measurements from each experiment is aggregated together to get a converged estimate. Particularly, We make the following design decisions when designing this procedure.

- **Statistical aggregation:** First, we need to aggregate the statistics reported by multiple instances of `Treadmill` in each experiment. In this process, the common practice that combines distributions obtained from all `Treadmill` instances to a holistic distribution and then extracts interested metrics (*e.g.*, 99th-percentile latency) could be heavily biased by outliers as we illustrated in Figure 5.2. Instead, we first compute the interested metrics from each individual `Treadmill` instance, and then combine them by applying aggregation functions (*e.g.*, mean, median) on these metrics.

- **Client-side queueing bias:** By leveraging multiple instances of `Treadmill`, that each of them is highly performing, we can keep all the clients under low utilization thus prevent the measurement from client-side queueing bias.

- **Performance hysteresis:** To avoid performance hysteresis, multiple measurements are taken by repeating the same experiment multiple times. After each experiment, we record the collected measurements and repeat this procedure until the mean of the collected the measurements has already converged.

5.2.3 Evaluation

In this section, we evaluate the accuracy of the tail latency measurement obtained from `Treadmill`. We focus on Memcached [75] due to its popularity in both industry [28, 129] and academia [69, 114, 93, 92, 112, 87], as well as its stringent performance needs. Besides `Treadmill`, we also deploy two other recently published load testers CloudSuite [74] and Mutilate [109] for comparison.

To set them up, we explicitly follow the instructions they publish online. Specifically, 1 machine is used for running Memcached server, and 1 machine is used for the load tester from CloudSuite. Mutilate runs on 8 “agent” clients and 1 “master” client as suggested in the instructions, and also sends requests to 1 Memcached server. For `Treadmill`, we also use 8 clients in order to compare against Mutilate with the same amount of resource usage. Table 5.2 shows the hardware specifications of the system under test, which is used for all the experiments in this paper.

Table 5.2: Hardware specification of the system under test.

	Specification
Processor	Intel Xeon E5-2660 v2
DRAM	144GB @ 1333MHz
Ethernet	10GbE Mellanox MT27500 ConnectX-3
Kernel Version	3.10

When setting up these load testers, we also start a tcpdump process on the load test machines to measure the ground truth latency distribution. Tcpdump provides a good ground truth measurement because it measures the latency at network-level, thus eliminates potential client-side queueing delay. The tcpdump process is pinned on an idle physical core to avoid possible probe effect.

Tcpdump records the timestamps that request and response packets flow through the network interface card (NIC). By matching the sequence IDs of the packets, we can map each request to its corresponding response and calculate the time difference between the two timestamps as the ground truth latency in our evaluation. However, this ground truth latency is *expected to be lower* than the one that the load testers measure, because the timestamps tcpdump reports are taken when the network packets arrive the NIC. Certain amount of time is spent in kernel space to handle the network interrupts before the packets reach the user code, where these load testers reside.

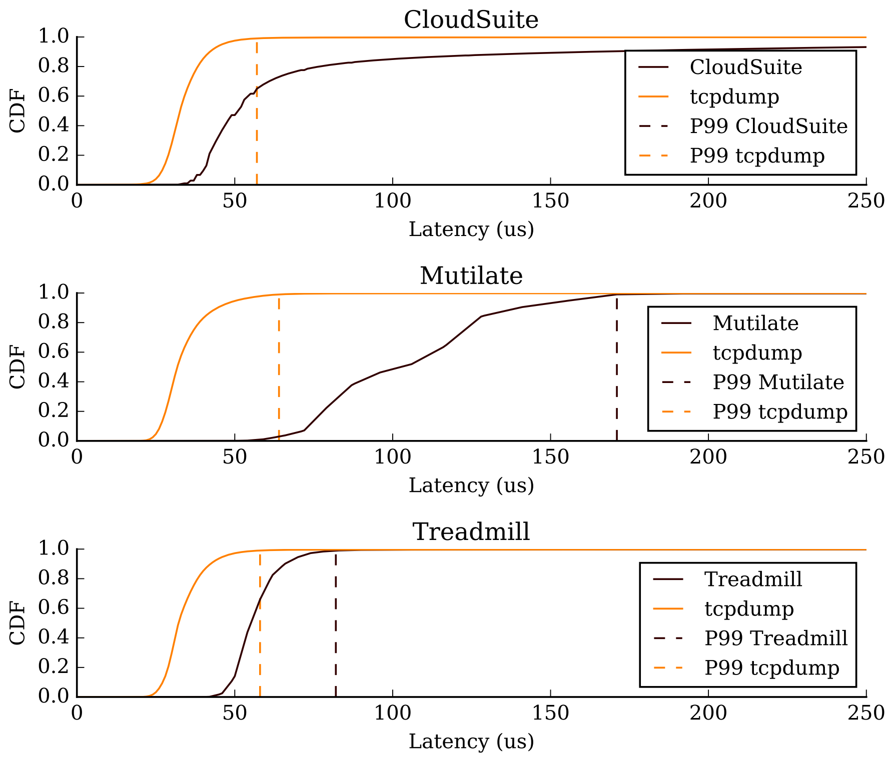


Figure 5.5: Latency distributions measured by CloudSuite, Mutilate and `Treadmill` at 10% server utilization, in which only `Treadmill` accurately captures the ground truth distribution measured by tcpdump. CloudSuite heavily overestimates the tail latency due to client-side queueing delay, and Mutilate also overestimates the tail latency and fail to capture the shape of the ground truth distribution. In contrast, `Treadmill` precisely captures the shape of ground truth distribution and maintains a constant gap to the tcpdump curve even at high quantiles. Note the gap between tcpdump measurement and load tester measurement is *expected* due to kernel space interrupt handling as we explain in the experimental setup.

5.2.3.1 Measurement under 10% Utilization

In the first experiment, we use the three load testers to send 100k requests per second (RPS) to the Memcached server. This translates to 10% CPU utilization on the server. We modified all three load testers to report the entire latency distribution at the end of execution as shown in Figure 5.5, in which the tcpdump curve shows the latency distribution measured by tcpdump as ground truth. The 99th-percentile latency measured by each tool is plotted in the figure with the dashed line.

As shown in Figure 5.5, CloudSuite measures a drastically higher tail latency

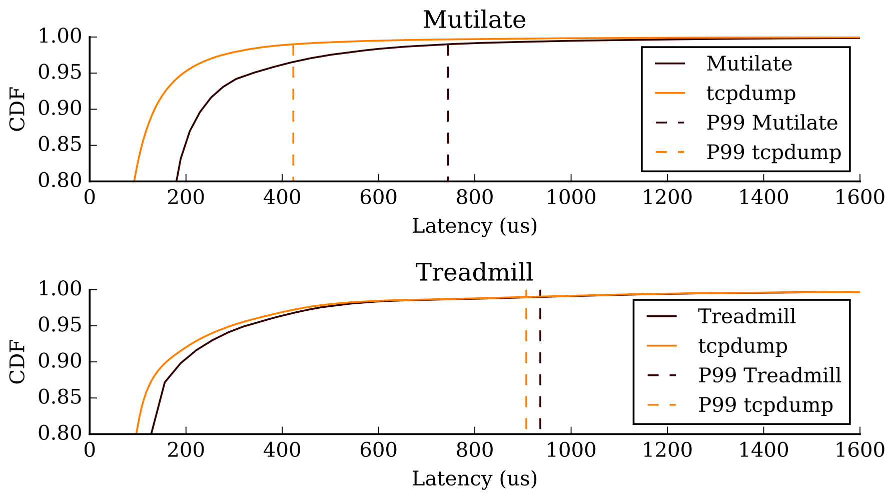


Figure 5.6: Latency distributions measured by Mutilate and `Treadmill` at 80% CPU utilization, where the ground truth tail latency measured by tcpdump is underestimated in the Mutilate experiment due to closed-loop controller. CloudSuite is not efficient enough to saturate the server at such high utilization because it runs a single client, thus not shown in the figure. As we illustrated in Figure 5.1, Mutilate, which runs a closed-loop controller, limits the maximum number of outstanding requests thus underestimates the ground truth tail latency. However, `Treadmill` is still able to precisely measure even the high percentile latency, and the expected gap between `Treadmill` and tcpdump remains the same (30μs) as during low utilization shown in Figure 5.5.

(99th-percentile latency is higher than 250μs thus not shown in the figure), because of heavy client-side queueing bias. This is due to the fact that it runs a single client to generate the load. Although Mutilate leverages 8 clients, it still fails to capture the shape of the ground truth latency distribution, and overestimates the tail latency. Due to the improper query inter-arrival generation, we notice that the ground truth latency distribution measured in Mutilate experiment is different from the ones measured in CloudSuite and `Treadmill` experiments. In contrast, `Treadmill` precisely captures the shape of the ground truth latency distribution, thus achieves highly accurate measurements. There is a fixed offset between the tcpdump and `Treadmill` curves, due to the expected computation spent in kernel space for interrupt handling.

5.2.3.2 Measurement under 80% Utilization

Similarly, we construct another experiment with these three load testers to send 800k requests per second (RPS) to one Memcached server, which runs at 80% CPU utilization. In this experiment, we find that CloudSuite is not efficient enough to send this many requests because of the performance limitation of a single client; we only report the measurements from Mutilate and `Treadmill` in Figure 5.6.

Similar to the previous experiment, the measured ground truth latency distributions from Mutilate experiment and `Treadmill` experiment are drastically different, especially at high quantile. This is due to the fact that tcpdump measures the ground truth driven by the control loop of the load tester. Mutilate runs a closed-loop controller, which artificially limits the maximum number of outstanding requests as we illustrated in Figure 5.1, therefore heavily underestimates the 99th-percentile latency by more than $2\times$. Open-loop controller does a much better job properly exercising the queueing behavior of the system, because the number of outstanding requests is not limited, which reassembles a realistic setting in production environment. Although the implementation of Mutilate overestimates the tail latency from the "ground truth", the 99th-percentile latency measured by Mutilate is still underestimated. Note that `Treadmill` still maintains a fixed offset to the ground truth latency distribution measured by tcpdump, and the offset is exactly the same (30μs) as during low utilization shown in Figure 5.5.

In conclusion, CloudSuite suffers from heavy client-side queueing bias, and Mutilate cannot properly exercise the queueing behavior of the system due to the closed-loop controller, whereas `Treadmill` precisely measures the tail latency even at high utilization.

5.3 Tail Latency Attribution

With sufficient amount of samples, `Treadmill` is able to obtain accurate latency measurements even at high quantiles. However, we sometimes find the measured latency from each run converges to a different value as shown in Figure 5.4. This suggests that the systems or the states of the system we measure are changing across runs, which may also happen in production environment if we do not have a technique to carefully control it. In this section, we analyze this variance of the tail latency using a recently developed statistical inference technique, quantile regression [105], and attribute the source of variance to various factors.

5.3.1 Quantile Regression

To analyze the observed variance in a response variable, analysis of variance (ANOVA) is often used to partition the variance and attribute it to different explanatory variables. However, the classic ANOVA technique assumes normally distributed residuals and equality of variances, which have been demonstrated unsuitable by prior work [57] for many computer system problems due to the common presence of non-normally distributed data. In addition, ANOVA can only attribute the variance of the sample means. In contrast, quantile regression [105] is a technique proposed to attribute the impact of various factors on any given quantiles, which does not make any assumption on the distribution of the underlying data. Therefore, quantile regression is particularly suitable for our purpose of analyzing the sources that contribute to the tail latency.

Similarly to ANOVA, quantile regression takes a number of samples as its input, where each sample includes a set of explanatory variables x_i and a response variable y. The response variable is expected to vary depending on the explanatory variables. Quantile regression produces estimates of coefficients c_i that minimizes the prediction error on a particular quantile τ for given X as shown in Equation 5.1. In addition to individual explanatory variables, it can also model the interactions among them by including their products (*e.g.*, $c_{12}(\tau)x_1x_2$ in Equation 5.1). It uses numerical optimization algorithm to minimize a loss function, which assigns a weight τ to underestimated errors and $(1-\tau)$ to overestimated ones for τ-th quantile, instead of minimizing the squared error in ANOVA.

$$\begin{aligned} Q_y(\tau|X) = & c_0(\tau) + c_1(\tau)x_1 + c_2(\tau)x_2 + \cdots + \\ & c_{12}(\tau)x_1x_2 + c_{13}(\tau)x_1x_3 + \cdots + \\ & \ldots \end{aligned} \tag{5.1}$$

In this case, we design the response variable to be a particular quantile (*e.g.*, 99th-percentile) of the latency distribution and the explanatory variables to be a set of factors that we suspect to have an impact on the latency distribution.

5.3.2 Factor Selection

First of all, we need to identify the potential factors that may affect the tail latency. We list all the factors we suspect to have an impact on the tail latency. Then we use null hypothesis testing on a large number of samples collected from repeated experiments under random permutations of all the factors, to identify the factors that actually have an impact on the tail latency. Although the factors may vary depending on the workload and the experimental environment, we find a list of factors consistently affecting the tail latency across various workloads we have experimented with.

- *NUMA Control:* The control policy for non-uniform memory access (NUMA) determines the memory node(s) to allocate for data. The same-node policy prefers to allocate memory on the same node until it cannot be allocated anymore, whereas the interleave policy uses round robin among all nodes.

- *Turbo Boost:* Frequency up-scaling feature is implemented on many modern processors, where the frequency headroom heavily depends on the dynamic power and thermal status. The scaling management algorithm is implemented in processor's hardware power control unit, and it is not clear how it will impact the tail latency quantitatively.

- *DVFS Governor:* Dynamic voltage frequency scaling allows the operating system to up-scale the CPU frequency to boost performance and down-scale to save power dynamically. In this section, we study two commonly used governors including performance (always operating at the highest frequency) and ondemand (scaling up the frequency only when utilization is high).

- *NIC Affinity:* The hardware network interface card (NIC) uses receive side scaling (RSS) to route the network packets to different cores for interrupt handling. The routing algorithm is usually implemented through a hashing function computed from the packet header. For example, the NIC on our machine (shown in Table 5.2) provides a 4-bit hashing value, which limits the number of interrupt queues to $2^4 = 16$. We study the impact of mapping all the interrupt queues to cores on the same CPU socket, and evenly spread across the two sockets.

Therefore, we use a 2-level full factorial experiment design with the 4 factors listed above as shown in Table 5.3.

Table 5.3: Quantile regression factors.

Factor	Low-Level	High-Level
NUMA Control (numa)	same-node	interleave
Turbo Boost (turbo)	off	on
DVFS Governor (dvfs)	ondemand	performance
NIC Affinity (nic)	same-node	all-nodes

In addition to the 4 factors listed above in isolation, we also model the interactions among combinations of them, because they might not necessarily be independent from each other. For example, the impact of DVFS governor may depend on Turbo Boost because they can indirectly interact with each other due to the contention in thermal headroom.

5.3.3 Quantifying Goodness-of-fit

In ANOVA, coefficient of determination R^2 is often used to quantify the fraction of variance that the model is able to explain. However, an equivalent of R^2 does not exist for quantile regression. Therefore, we define a pseudo-R^2 metric in Equation 5.2 using the same idea. The metric falls in the range of $[0, 1]$, where 1 means the model perfectly predicts the given quantile and 0 means its accuracy is the same as the best constant model that always predicts the same value regardless of the explanatory variables. In the equation, the numerator represents the sum of the prediction errors of the quantile regression model, and the denominator is the error of the best constant model.

$$pseudo\text{–}R^2_\tau = 1 - \frac{\sum_{i=0}^{N} w(\tau, err^\tau_{qr}(i))|err^\tau_{qr}(i)|}{\sum_{i=0}^{N} w(\tau, err^\tau_{const}(i))|err^\tau_{const}(i)|} \tag{5.2}$$

For each sample, the prediction error is computed as the product of the absolute prediction error and a weight. The prediction error for sample i at τ-th quantile is defined in Equation 5.3 as the difference between empirically measured quantile y_i^τ and the predicted quantile $model^\tau(X_i)$ conditional on explanatory variables X_i.

$$err^\tau_{model}(i) = y_i^\tau - model^\tau(X_i) \tag{5.3}$$

The weight assigned to each error is defined in Equation 5.4 as $(1 - \tau)$ for overestimation and τ for underestimation, which is the same as the loss function in quantile regression.

$$w(\tau, err) = \begin{cases} (1-\tau) & : err < 0 \\ \tau & : err \geq 0 \end{cases} \tag{5.4}$$

5.4 Evaluation of Hardware Features

The precision of tail latency measurements achieved by `Treadmill` enables the possibility of understanding and attributing the sources of tail latency variance. In this section, we present the complex and counter-intuitive performance behaviors identified through attributing the source of tail latency, and demonstrate the effectiveness of our methodology in improving tail latency.

5.4.1 Experimental Setup

We leverage quantile regression to analyze the measurements obtained under various configurations presented in Table 5.3 to understand the sources of tail latency variance.

To perform quantile regression, we first obtain latency samples under various configurations. We randomly choose one permutation of the configurations for each experiment to preserve independence among experiments, until we have at least 30 experiments for each permutation of the configurations. Given we are studying 4 factors, we will need at least $2^4 \times 30 = 480$ experiments. For each experiment, we randomly sub-sample 20k latency samples during the time when the latency distribution has already converged. We make sure this sub-sampling does not hurt the precision of the analysis by comparing against a model obtained using more samples, and we observe no significant difference.

Each factor is coded as 0 at low-level, and 1 at high-level in the samples. Before feeding the data into the quantile regression model, we perturb the data using a symmetric variance at 0.01 standard deviation. This is useful to prevent the numerical

Table 5.4: Results of quantile regression for Memcached at high utilization, which detail the contribution of each feature on the latency. The first several rows show the base latency (Intercept) and the latency of each feature enabled in isolation. The other rows provide the interaction latency effect of multiple features. For example, "turbo" is the best single feature (-29μs) in isolation to turn on to reduce 99th-percentile latency. Turning "nic" to high-level is only beneficial if "dvfs" is set to high-level (29 + -8 + -58 = -37μs), otherwise the net effect would be an latency increase (29μs). Surprisingly, setting "turbo" on, which is beneficial in isolation, in addition to "nic" and "dvfs" would actually increase the 99th-percentile latency (-29 + -8 + 29 + 40 + 23 + -58 + 4 = 1μs) due to the negative interaction among them. Note that for some rows, the uncertainty in the data is significant, and we choose a p-value of 0.05 to highlight these values in bold.

	50th-Percentile			95th-Percentile			99th-Percentile		
Factor	Est.	Std. Err	p-value	Est.	Std. Err.	p-value	Est.	Std. Err	p-value
(Intercept)	65 μs	<1 μs	<1e-06	155 μs	<1 μs	<1e-06	355 μs	5 μs	<1e-06
numa	2 μs	<1 μs	<1e-06	24 μs	<1 μs	<1e-06	56 μs	8 μs	<1e-06
turbo	-2 μs	<1 μs	<1e-06	-12 μs	<1 μs	<1e-06	-29 μs	7 μs	1.00e-04
dvfs	1 μs	<1 μs	<1e-06	<1 μs	<1 μs	**2.60e-01**	-8 μs	8 μs	**3.54e-01**
nic	<1 μs	<1 μs	<1e-06	2 μs	<1 μs	2.07e-03	29 μs	8 μs	1.10e-04
numa:turbo	3 μs	<1 μs	<1e-06	5 μs	1 μs	2.40e-04	21 μs	11 μs	**6.37e-02**
numa:dvfs	-3 μs	<1 μs	<1e-06	-29 μs	1 μs	<1e-06	-57 μs	11 μs	<1e-06
numa:nic	<1 μs	<1 μs	<1e-06	-6 μs	1 μs	4.00e-05	-20 μs	11 μs	**6.91e-02**
turbo:dvfs	<1 μs	<1 μs	<1e-06	14 μs	1 μs	<1e-06	40 μs	11 μs	3.30e-04
turbo:nic	3 μs	<1 μs	<1e-06	23 μs	1 μs	<1e-06	23 μs	10 μs	2.90e-02
dvfs:nic	-1 μs	<1 μs	<1e-06	-15 μs	1 μs	<1e-06	-58 μs	11 μs	<1e-06
numa:turbo:dvfs	2 μs	<1 μs	<1e-06	12 μs	2 μs	<1e-06	3 μs	16 μs	**8.70e-01**
numa:turbo:nic	<1 μs	<1 μs	**6.25e-01**	-7 μs	2 μs	8.00e-05	-14 μs	15 μs	**3.59e-01**
numa:dvfs:nic	3 μs	<1 μs	<1e-06	34 μs	2 μs	<1e-06	79 μs	15 μs	<1e-06
turbo:dvfs:nic	<1 μs	<1 μs	**7.66e-01**	-9 μs	2 μs	<1e-06	4 μs	14 μs	**7.96e-01**
numa:turbo:dvfs:nic	-8 μs	<1 μs	<1e-06	-43 μs	3 μs	<1e-06	-83 μs	23 μs	2.50e-04

optimizer from getting trapped in local optimal, because all explanatory variables are discrete values (*i.e.*, dummy variables). The perturbation is small enough that it does not affect the quality of the regression itself.

5.4.2 Memcached Result

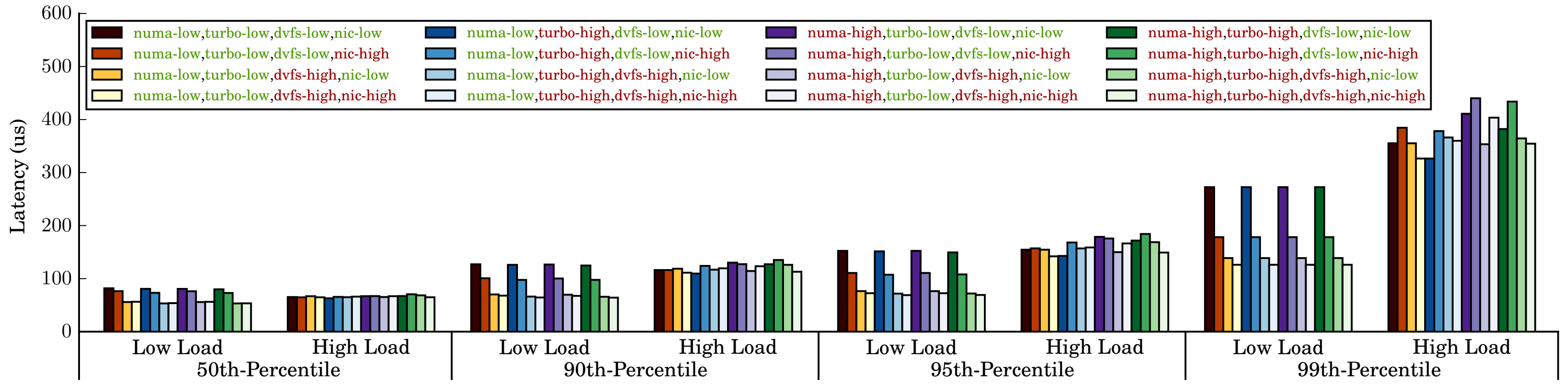


Figure 5.7: Estimated latency of Memcached at various percentiles under low utilization and high utilization using the result from quantile regression.

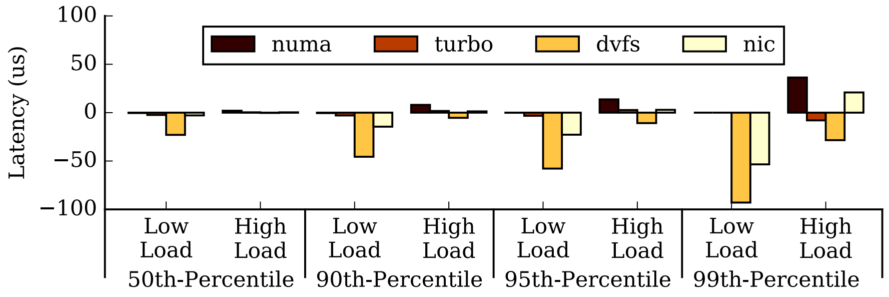


Figure 5.8: The average impact in latency of turning each individual factor to high-level for Memcached, assuming each of the other factors have equal probability of being low-level and high-level. Negative latency means latency reduction, and positive number means latency increase.

Table 5.4 shows the result from quantile regression for different percentiles, including 50th-, 95th- and 99th-, for Memcached workload at 70% server utilization. For each percentile, *Est.* shows the estimated coefficient (*i.e.*, $c_i(\tau)$ in Equation 5.1) of each factor, where negative value means turning the factor to high-level reduces the corresponding quantile latency. To estimate the quantile latency for a given hardware configuration, one needs to add up all the qualified estimated coefficients (*Est.* in the table) and the intercept. For example, to estimate the 95th-percentile latency for a configuration that only "numa" and "turbo" are turned to high-level, one needs to add their coefficients of them in isolation ($24 + -12 = 12\mu$s), and their interaction "numa:turbo" (5μs), and the intercept (155μs), therefore the estimated 95th-percentile latency is $12 + 5 + 155 = 172\mu$s. *Std. Err* is the estimated standard error for the coefficient estimation at 95% confidence interval. *P-value* is the standard p-value for null hypothesis testing, which is the probability of obtaining a result equal or more extreme than the observed one. A p-value smaller than 0.05 is usually considered as strong presumption against null hypothesis, which suggests the corresponding factor has a significant impact on the percentile latency.

To summarize the result, Figure 5.7 shows the estimated latency of all factor permutations at various percentiles under low and high server utilization correspondingly. From the result, we have several findings as follows:

- **Finding 1.** *The variance of latency increases from lower to higher server utilization, because of the increasing variance in number of outstanding requests.* This is similar to what we observe in a M/M/1 queueing model [88], that the variance of number of outstanding requests $\frac{\rho}{(1-\rho)^2}$, where ρ is the server utilization, grows as the utilization increases.

- **Finding 2.** *The variance of latency increases from lower to higher quantile as suggested by the growing standard error shown in Table 5.4, because the variance of a quantile is inversely proportional to the density [103].* This also explains the reason why we observe many statistically insignificant cases (p-value > 0.05) and the uncertainty is high at high quantiles.

- **Finding 3.** *The latency could be higher at lower utilization when the DVFS governor is turned to ondemand policy, because of frequent transitions among frequency steps.* The 50th- and 90th-percentile latencies are higher during low load than high load under ondemand DVFS governor. This is because requests have a higher probability of experiencing the overhead of transitioning from lower to higher frequency steps, whereas the CPU is kept at high frequency during high load and does not need many transitions.

- **Finding 4.** *Turning NIC affinity policy from same-node to all-nodes during low load can significantly reduce the latency when DVFS governor is set to ondemand.* The cores have larger utilization range under same-node policy than all-nodes, which leads to higher probability of experiencing frequency step transitions. This does not occur at high load because the utilization is already high enough that the number of frequency transitions is negligible. Prior study performed on the same hardware factors in isolation fails to capture such interacting behaviors among multiple factors due to the limitation of isolated study.

- **Finding 5.** *As shown in Table 5.4, the interactions among factors are demon-*

strated to have statistically significant impact on the latency as many of them have a p-value smaller than 0.05. In addition, the estimated coefficients of interactions are sometimes larger than individual factors, which means the interactions among factors can have higher impact on the latency. For example, turning NUMA control policy to interleave increases the 99th-percentile latency by 56μs as shown in the table, but its positive interaction with performance DVFS governor results in a 9μs improvement. These interacting behaviors are complicated and sometimes counter-intuitive, and cannot be captured by isolated studies of individual factors. Therefore, it is necessary to use statistical techniques like quantile regression to model the interactions.

Due to the interactions among factors, we cannot simply decompose the variance of the tail to each individual factor. However, by assuming all the other factors are randomly selected (*i.e.*, each factor has equal probability of being low-level and high-level), we can quantify the impact of each factor on average case as shown in Figure 5.8.

- **Finding 6.** *Interleaved NUMA control policy increases the latency by up to 44μs especially during high load.* This is caused by bad connection buffers allocation that majority of the server threads have their connection buffers allocated on the remote memory node, while same-node policy guarantees half of the threads get their buffers allocated on the local node. We only observe this behavior during high load because the high queueing delay magnifies the overhead of accessing remote memory node.

- **Finding 7.** *The amount of impact each factor contributes varies depending on the load levels.* For example, DVFS governor has the highest impact at low load, whereas NUMA policy is the biggest contributor to the variance at high load. This is caused by the complex interacting behavior among different features, which again, is not captured by isolated studies in prior works.

5.4.3 Mcrouter Results

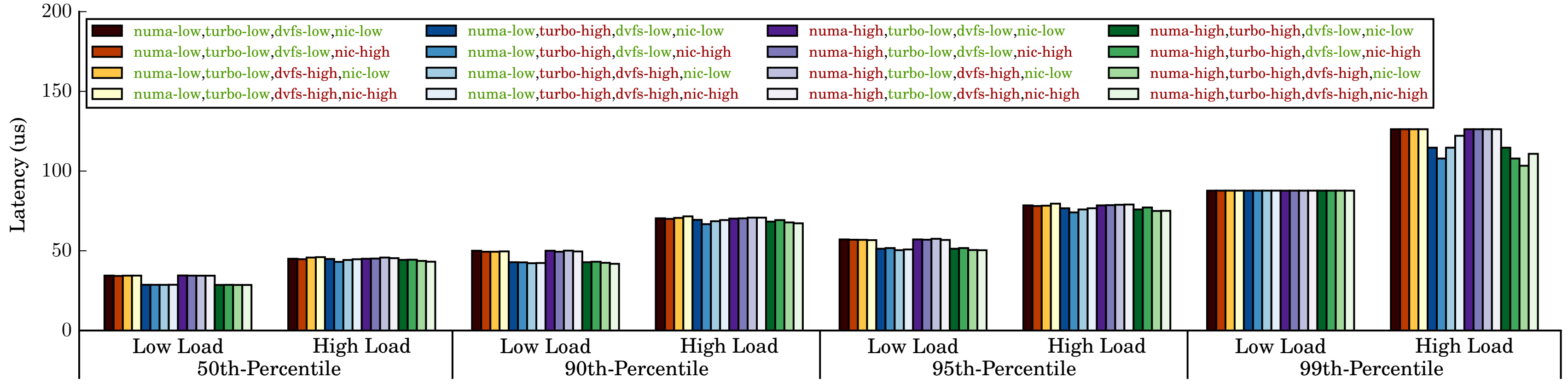


Figure 5.9: Estimated latency of mcrouter at various percentiles under low utilization and high utilization using the result from quantile regression.

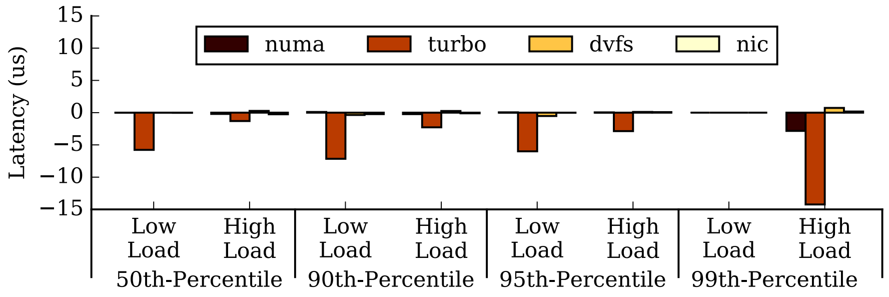


Figure 5.10: The average impact in latency of turning each individual factor to high-level for mcrouter, assuming each of the other factors have equal probability of being low-level and high-level. Negative latency means latency reduction, and positive number means latency increase.

Similarly, we also construct experiments with mcrouter workload as shown in Figure 5.9, which is a configurable protocol router that turns individual cache servers into massive-scale distributed systems. Figure 5.10 shows the average impact of the 4 factors assuming other factors are selected randomly with equal probability.

- **Finding 8.** *Turbo Boost significantly improves the latency especially during low load for mcrouter.* This is because a large fraction of the computation mcrouter needs to do is to deserialize the request structure from network packets, which is CPU-intensive and can easily be accelerated by frequency up-scaling. However, this difference is much smaller, sometimes statistically insignificant, during high load, because the available thermal headroom is smaller compared to low load.

5.4.4 Quantifying Goodness-of-fit

Although the low p-values obtained from quantile regression suggests high confidence that the studied factors have significant impact on the tail latency, it is also possible that they only contribute to a small fraction of the total variance. Therefore, we quantify the goodness-of-fit in this section, which demonstrates that our model covers the majority of the observed variance.

Figure 5.11 shows the pseudo-R^2 values, we previously defined in Equation 5.2,

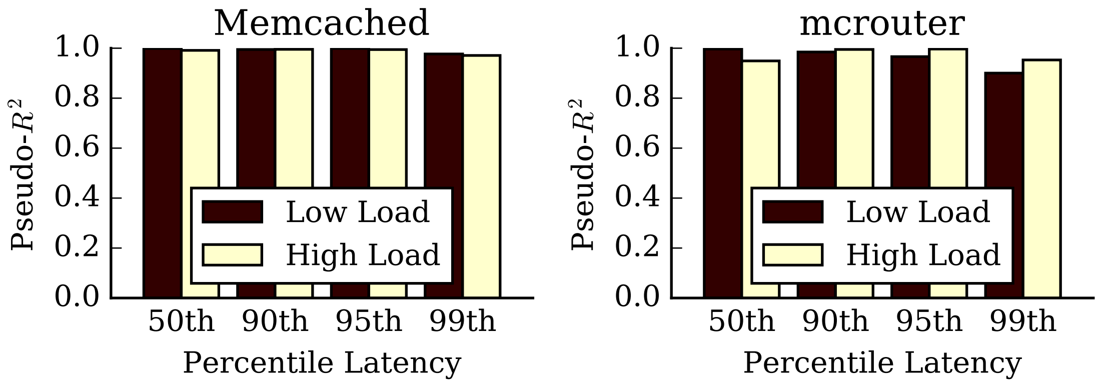


Figure 5.11: Pseudo-R^2 (shown in Equation 5.2) of the quantile regression results at various load levels and percentiles, which demonstrates good coverage of sources of variance. Pseudo-R^2 quantifies the goodness-of-fit of the model, which ranges in [0, 1] that higher value indicates better model fit. Our regression models show high pseudo-R^2 values (>0.90), which suggests that they are able to explain majority of the variance.

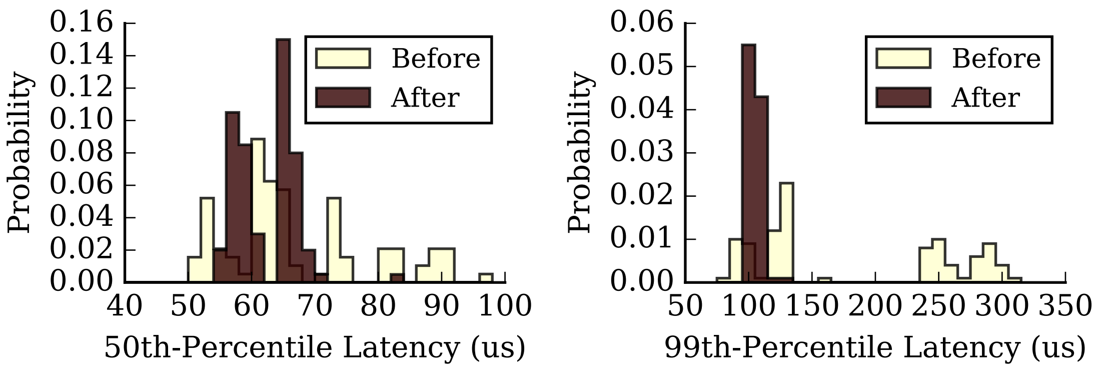


Figure 5.12: Using the knowledge we gain from quantile regression, both the latency and the variance of latency are significantly reduced after carefully controlling the factors contributed to the variance. The average 99th-percentile latency in 100 experiments is reduced from 181μs to 103μs, and the standard deviation is reduced from 78μs to 5μs.

of the quantile regression models at various percentiles, which quantifies the variance that can be explained. Our models have consistently high pseudo-R^2 values (the lowest one is 0.9), which suggests that they are able to explain the majority of the observed variance.

5.4.5 Improving Tail Latency

We further evaluate the quantile regression results in Figure 5.12, in which we perform the same experiment 100 times using randomly selected configurations as "before", and compare against the best configuration for 99th-percentile latency recommended by our quantile regression model as "after". As we can see from the figure, both latency and the variance of latency have been significantly reduced. Specifically, the expected 50th-percentile latency has been reduced from 69μs to 62μs, and the standard deviation has been reduced from 13μs to 5μs. The expected 99th-percentile latency has been reduced from 181μs to 103μs, and the standard deviation has been reduced from 78μs to 5μs. The reductions we achieve on 99th-percentile latency are much larger than on 50th-percentile, because we optimize for 99th-percentile when choosing the best configuration.

5.5 Summary

In this Chapter, we identify four common pitfalls through an in-depth survey of existing tail latency measurement methodologies. To overcome these pitfalls, we design a robust procedure for accurate tail latency measurement, which uses `Treadmill`, a modular software load tester we develop. With the superior measurements achieved by this procedure, we leverage quantile regression to analyze, and attribute the sources of variance in tail latency to various hardware features of interest. Using the knowledge we gain from the attribution, we reduce the 99th-percentile latency by 43% and its variance by 93%.

CHAPTER VI

TailSniping: Pinpoing Root Causes of Qos Anomalies

One of the key challenges in modern data centers is to enforce quality of service (QoS) requirements for user-facing services. Architectural and low-level system misconfigurations commonly cause QoS *performance anomalies* – significant unexplained QoS degradations. Unfortunately, conventional performance diagnosis techniques only provide application-level analysis, failing to diagnose the important hardware and low-level system issues, therefore these anomalies often remain unaddressed, causing sustained QoS violations.

In this Chapter, we introduce `TailSniping`, a lightweight infrastructure for automatically detecting, diagnosing and correcting hardware and low-level system configuration issues that impact QoS in data centers. Leveraging the massive amount of data collected by monitoring infrastructures common to modern data centers, our approach goes beyond conventional techniques by continuously mining application performance characteristics in production to detect performance anomalies in a low-overhead, statistically robust manner.

Unlike prior work that relies on correlation analysis to identify causes to anomalies, our approach enacts carefully designed causal inference micro-experiments to accurately pinpoint the root causes of the anomalies.

6.1 Performance Anomalies

In this section, we present a survey of common performance anomalies in data centers that are caused by system-level misconfigurations. These misconfigurations are common, yet challenging to diagnose for a number of reasons:

- The workloads of modern data centers are constantly changing as the user base grows, new products are launched, and software systems are frequently updated [19, 3].
- Many misconfigurations do not manifest themselves until the system is under certain states (*e.g.*, suboptimal frequency boost configurations may remain hidden until the server is under heavy load [181]).
- The heterogeneity (*i.e.*, composed of different generations of hardware [83, 121] and variety of network topologies [161]) of modern data centers further complicates system configurations.
- Identifying the optimal configuration in controlled offline experiments is extremely difficult, if possible, because production environment is much more complex [161].

Consequently, data centers commonly experience suboptimal performance, which results in performance anomalies, and even service outages [18, 5, 177].

6.1.1 Unexpected Interference

In modern data centers, many servers are provisioned to serve interactive services (*e.g.*, web search and social network) at a given QoS target. However, it is not uncommon that the low-priority co-locating workloads may generate unexpectedly significant interference, resulting in performance anomalies on the interactive services.

For example, modern data centers co-locate low-priority batch processing workloads with latency-critical interactive services to increase server utilization [164, 122, 63, 64, 171, 111]. However, co-location may cause performance anomalies for interactive services, when the interactive services demand more resources dynamically (*e.g.*, load spike) or the low-priority applications consume more resources than provisioned. In addition, maintenance activities (*e.g.*, data reconstruction in distributed file systems, garbage collections in managed languages) can cause significant performance anomalies in production environment as reported by Google [60].

6.1.2 NUMA Locality

As the working set size of the data center workloads grows, non-uniform memory access (NUMA) architecture has become ubiquitous. However, NUMA architecture introduces new challenges on managing memory locality due to the performance overhead of accessing remote memory node(s).

Many applications are designed and developed at a level of abstraction higher than explicitly managing NUMA, so it is not surprising that they experience performance anomalies due to suboptimal NUMA locality when the OS dynamically decides where to allocate memory at runtime [152, 167, 56, 111]. Furthermore, NUMA locality-caused performance anomalies can sometimes remain hidden unless the system is under certain states (*e.g.*, cross-socket memory bandwidth is saturated), which makes them hard to diagnose and debug [181].

6.1.3 Thread-to-Core Mapping

Managing thread-to-core mapping is particularly challenging given its complex interactions with other components of the system (*e.g.*, contention among application threads, cache coherence traffic, cross socket communication, remote NUMA accesses). The heterogeneity at both server-level and cluster-level further complicates the thread-to-core configuration. At server-level, logical cores (*i.e.*, SMT contexts) on the same physical core share private cache(s) and functional units, and cores on the same CPU socket contend for last-level cache and memory bandwidth and incur additional overhead when accessing remote memory node(s). At the cluster-level, many different machine generations are in operation at the same time, as reported by Google [121].

Given all the complexity, there is no single configuration that always yields optimal performance. Therefore, production systems can sometimes suffer from serious performance anomalies caused by suboptimal thread-to-core mappings, especially when the decisions are left to the OS dynamically at runtime [84] (*e.g.*, over-subscription, unnecessary cross socket communication).

6.1.4 Voltage Frequency Scaling

Power counts for a large fraction of the total cost of operating a data center. Therefore, many CPU dynamic voltage frequency scaling (DVFS) management systems [169] have been developed and deployed in data centers to save energy by downclocking. However, it is very challenging to always configure DVFS optimally, because applications prefer different voltage frequency scaling policies [111, 118, 120, 99] even under different states [99, 181] (*e.g.*, load level, interrupt queue-to-core mapping) due to its complex interaction with other components of the system.

For example, the `ondemand` DVFS governor, which lowers the CPU voltage frequency when the utilization is low, yields 2$\times$ degradation on 99th-percentile latency for Memcached during low load due to the overhead of waking the CPU up from lower frequency steps, while causing no degradation during high load [181]. The impact of the DVFS governor also varies drastically depending on whether the hardware interrupts are handled by one or all CPU sockets [181].

6.1.5 Network Interrupt Handling

Many interactive Internet services heavily rely on the network for communication, which requires significant amount of work on the processor to handle the hardware interrupts issued by the network interface card (NIC). Many vendors have integrated advanced features to optimize the mapping between the hardware interrupt queues (IRQs) on NICs and the CPU cores to handle the interrupts [7, 10], which requires careful tuning under various configurations [65, 131].

For example, spreading interrupts across all cores instead of having dedicated cores for interrupt handling can cause significant performance anomalies for Memcached (*i.e.*, $3\times$ degradation on tail latency [111]). However, this is not always true for all circumstances. For instance, spreading interrupt across cores becomes the preferable configuration when the interrupt flows can be directed to the correspondingly cores that process the queries using Flow Director [7].

6.1.6 Network Saturation

Due to the complex network topologies of modern data centers, it is not uncommon that network saturation sometimes causes performance anomalies [161]. These anomalies are hard to diagnose because they only manifest themselves when the load on certain components of the network is high enough. In addition, there have also been user-visible incidents caused by stale and inconsistent configurations in query routing systems [147, 113].

For example, unbalanced load can cause network saturation on certain regions of the network, resulting in long network latency and performance anomalies [161]. Such problem is extremely hard to investigate in controlled offline experiments, and can only be discovered when the production system is being exercised in specific ways. Additionally, data center operators (*e.g.*, Google [5] and Facebook [18]) have reported user-visible incidents caused by anomalies in network routing configurations.

Table 6.1: An example of measurements collected by the continuous monitoring infrastructure. Each record is indexed by *Time T*, *Server S* and *Application A*, and composed of a set of observations *O* (*i.e.*, *Memory Usage*, *CPU Frequency*, *LLC-misses/s*, *Load*) and a QoS metric *Q Latency*.

Time	**Server**	**Application**	**Memory**	**CPU Freq**	**LLC-misses/s**	**Load**	**Latency**
2016-04-10 16:36:01	server0	Memcached	42.4 GB	1.8GHz	4.24E6	802,210 req/s	47μs
2016-04-10 16:36:01	server1	Web-Search	442.6 MB	2.4GHz	1.22E6	782 req/s	57ms
2016-04-10 16:36:16	server1	Web-Search	467.7 MB	2.4GHz	1.57E6	791 req/s	76ms
2016-04-10 16:36:16	server1	MySQL	16.2 GB	2.4GHz	3.40E7	1,965 req/s	95ms

6.1.7 Load Fluctuation

The query load changes constantly in production data centers as user activity varies over time (*e.g.*, the diurnal pattern in a Google Web-Search cluster [124]), which results in varying application performance. This is not a performance anomaly, but can cause application performance degradation and a diagnosis system should not mistakenly attribute degradation due to load fluctuation to other factors.

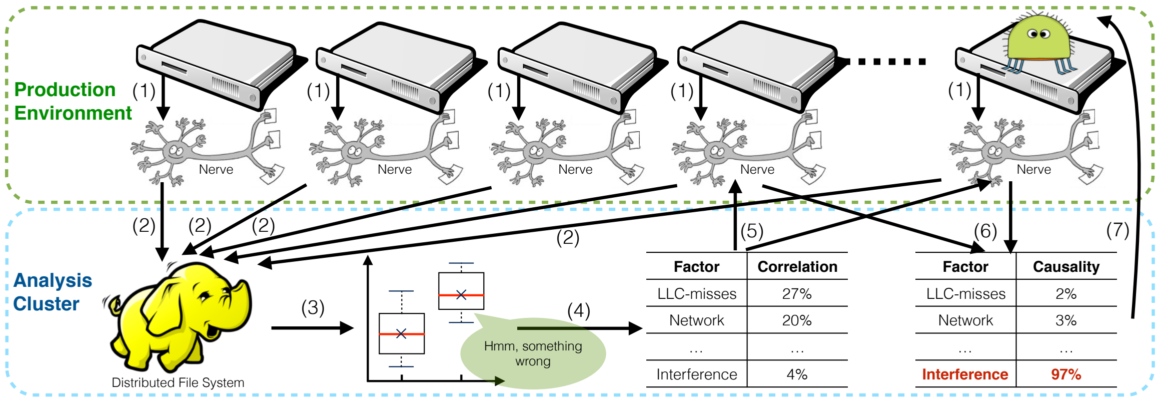
On the other hand, load fluctuation can also be caused by actual anomalies like poor load balancing. For example, a few TAO [42] caching servers at Facebook were running out of CPU cycles because they stored a significant fraction of the frequently-accessed data (*e.g.*, popular content), which resulted in long latency accessing the stored data, while other caching servers had many idle cycles [161].

6.2 TailSniping

To automatically diagnose and correct these performance anomalies, we build `TailSniping`. In this section, we first present an overview of `TailSniping` (Section 6.2.1). We then describe its two major components in detail: `Nerve` (Section 6.2.2) and `Brain` (Section 6.2.3).

6.2.1 Overview

Figure 6.1 presents an overview of `TailSniping`, which is composed of 2 major components: `Nerve` and `Brain`. `Nerve` continuously runs on each server in produc-



Factor	Correlation
LLC-misses	27%
Network	20%
...	...
Interference	4%

Factor	Causality
LLC-misses	2%
Network	3%
...	...
Interference	97%

Figure 6.1: Overview of `TailSniping` methodology – (1) `Nerve` continuously monitors runtime information on each server; (2) the measurements collected by `Nerve` is stored to the distributed file system; (3) `Brain` searches for performance anomalies by comparing statistical properties of the collected performance measurements; (4) a correlation analysis is performed to find all the correlating factors as the candidate causes; (5) `Brain` conducts one micro-experiment for each factor through `Nerve` to test the hypothesis that the candidate cause is the root cause; (6) a conclusion will be drawn once one hypothesis has been accepted; (7) the corresponding misconfiguration will be corrected in production environment.

tion environment to (1) monitor the runtime information and (2) store the collected measurements into a distributed file system. The analysis engine `Brain` runs in a separate analysis cluster, and asynchronously (3) mines the data to detect potential performance anomalies. Once a performance anomaly has been detected, `Brain` (4) performs a correlation analysis to find the correlating factors as candidate causes. Then `Brain` (5) conducts a micro-experiment for each candidate cause through `Nerve` to test the null hypothesis that it is the root cause of the observed anomaly. Once a hypothesis has been accepted, `Brain` (6) identifies the root cause, and (7) corrects the corresponding misconfiguration on the production servers.

6.2.2 Nerve

It is a common practice for data centers to deploy continuous monitoring infrastructure in production environment, such as GWP [133, 98] at Google and Scuba [23, 31] at Facebook. Similarly, `Nerve` is responsible for monitoring runtime information in the production environment (step (1) in Figure 6.1), storing the collected data to a distributed file system (step (2) in Figure 6.1), which will be used by `Brain` for detecting performance anomalies and diagnosing root causes, and executing micro-experiments invoked by `Brain` (step (5) in Figure 6.1).

6.2.2.1 Design Principles

There are 3 fundamental principles that we follow when designing `Nerve`.

- **Coverage:** Given the large amount of resources available on these production systems, the infrastructure needs to collect a rich set of runtime information to cover the possible causes of anomalies.
- **Generality:** The infrastructure needs to remain generic, because the servers in the production environment can be provisioned to serve various applications.
- **Low Overhead:** In order to be continuously deployed in production environment, the infrastructure needs to have an overhead no more than 1-2% [133].

6.2.2.2 Data Collection

Table 6.1 shows a simplified example of measurements collected by such monitoring infrastructure, in which each record is indexed by a time T, a server S and an application A. In each record, there are five types of measurements:

- **Time T:** The time when the record is collected (column *Time* in Table 6.1).
- **Server S:** The server that the record is collected on (column *Server* in Table 6.1).
- **Application A:** The application that the server is provisioned to run (column *Application* in Table 6.1).

- **Observations O:** Runtime statistics exposed by hardware (*e.g.*, performance monitoring unit measurements), the operating system (*e.g.*, statistics about running processes in `/proc/`), and the application (*i.e.*, request load information). For example, this includes columns *Memory Usage*, *CPU Frequency*, *LLC-misses/s*, and *Load* in Table 6.1).

- **QoS Q:** Quality of service metrics specified by the application (*e.g.*, column *Latency* in Table 6.1). The metric varies depending on the nature of the applications. For example, web search engines often use query latency as their QoS, and video streaming applications typically define their QoS as the number of frames being served per second.

When multiple applications are co-locating on the same server, there will be multiple entries in the table indexed by the same T and S, but different A (*e.g.*, 2nd and 4th rows in Table 6.1). We define $\Re(T, S, A, O, Q)$ as the relation of the measurements collected by the monitoring infrastructure, and the same notation will be used through out the paper.

To achieve high coverage of the potential root causes, `Nerve` collects data from 3 primary sources, covering architectural and system-level runtime information, to facilitate root cause diagnosis.

- **Hardware:** `Nerve` collects hardware runtime statistics (*e.g.*, CPU core frequency, LLC-misses/s) using a configurable set of hardware performance monitoring units (PMUs) with `libpfm` [12] as parts of observations O in $\Re(T, S, A, O, Q)$.

- **Software:** We monitor process-specific information exposed by the operating system in `/proc/` [15] (*e.g.*, CPU utilization, context switches) also as parts of the observations O in $\Re(T, S, A, O, Q)$.

- **Application:** We also pull information from the applications that the servers are provisioned to serve. This includes the request load information (*e.g.*, number of requests per second) as parts of the observations O in relation $\Re(T, S, A, O, Q)$, and the QoS metric Q (*e.g.*, query latency) in the relation. By collecting the request load information, we can potentially attribute the source of "performance anomaly" to the fluctuating load of the application itself, which commonly occurs in data centers [124].

6.2.2.3 Anomalies vs. Performance Variation

Due to the complex nature of modern systems, it is common that their performance behavior is non-deterministic to a degree. However, this expected variability is different from the unexpected degradation caused by performance anomalies. Therefore, we need a formal definition of the performance anomaly that is robust to differentiate between the two.

We use $\underset{T_i,S_j,A}{\Phi}(Q)$ to represent a particular statistical property (*e.g.*, mean, median, 99th-percentile) computed over all the QoS metric measurements Q indexed by time T_i and server S_j for application A. A performance anomaly is defined as the scenario that the difference between the computed statistical properties at different time T_i and T_k or on different server S_j and S_l for the same application A is *statistically significant*. The *statistical significance* is defined in Equation 6.1, in which $\sigma(\underset{T,S,A}{\Phi}(Q))$ denotes the standard deviation of $\underset{T,S,A}{\Phi}(Q)$.

$$\left| \underset{T_i,S_j,A}{\Phi}(Q) - \underset{T_k,S_l,A}{\Phi}(Q) \right| > \sigma(\underset{T,S,A}{\Phi}(Q)) \tag{6.1}$$

Using this equation, we only conclude a performance anomaly when the variability of the statistical property $\underset{T_i,S_j,A}{\Phi}(Q)$ computed over Q is larger than its empirically measured standard deviation. Intuitively, the statistical significance is used to avoid concluding a performance anomaly when the variability is expected and caused by the nature of the system.

6.2.3 Brain

`Brain` continuously mines the data collected by `Nerve` to detect potential performance anomalies (step (3) in Figure 6.1). When an anomaly has been detected, `Brain` diagnoses the root cause using correlation analysis (step (4) in Figure 6.1) and causal inference (step (5) and (6) in Figure 6.1). Once the root cause has been identified, the system can automatically correct the corresponding misconfigurations in the production environment (step (7) in Figure 6.1).

In this section, we focus on *anomaly detection* (Section 6.2.3.1), and *anomaly diagnosis* (Section 6.2.3.2) techniques.

6.2.3.1 Anomaly Detection

Design Principles

The goal of anomaly detection is to draw a conclusion on whether 2 sets of performance measurements X and Y are significantly different, that in statistical terms they are most likely to be drawn from different populations. Naively, we can use Equation 6.1 to serve this purpose, when we have the entire population of both QoS measurements $Q(X)$ and $Q(Y)$ (*e.g.*, latency of every single request in sets X and Y). However, QoS measurements are often down-sampled to reduce the overhead due to the large number of requests pumping through the system. For example, a typical `Memcached` server can serve up to hundreds of thousands of requests per second, that recording the latency of every single request is simply infeasible. Therefore, the key principles for designing a performance anomaly detecting technique include:

- **Coping with Sampled Data:** Because only a subset of the QoS measurements Q are presented, the main challenge is that we no longer have access to the entire population. Therefore, we need a technique that copes with sampled data, while still providing high statistical confidence levels to ensure that the anomalies identified are genuine.

- **Robust to Various Statistical Properties:** It is common that data center applications define their QoS using different statistical properties (*e.g.*, average latency, 99th-percentile latency). A mechanism that is robust enough to handle various statistical properties is required.

Idea

To cope with sampled data, we leverage statistical hypothesis testing techniques to determine whether two sets of QoS metrics Q are statistically significantly different. Specifically, we use a standard parametric hypothesis testing technique, unpaired Student's t-test procedure [104] with significant amount of samples (*e.g.*, >30), when the QoS metric is defined as the average performance (*e.g.*, average latency). Because non-normally distributed data commonly exists in computer systems [57], we cannot apply Student t-test when only small number of samples is given, which requires the underlying population being normally distributed. Therefore, we use Student t-test with significant amount of samples (*e.g.*, >30), because Central Limit Theorem provides the guarantee that the sampled mean will always follow a normal distribution even if the original population is not normally distributed [135]. However, there is no such guarantee when the QoS metric is defined as certain quantiles (*e.g.*, median, 99th-percentile latency). Therefore, we leverage a non-parametric hypothesis testing technique developed by Campbell and Gardner [43], which does not make any assumptions about the underlying distribution of the data, thereby can handle non-normally distributed data.

Technique

When the QoS metric is defined as the average performance, we directly apply the standard unpaired Student's t-test procedure [104] with a significant sample size. When the QoS metric is defined as certain quantile of the distribution (*e.g.*, 99th-percentile latency), we leverage a non-parametric hypothesis testing technique developed by Campbell and Gardner [43], which is robust enough to handle non-normally distributed data.

Suppose we would like to compare 2 sets of QoS measurements $Q(X)$ and $Q(Y)$, and draw a conclusion on whether their q-th quantiles are significantly different at a confidence level $(1-\alpha)$. We first compute the r and s quantity for both $Q(X)$ and $Q(Y)$ using Equation 6.2, in which n is the number of samples, and $N_{1-\frac{\alpha}{2}}$ is the corresponding value from the standard Normal distribution at the $(1-\frac{\alpha}{2})$ quantile (*e.g.*, 1.96 for $\alpha = 0.95$).

$$\begin{aligned} r =& nq - (N_{1-\frac{\alpha}{2}}\sqrt{nq(1-q)}) \\ s =& 1 + nq + (N_{1-\frac{\alpha}{2}}\sqrt{nq(1-q)}) \end{aligned} \tag{6.2}$$

We round r and s to the closest integers as $[r]$ and $[s]$. Then we find the $[r]$-th sample $Q_{[r]}$ and $[s]$-th sample $Q_{[s]}$ in increasing order from Q as our confidence interval at confidence level $(1-\alpha)$.

Then we compute the confidence interval for both sets of the q-th quantile at confidence level $(1-\alpha)$ as $(Q^q_{[s]}(X), Q^q_{[r]}(X))$ and $(Q^q_{[s]}(Y), Q^q_{[r]}(Y))$. Therefore, we have the confidence interval of their difference in q-th quantile as shown Equation 6.3:

$$\Big(Q^q_{[s]}(X) - Q^q_{[s]}(Y), Q^q_{[r]}(X) - Q^q_{[r]}(Y)\Big) \tag{6.3}$$

If this interval does not include 0, we reject the null hypothesis and conclude that there is a statistically significant difference between the q-th quantiles of $Q(X)$ and $Q(Y)$, which means a performance anomaly has been detected. Otherwise, the anomaly detection cannot draw any conclusion, and `Brain` continues to scan more data for anomalies.

6.2.3.2 Anomaly Diagnosis

Design Principles

The key task of the diagnosis engine is to accurately diagnose the root causes of the detected performance anomaly. The biggest challenge is to be able to identify the actual cause from the large number of correlating symptoms due to the complex interaction among resources. For example, lowering the CPU frequency will degrade the performance, as well as causing many other symptoms (*e.g.*, decreasing instructions/s, LLC-misses/s). As a result, correlation analysis will attribute the degraded performance to all the correlating factors including the symptoms. Therefore, the key design principle is to be able to identify the root causes from many correlating factors.

- **Root Cause Identification:** Correlation analysis has been commonly used by prior works [52, 165, 168, 170, 101, 127] in various studies, but it is not capable of diagnosing the root causes. To identify the root causes, a technique that infers true causal relationship rather than correlation is needed.

- **Unknown Factor:** It is impractical to cover all the possible causes of all performance anomalies. Therefore, it is important for the technique to be able to determine the actual root cause cannot be attributed to any monitored causes (*e.g.*, network traffic is not being monitored when the root cause is misrouted network traffic), rather than attributing the cause to other factors.

Idea

To identify the root cause, `Brain` first performs *correlation analysis* on the observations collected under normal execution and during the detected anomaly to find all the correlating factors with the QoS Q as candidate causes. For each candidate cause, it automatically conducts a *micro-experiment* by artificially reproducing the candidate cause on an identical server, or suspending the candidate cause on the abnormal. When reproducing on an identical server, it constructs a *null hypothesis* that the runtime statistics population collected in the reproduction is not significantly different from the one collected during the anomaly. When suspending on the abnormal server, it constructs a *null hypothesis* that the runtime statistics population collected after the suspension is not significantly different from the one collected during normal execution. If it fails to reject the null hypothesis, which means the two populations are similar enough that they are most likely to be drawn form the same distribution, we conclude the corresponding candidate cause is the actual root cause. If all the candidate causes are rejected, we conclude the anomaly is caused by an unknown factor that is not being monitored, which suggests the data center operators to extend the list of monitored runtime statistics and candidate causes.

Suppose an unexpected co-locating application is slowing down the Web-Search engine due to interference, the CPU utilization of the co-locating application will appear to be correlated with the QoS of Web-Search. A micro-experiment will be performed by temporarily suspending the co-locating application, and a null hypothesis will be constructed to evaluate whether the runtime statistics population collected after the suspension is similar enough to the one collected during normal execution. As they are similar enough, the unexpected interference caused by the co-locating application is concluded as the root cause of the observed performance anomaly.

Technique

Correlation analysis: We use analysis of variance (ANOVA) when the QoS metric Q is defined as the average performance (*e.g.*, average latency), and quantile regression [105] as suggested by prior work [57] when Q is defined as quantiles (*e.g.*, 99th-percentile latency). Both techniques give us a list of correlating factors ranked by the variance each factor is able to explain, which intuitively corresponds the contribution of each factor to the QoS.

Micro-experiment: With the exception of unexpected interference anomalies, micro-experiments are conducted on a separate "normal-performing" server (*i.e.*, one isnt experiencing the anomaly) with identical hardware specifications. This mitigates the possibility of introducing additional slowdowns beyond the anomalous server.

- *Unexpected Interference:* Pause the co-locating application by its process ID.
- *NUMA Locality:* Migrate the memory pages using `migrate_pages` [13] and `move_pages` [14], and change thread-to-core mapping [16] to reproduce the exact memory allocation on a normal server.
- *Thread-to-Core Mapping:* Migrate threads to reproduce the exact same mapping on a normal server [16].
- *Voltage Frequency Scaling:* Change DVFS setting to reproduce the same setting on a normal server [2].
- *Network Interrupt Handling:* Change the network interface card configuration [9] to reproduce the exact same setting on a normal server.

Some candidate causes are hard to suspend on the abnormal server, so we reproduce the same anomaly on a normal server:

- *Network Saturation:* Send same amount of synthetic network traffic to the normal server using `iperf3` [11].
- *Load Fluctuation:* Route same amount of shadow request traffic to the normal server by configuring the request routing (routing shadow request traffic is a common practice in production data centers, *e.g.*, [17] using `mcrouter` [113], [161]).

Note that the micro-experiment methodology is generalizable to other QoS anomalies, and can be easily extended.

Null hypothesis testing: The goal of null hypothesis testing is to determine whether the two populations X and Y (*e.g.*, $\langle O_{abnormal}, Q_{abnormal} \rangle$ collected during the anomaly and $\langle O_{candidate}, Q_{candidate} \rangle$ collected during reproduction of the candidate cause on a normal server) are similar enough that they are most likely to have been drawn from the same distribution. If they are drawn from the same distribution, *candidate* is the root cause of the observed performance anomaly.

Because $\langle O, Q \rangle$ has many dimensions, a hypothesis testing technique that copes with multidimensional data is needed. Therefore, we choose to use the unpaired two sample Hotelling's T-squared test [95], which is designed for multivariate hypothesis testing.

Specifically, we first compute the T^2 value using Equation 6.4, in which $\overline{X}$ is the mean vector of X, n_X is the number of samples in X, and $\sigma(X)$ is the standard deviation vector of X.

$$\begin{aligned} T^2 =& (\overline{X} - \overline{Y})^T \left[S(\frac{1}{n_X} + \frac{1}{n_Y}) \right]^{-1} (\overline{X} - \overline{Y}) \\ S =& \frac{(n_X - 1)\sigma(X) + (n_Y - 1)\sigma(Y)}{(n_X - 1) + (n_Y - 1)} \end{aligned} \tag{6.4}$$

Similarly to F-test, we then compute the F-value, degrees of freedom $df1$ and $df2$ using Equation 6.5, in which k is the number of dimensions of X.

$$\begin{aligned} F &= \frac{(n_X + n_Y + k)T^2}{k(n_X + n_Y)} \\ df1 &= k \\ df2 &= n_X + n_Y - k - 1 \end{aligned} \tag{6.5}$$

Once we have the F-value and the degrees of freedom $df1$ and $df2$, we have simply look up the p-value from the F-distribution using Equation 6.6.

$$p\text{-}value = 1 - ProbF(F, df1, df2) \tag{6.6}$$

If the p-value is smaller than a small value α (usually 0.01 or 0.05), we reject the null hypothesis and conclude that X and Y are drawn from different populations. Otherwise, we fail to reject the null hypothesis and therefore X and Y are likely to have been drawn from the same population. Particularly in our use case, we conclude that *candidate* is the root cause of the performance anomaly if we fail to reject the null hypothesis. When all the null hypotheses have been rejected, which means none of the candidate factor is the actual root cause, we conclude that the observed performance anomaly is caused by an unknown factor.

6.3 Evaluation

In this section, we evaluate our `TailSniping` system's effectiveness in achieving the following goals.

- Low overhead of continuously monitoring (Section 6.3.2)
- High accuracy of performance anomaly detection and pinpointing the root causes (Section 6.3.3)
- High scalability of the diagnosis procedure (Section 6.3.4)
- High effectiveness in improving data center performance by correcting the anomalies (Section 6.3.5)

6.3.1 Experimental Setup

All experiments are conducted on real systems. The specifications of the servers used are shown in Table 6.2. We use five popular data center applications as our

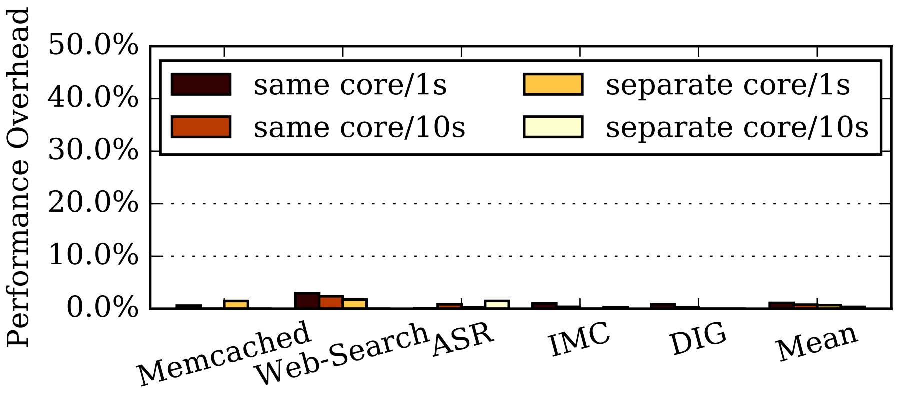


Figure 6.2: Normalized performance overhead of `Nerve`.

benchmarks as shown in Table 6.3. The events monitored by `Nerve` are listed Table 6.4. The events with a star in front (*e.g.*, app_{irq}) are monitored per process. The corresponding events of the application A in $\Re(T, S, A, O, Q)$ is denoted as app_*, and 3 other most CPU-intensive co-running processes are monitored and denoted as $\text{co-run}_*^{\text{k}}$ where k is the ranking in utilization. The rest of the events (*e.g.*, $\text{send}_{\text{bytes}}$) are monitored system-wide.

Table 6.2: Machine specifications.

Item	**Specification**
Processor	Intel Xeon E5-2407 v2 @ 2.40GHz
Microarchitecture	Ivy Bridge-EN
Memory	144GB @ 1333MHz
Operating System	Linux (kernel version 3.19.0)

6.3.2 Monitoring Overhead

We first evaluate the performance overhead of the continuous monitoring infrastructure `Nerve` to ensure it is deployable in production environment. Note that these measurements are already monitored in production data centers as a common practice [133, 98, 23, 31], so `Nerve` is not introducing any additional overhead. We quantify the worst case performance overhead of `Nerve` by running each of the applications at a high utilization (*i.e.*, 60%), which is significantly higher than the typical utilization

Table 6.3: Benchmark specifications.

Name	**Source**	Description
Memcached	CloudSuite [74]	data caching server
Web-Search	CloudSuite [74]	web search engine
ASR	DjiNN [89]	automatic speech recognition
IMC	DjiNN [89]	image classification
DIG	DjiNN [89]	digit recognition

Table 6.4: Events monitored by `Nerve`.

Name	**Description**
load	query load of the application
$\text{send}_{\text{bytes}}$	no. bytes sent through network
$\text{send}_{\text{packets}}$	no. packets sent through network
$\text{recv}_{\text{bytes}}$	no. bytes received from network
$\text{recv}_{\text{packets}}$	no. packets received from network
$^*\text{app}_{\text{irq}}$	avg. no. interrupts handled by app's cores
$^*\text{app}_{\text{freq}}$	avg. frequency of app's cores
$^*\text{app}_{\text{cpu\_util}}$	cpu utilization of app
$^*\text{app}_{\text{v\_ctxt\_sw}}$	no. voluntary context switches
$^*\text{app}_{\text{nv\_ctxt\_sw}}$	no. non-voluntary context switches
$^*\text{app}_{\text{io\_read}}$	no. I/O read operations
$^*\text{app}_{\text{io\_write}}$	no. I/O write operations
$^*\text{app}_{\text{vmem}}$	virtual memory usage
$^*\text{app}_{\text{rmem}}$	real memory usage
$^*\text{app}_{\text{cycles}}$	no. of cycles of app
$^*\text{app}_{\text{insts}}$	no. of instructions of app
$^*\text{app}_{\text{l1d\_miss}}$	no. L1 d-cache misses of app
$^*\text{app}_{\text{l1i\_miss}}$	no. L1 i-cache misses of app
$^*\text{app}_{\text{l2\_miss}}$	no.L2 cache misses of app
$^*\text{app}_{\text{llc\_miss}}$	no. LLC misses of app
$^*\text{app}_{\text{br\_miss}}$	no. branch mispredictions of app
$^*\text{app}_{\text{r\_numa}}$	no. remote NUMA accesses of app
$^*\text{app}_{\text{l\_numa}}$	no. local NUMA accesses of app
$^*\text{app}_{\text{pg\_fault}}$	no. page faults of app

levels these Internet services operate at (*i.e.*, 30% as reported by Google [36]). We also configure `Nerve` to sample at two different rate (*i.e.*, every 1s and every 10s), and in two different co-locating settings (*i.e.*, sharing the same cores as the application and running on a separate core).

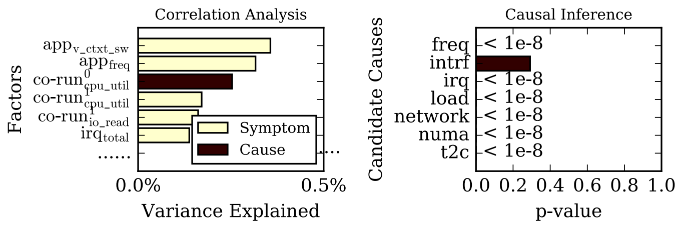
Figure 6.2 shows the normalized performance overhead of `Nerve` on the average latency. As we can see from the figure, the performance overhead `Nerve` introduces is negligible for most of the applications. Web-Search experiences slightly higher overhead than the other applications, due to the overhead of aggregating statistics of many utility child threads created by JVM (*e.g.*, garbage collection, signal dispatcher) as it is written in Java (*i.e.*, Apache Solr [1]). Overall, the performance overhead of `Nerve` is very small (*i.e.*, ¡ 1% even when sharing on the same cores as the applications). In addition to performance overhead, the data generated by `Nerve` is less than 1KB/s per application.

6.3.3 Performance Troubleshooting

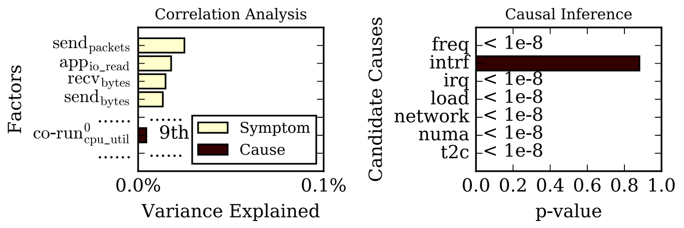
In this section, we evaluate the effectiveness of `TailSniping` in detecting and diagnosing the root causes of six hardware and system-level performance anomalies listed in Section **??**. Similar to prior work [25, 52, 165, 168, 49, 128, 134, 145, 33, 40, 146, 170, 34, 101, 173, 138, 127, 32, 137, 50], we construct experiments by artificially injecting performance anomalies. We also construct experiments to evaluate its capability of identifying unknown causes when the corresponding events are not being monitored (Section 6.3.3.8). In all experiments, it takes ∼30 seconds to detect the performance anomaly, and ∼30 seconds to conduct a micro-experiment, which is much more efficient than the manual diagnosis used in production [161]. We also present the results produced by state-of-the-art correlation techniques as baseline for comparison.

We focus on single-factor anomalies in our evaluation because single-factor anomalies are common and automatically diagnosing single-factor anomalies is still an open research question. That being said, we believe our technique can also be applied to multiple-factor anomalies by conducting micro-experiments with multiple factors combined.

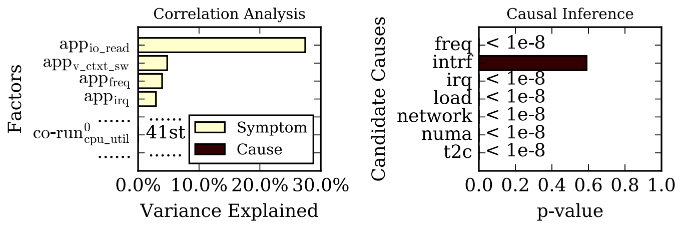
Result Interpretation



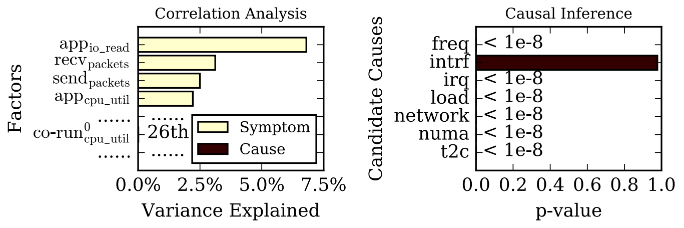
(a) Web-Search



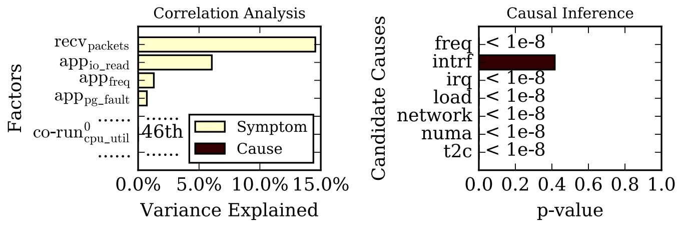
(b) Memcached



(c) ASR



(d) IMC



(e) DIG

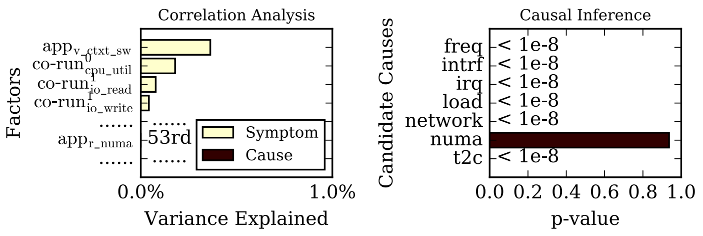
Figure 6.3: Unexpected Interference. `TailSniping` accurately pinpoints the root cause of the anomaly, while correlation analysis used in prior work fails to identify the actual root cause (*i.e.*, low in ranking).

For each experiment, we present a figure (*e.g.*, Figure 6.3a) with the ranked correlating factors produced by the *baseline correlation analysis on left*, and the p-values of all candidate causes generated by *causal inference on right*. The baseline correlation analysis ranks all the correlating factors based on their contribution to the performance anomaly (*e.g.*, the left figure in Figure 6.3a suggests $\text{app}_{\text{v\_ctxt\_sw}}$ contributes the most to the detected performance anomaly). The p-values produced by causal inference on right present the output of the null hypothesis testing for each candidate cause. A p-value greater than 0.01 suggests it is the root cause of the performance anomaly, otherwise it is only a correlating symptom (*e.g.*, the right figure in Figure 6.3a suggests the unexpected interference `intrf` is the root cause). In each figure, we also highlight the actual root cause denoted as *cause* using a darker color than the correlating factors denoted as *symptom*. In all figures, the unexpected interference is denoted as `intrf`, NUMA locality as `numa`, thread-to-core mapping as `t2c`, voltage frequency scaling as `freq`, network interrupt handling as `irq`, network saturation as `network`, and load fluctuation as `load` for short.

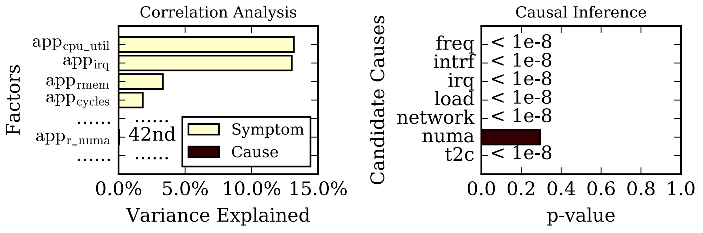
6.3.3.1 Unexpected Interference

We run `462.libquantum` from SPEC [91] as the source of unexpected interference. We find all five applications experience significant degradation caused by the interference, and the anomaly detection engine successfully detects all.

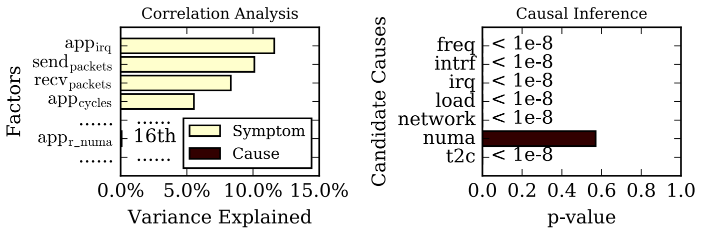
As shown in Figure 6.3, the baseline correlation analysis often ranks the actual cause $\text{co-run}_{\text{cpu\_util}}$ (*i.e.*, CPU utilization of the most CPU-intensive co-runner) very low among many other correlating symptoms (*i.e.*, 3rd for Web-Search, 9th for Memcached, 41st for ASR, 26th for IMC and 46th for DIG). However, the causal inference analysis is able to pinpoint the exact root cause (*i.e.*, the p-value of the null hypothesis that `intrf` is the root cause is much bigger than 0.01 while the p-values of other candidate causes are close to 0).



(a) Memcached



(b) ASR



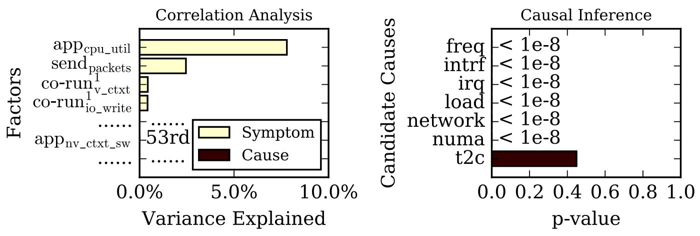
(c) IMC

Figure 6.4: NUMA locality.

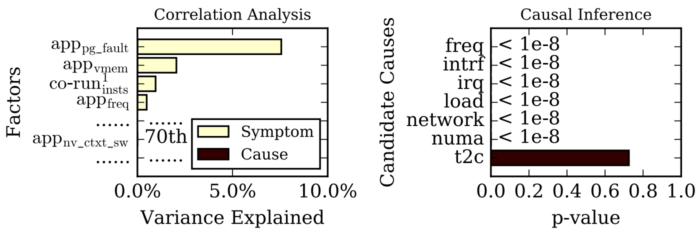
6.3.3.2 NUMA Locality

In this experiment, we introduce the performance anomaly by using a suboptimal NUMA allocation policy that increases the number of remote NUMA accesses. Only three out of the five applications experience performance degradation, which the anomaly detection engine is able to accurately detect. Web-search is not sensitive to remote NUMA accesses, which is consistent with the observation that Web-search has low memory bandwidth usage made by prior work [74]. The working set size of DIG is small enough to fit entirely into LLC, because the neural network used in the application has only 60K parameters.

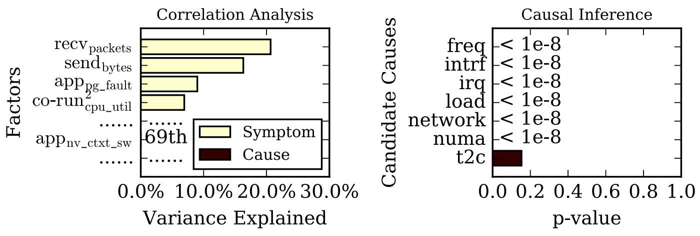
As shown on left in Figure 6.4, the actual root cause $\text{app}_{\text{r\_numa}}$ (*i.e.*, number of remote NUMA accesses) is again ranked relatively low by the baseline correlation



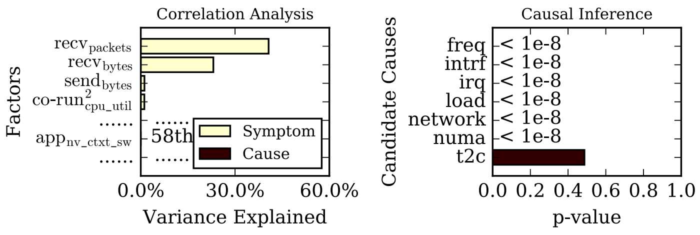
(a) Web-Search



(b) ASR



(c) IMC



(d) DIG

Figure 6.5: Thread-to-core mapping.

analysis (*i.e.*, 53rd for Memcached, 42nd for ASR and 16th for IMC). However, our causal inference procedure can accurately pinpoint the actual root cause as shown in the high p-values for candidate cause `numa`.

6.3.3.3 Thread-to-Core Mapping

In this experiment, we misconfigure the applications to run on less number of cores than the number of threads they have to introduce the anomaly. Four out of the five application as found to be sensitive to the misconfigured thread-to-core mapping, and the anomaly detection procedure detects all of them. Memcached does not suffer significantly from this particular thread-to-core mapping, which aligns with the observation made in prior work [109] that it does not suffer much from context switching unless the cores are over-utilized.

Figure 6.5 shows the results produced by the baseline correlation analysis on left, in which the root cause $\text{app}_{\text{nv\_ctxt\_sw}}$ (*i.e.*, number of involuntary context switches) has very low rankings (*i.e.*, lower than 50th for all applications). As shown by the p-values of the causal inference results on right, `TailSniping` accurately identify the exact root cause among many correlating symptoms.

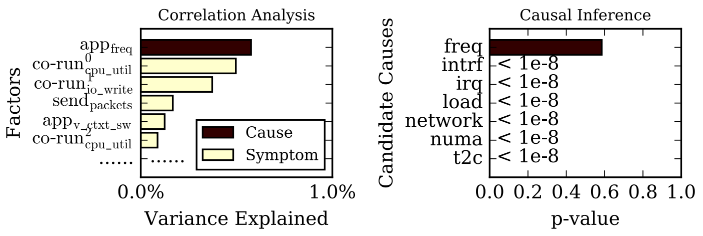
6.3.3.4 Voltage Frequency Scaling

In this experiment, we modify the DVFS governor to `ondemand`, which tries to lower the CPU frequency when its utilization is low, to introduce the performance anomaly. We find all five applications are sensitive to this misconfiguration, and the anomaly detection procedure successfully detects the anomaly.

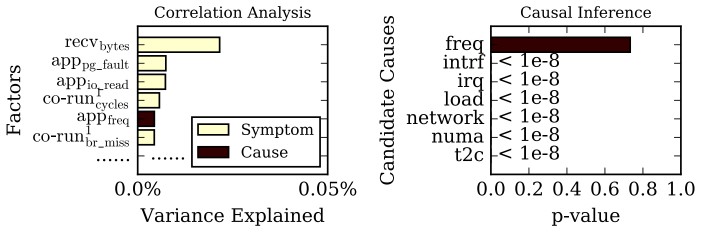
As shown in Figure 6.6, although the baseline correlation analysis ranks the root cause app_{freq} (*i.e.*, the average CPU frequency of the cores that the application runs on) relatively high, our causal inference technique does a strictly better job by accurately pinpointing the exact root cause.

6.3.3.5 Network Interrupt Handling

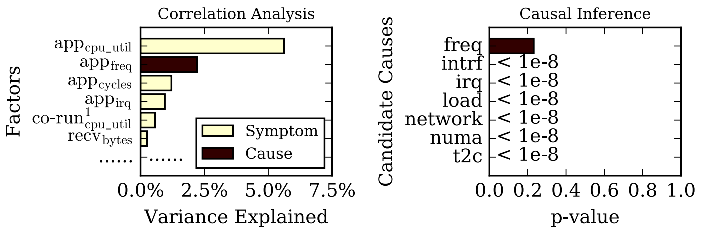
In this experiment, we configure the IRQ-to-core mapping to route the interrupts to the same cores as the ones that the application is running on, rather than having



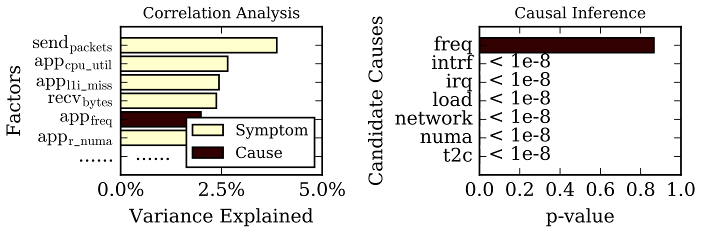
(a) Web-Search



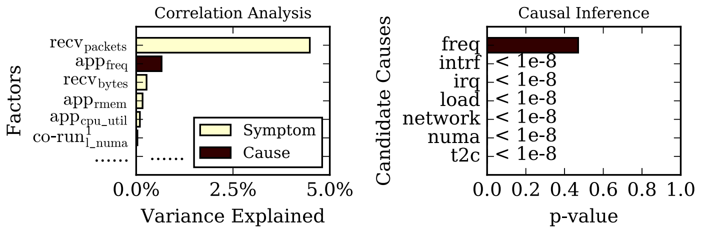
(b) Memcached



(c) ASR

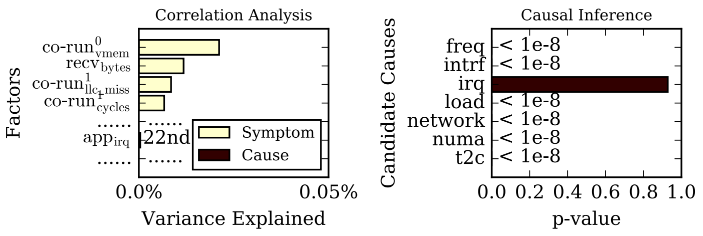


(d) IMC

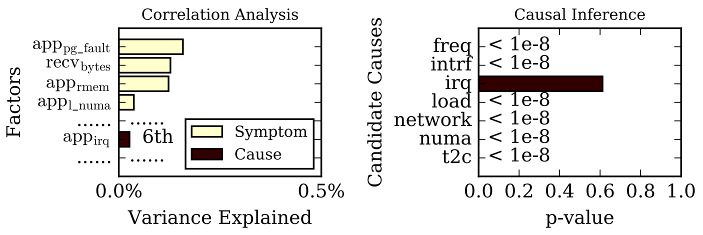


(e) DIG

Figure 6.6: Voltage frequency scaling.



(a) Memcached



(b) DIG

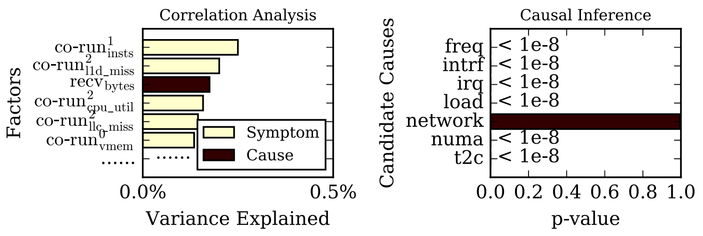
Figure 6.7: Network interrupt handling.

a dedicated core to handle the interrupts. Memcached and DIG are the only applications that we find to be sensitive to this misconfiguration, and the anomaly detection procedure successfully detects that. This is because the other three applications have much longer query latency that the relative slowdown they experience when waiting for the interrupts to be handled is negligible.

As shown in Figure 6.7, the baseline correlation analysis only ranks the root cause app$_{irq}$ (*i.e.*, average number of interrupts handled by the cores that the application runs on) the 22nd and the 6th, whereas the causal analysis identifies it as the root cause as indicated by the large p-values on right.

6.3.3.6 Network Saturation

In this experiment, we reproduce this anomaly by sending network traffic to the same server that the application is running on at a rate of 512Mbps. The anomaly detection engine precisely detects that the only application that experiences significant performance degradation is Memcached. This is because the other applications have much lower network usage, thereby are not sensitive to bandwidth contention.



(a) Memcached

Figure 6.8: Network saturation.

Figure 6.8 shows the results produced by the baseline correlation analysis on left, and the root cause recv$_{\text{bytes}}$ (*i.e.*, the number of bytes received over the network) is only ranked as the 3rd important factor. In contrast, the causal inference procedure strongly suggests it is the root cause of the observed anomaly by a p-value close to 1 in `network`.

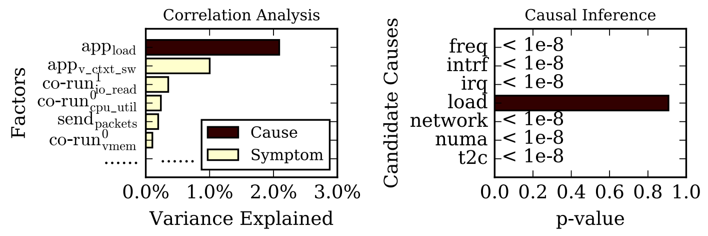
6.3.3.7 Load Fluctuation

In this experiment, we increase the application load by 25% and find the query latency of all five applications increases significantly, which is accurately captured by the performance anomaly detection procedure.

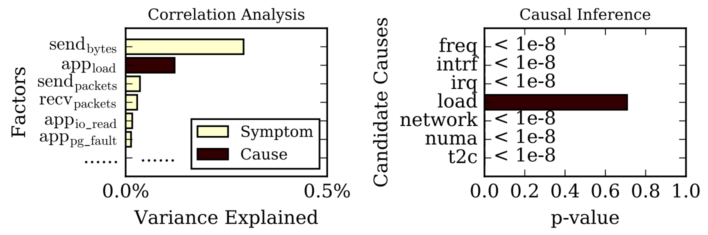
As shown in Figure 6.9, the baseline correlation analysis fails to point out the application load as the root cause for Memcached, ASR and DIG. However, our causal inference procedure again successfully pinpoints the exact root cause for all five applications as indicated by the p-values larger than 0.01 in `load` on right.

6.3.3.8 Unknown Factor

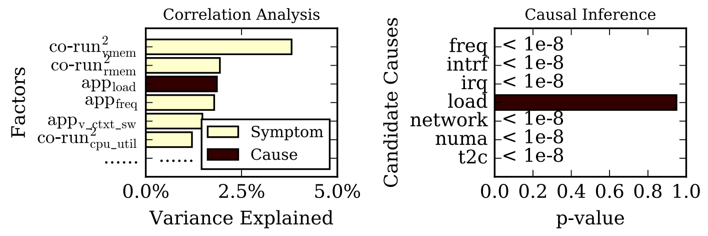
We also construct a set of experiments, in which we remove the actual root cause from the events being monitored (*e.g.*, removing CPU frequency-related measurements in the data when reproducing a voltage frequency scaling misconfiguration), to evaluate the effectiveness of `TailSniping` determining the root cause is not presented in the monitored events. In all cases, the causal inference procedure rejects all the



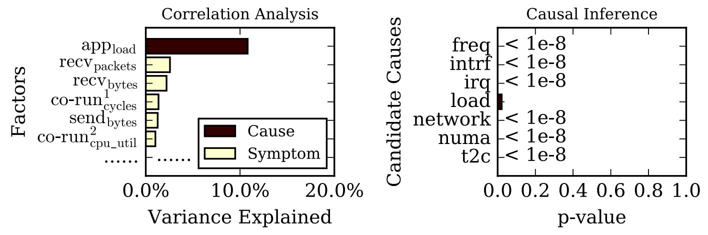
(a) Web-Search



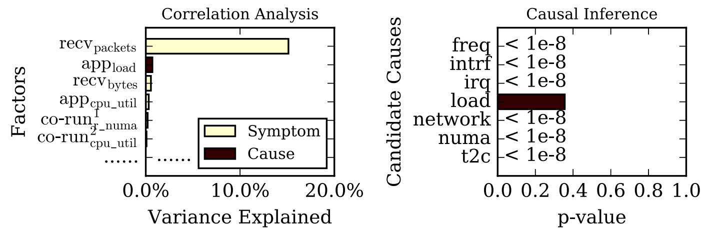
(b) Memcached



(c) ASR



(d) IMC



(e) DIG

Figure 6.9: Load fluctuation.

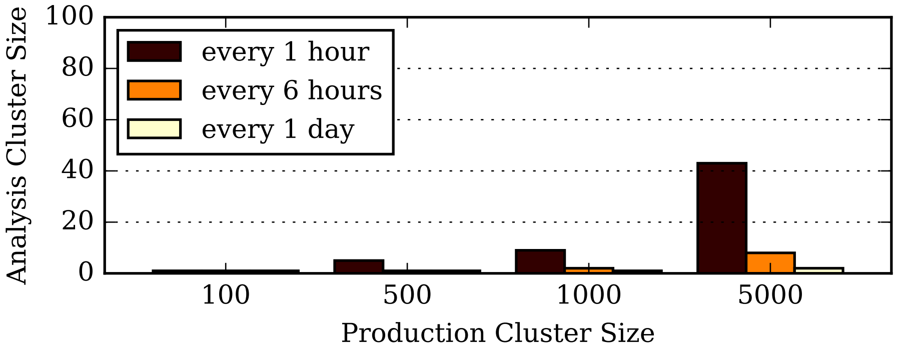


Figure 6.10: The number of servers needed in the analysis cluster in order to serve the analysis load with less than 10% probability of incurring any wait, at various production cluster sizes and various anomaly occurrence frequencies.

null hypotheses as the p-values are close to 0, and precisely concludes that the actual cause is not presented in the monitored events.

6.3.4 Scalability

In this section, we evaluate the scalability of `TailSniping` by quantifying the number of servers needed to perform the correlation analysis and the causal inference. We assume the servers used to perform the analysis are the same as the machines in the production environment as shown in Table 6.2, and each server experiences performance anomalies independently. The system is modeled by a `M/M/c` queueing system [88], where `c` is the number of servers in the analysis cluster.

Figure 6.10 shows the number of machines needed in the analysis cluster to perform the analysis with less than 10% probability of incurring any waiting time, as a function of production cluster size and the rate of performance anomaly occurrences. As we can see from the figure, the number of machines needed in the analysis cluster is negligible comparing to the number of machines in the production environment (*i.e.*, ¡ 1%). Only less than 10 machines are needed in the analysis cluster for a production cluster that has 1,000 machines.

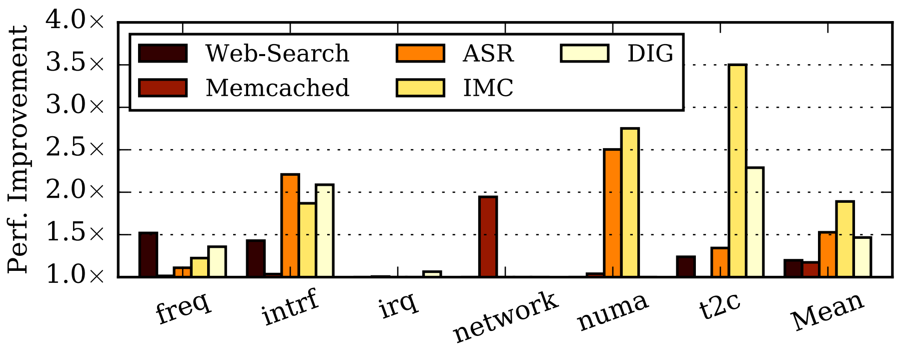


Figure 6.11: The performance improvement `TailSniping` achieves by successfully detecting, diagnosing and correcting the performance anomalies caused by low-level misconfigurations on various applications. It improves the overall performance by up to 3.5× and 1.5× on average.

6.3.5 Performance Improvement

In this section, we evaluate the performance improvement `TailSniping` achieves by successfully detecting, diagnosing and correcting various performance anomalies caused by hardware and low-level misconfigurations.

Figure 6.11 shows the performance improvement in average query latency, comparing to a data center that is not equipped with the proposed `TailSniping` system. As demonstrated in the figure, our technique can significantly improve the overall data center performance under various performance anomalies described by up to 3.5×, and 1.5× on average.

6.4 Summary

This Chapter presents `TailSniping`, a methodology for automatically detecting, diagnosing and correcting hardware and low-level system misconfiguration issues in data centers. We prototype `TailSniping` on real systems and demonstrate its effectiveness using five real-world data center applications. Our system can successfully detect, diagnose and correct a wide spectrum of performance anomalies, ranging from thread scheduling mismanagement to NUMA allocations to IRQ-to-core mappings, with under 1% monitoring overhead. By accurately diagnosing and correcting the root causes of the performance anomalies, we achieve 1.5× performance improvement on average.

CHAPTER VII

Conclusion

Modern data centers are built in massive scales to provide the computing resources to power the demand of cloud computing. However, it is well-known that they have been operated inefficiently, and their performance cannot sustain the fast growing future demand. This dissertation investigates the inefficiency of modern data centers, and proposes software and hardware solutions to architect future data centers for high efficiency and low latency.

Firstly, I conduct an in-depth analysis of performance interference on real-system SMT processors, and design `SMiTe`, a methodology that establishes a prediction model to precisely predict QoS degradation for SMT co-locations. Leveraging this methodology, I then build a cluster-level scheduler to enable SMT co-locations to improve data center efficiency without violating QoS requirements. Secondly, I characterize the dynamics of how resources are utilized in ten production data centers at Microsoft to extract insights and historical patterns. Based on these insights, I design, develop and deploy a system that makes intelligent decisions on task scheduling and data placement according to historical behavior of applications to further improve data center efficiency. Thirdly, I identify and empirically demonstrate four common pitfalls in existing methodologies for measuring tail latency, and propose a robust experimental methodology and a software load testing tool `Treadmill` to overcome these flaws to achieve microsecond-level precision tail latency measurements. The high precision measurements enable us to build statistical inference procedures to understand and attribute the source of tail latency. Lastly, I present a novel methodology that proactively enacts causal inference micro-experiments to diagnose the root causes of performance anomalies, whereas prior work using correlation analysis can only generate a list of correlating symptoms without providing any insight about causal relationship. With this novel methodology, I design and develop a scalable system, `TailSniping`, that automatically detects, diagnoses and corrects performance anomalies to increase the data center efficiency and reduce response latency.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Apache Solr. `http://lucene.apache.org/solr/`.

[2] CPU Frequency Scaling. `https://wiki.archlinux.org/index.php/CPU_frequency_scaling`.

[3] Engineering at Quora: Continuous Deployment at Quora. `https://engineering.quora.com/Continuous-Deployment-at-Quora`.

[4] Faban Performance Workload Creation and Execution Framework. `http://faban.org/`.

[5] Google API infrastructure outage incident report. `https://developers.googleblog.com/2013/05/google-api-infrastructure-outage_3.html`.

[6] Google Data Center PUE Performance. `http://www.google.com/about/datacenters/efficiency/internal/`.

[7] Intel Ethernet Flow Director and Memcached Performance. `http://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-flow-director.html`.

[8] Intel Next Generation Microarchitecture Codename Sandy Bridge: New Processor Innovations. Intel Developer Forum. 2010.

[9] Interrupts and IRQ Tuning.

[10] Introduction to Receive Side Scaling. `https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/introduction-to-receive-side-scaling`.

[11] iPerf – The network bandwidth measurement tool. `https://iperf.fr/`.

[12] libpfm – Improving performance monitoring on Linux. `http://perfmon2.sourceforge.net/`.

[13] Linux Programmer's Manual – migrate_pages. `http://man7.org/linux/man-pages/man2/migrate\_pages.2.html`.

[14] Linux Programmer's Manual – move_pages. `http://man7.org/linux/man-pages/man2/move\_pages.2.html`.

[15] Linux Programmer's Manual – Process information pseudo-filesystem. http://man7.org/linux/man-pages/man5/proc.5.html.

[16] Linux Programmer's Manual – sched_setaffinity. http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html.

[17] mcrouter – Shadowing setup. https://github.com/facebook/mcrouter/wiki/Shadowing-setup.

[18] More Details on Today's Outage. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919.

[19] Ship early and ship twice as often. https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920.

[20] SPEC jEnterprise. https://www.spec.org/jEnterprise2010/.

[21] SPEC SIP-Infrastructure 2011. https://www.spec.org/specsip/.

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2016.

[23] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into Data at Facebook. *Proc. VLDB Endow.*

[24] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into Data at Facebook. *Proceedings of the VLDB Endowment*, 2013.

[25] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[26] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[27] A. Alameldeen and D. Wood. Variability in Architectural Simulations of Multi-Threaded Workloads. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA)*, 2003.

[28] C. Aniszczyk. Caching with twemcache. https://blog.twitter.com/2012/caching-with-twemcache, 2012.

[29] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[30] J. Arulraj, G. Jin, and S. Lu. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[31] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[32] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[33] M. Attariyan and J. Flinn. Using Causality to Diagnose Configuration Bugs. In *Proceedings of the USENIX 2008 Annual Technical Conference on Annual Technical Conference (USENIX ATC)*, 2008.

[34] M. Attariyan and J. Flinn. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[35] L. A. Barroso, J. Clidaras, and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis lectures on computer architecture*, 2013.

[36] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[37] T. Bawden. Global Warming: Data centres to Consume Three Times as Much Energy in Next Decade, Experts Warn. *Independent*, 2016.

[38] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[39] R. Bertran, A. Buyuktosunoglu, M. S. Gupta, M. Gonzàlez, and P. Bose. Systematic Energy Characterization of CMP/SMT Processor Systems via Automated Micro-Benchmarks. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[40] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson. Lightweight, High-resolution Monitoring for Troubleshooting Production Systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[41] M. B. Breughe and L. Eeckhout. Selecting Representative Benchmark Inputs for Exploring Microprocessor Design Spaces. *ACM Transactions on Architecture and Code Optimization (TACO)*.

[42] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[43] M. J. Campbell and M. J. Gardner. Statistics in Medicine: Calculating confidence intervals for some non-parametric analyses. *British medical journal (Clinical research ed.)*, 1988.

[44] L. C. Carrington, M. Laurenzano, A. Snavely, R. L. Campbell, and L. P. Davis. How Well Can Simple Metrics Represent the Performance of HPC Applications? In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC)*, 2005.

[45] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term SLOs for Reclaimed Cloud Computing Resources. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[46] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 2006.

[47] F. J. Cazorla, A. Ramirez, M. Valero, P. M. Knijnenburg, R. Sakellariou, and E. Fernández. QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 2004.

[48] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 2008.

[49] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Faliure and Evolution Management. In *Proceedings of*

the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI), 2004.

[50] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[51] R. B. Clay, Z. Shen, and X. Ma. Accelerating Batch Analytics With Residual Resources From Interactive Clouds. In *Proceedings of the 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2013.

[52] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI)*, 2004.

[53] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[54] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You'Re Late Don'T Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[55] C. Curtsinger and E. D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[56] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[57] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney. Why You Should Care About Quantile Regression. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[58] M. De Vuyst, R. Kumar, and D. M. Tullsen. Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS)*, 2006.

[59] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 2013.

[60] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[61] C. Delimitrou and C. Kozyrakis. iBench: Quantifying interference for datacenter applications. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[62] C. Delimitrou and C. Kozyrakis. iBench: Quantifying Interference for Datacenter Applications. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[63] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[64] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[65] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[66] S. Eranian. Linux perf_events Subsystem Status Update. In *Petascale Tools Workshop*, 2013.

[67] S. Eyerman and L. Eeckhout. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. *SIGPLAN Not.*

[68] S. Eyerman and L. Eeckhout. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[69] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[70] X. Fan, W.-D. Weber, and L. A. Barroso. Power Provisioning for a Warehouse-sized Computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007.

[71] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at Facebook. *IEEE Internet Computing*, 2013.

[72] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. L1-bandwidth Aware Thread Allocation in Multicore SMT Processors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[73] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT)*, 2013.

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

[75] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004.

[76] A. Foundation. HDFS Architecture Guide, 2008.

[77] H. Fugal. Wangle. `https://github.com/facebook/wangle`.

[78] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the Annual Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, 2007.

[79] A. Goder, A. Spiridonov, and Y. Wang. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[80] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[81] I. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenHadoop: Leveraging Green Energy in Data-processing Frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[82] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM SIGCOMM Conference*, 2014.

[83] M. Guevara, B. Lubin, and B. C. Lee. Navigating heterogeneous processors with market mechanisms. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[84] M. Guevara, B. Lubin, and B. C. Lee. Strategies for anticipating risk in heterogeneous system design. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[85] P. Guide. Intel® 64 and IA-32 Architectures Software Developer's Manual. 2010.

[86] V. Gupta, M. Harchol-Balter, J. Dai, and B. Zwart. On the Inapproximability of M/G/K: Why Two Moments of Job Size Distribution Are Not Enough. *Queueing Systems*, 2010.

[87] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge. Integrated 3D-stacked Server Designs for Increasing Physical Density of Key-value Stores. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[88] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. 2013.

[89] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. DjiNN and Tonic: DNN As a Service and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[90] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[91] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 2006.

[92] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt. Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[93] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2015.

[94] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[95] H. Hotelling. *The generalization of Student's ratio.* Springer, 1992.

[96] C. H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[97] M. Isard. Autopilot: Automatic Data Center Management. *SIGOPS Operating Systems Review*, 2007.

[98] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Proceeding of the 42st Annual International Symposium on Computer Architecuture (ISCA)*, 2015.

[99] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks. Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2014.

[100] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[101] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box Problem Diagnosis in Parallel File Systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[102] H. Kasture and D. Sanchez. Ubik: Efficient Cache Sharing with Strict Qos for Latency-critical Workloads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[103] R. W. Keener. Theoretical Statistics: Topics for a Core Course. *International Statistical Review*, 2012.

[104] R. W. Keener. Theoretical Statistics: Topics for a Core Course. *International Statistical Review*, 2012.

[105] R. Koenker. *Quantile Regression.* 2005.

[106] M. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[107] M. A. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. M. Tikir, and S. Poole. Reducing Energy Usage with Memory and Computation-Aware Dynamic Frequency Scaling. In *Proceedings of the European Conference on Parallel Processing (EuroPar)*. 2011.

[108] D. Leonard. Is Your Data Center Sustainable? How To Proactively Prepare Your Business. *Forbes*, 2016.

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

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

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

[112] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[113] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani. Introducing Mcrouter: A Memcached Protocol Router for Scaling Memcached Deployments, 2014.

[114] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[115] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008.

[116] H. Lin, X. Ma, J. Archuleta, W.-C. Feng, M. Gardner, and Z. Zhang. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[117] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, 1988.

[118] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *Proceedings of the International Symposium on Computer Architecuture (ISCA)*, 2014.

[119] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[120] D. Lo and C. Kozyrakis. Dynamic Management of TurboMode in Modern Multi-Core Chips. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[121] J. Mars and L. Tang. Whare-map: Heterogeneity in "Homogeneous" Warehouse-scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[122] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.

[123] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action: Covers Apache Lucene 3.0.* Manning Publications Co., 2010.

[124] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-intensive Services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[125] D. Meisner, J. Wu, and T. F. Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[126] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[127] K. Nagaraj, C. Killian, and J. Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[128] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[129] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[130] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[131] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[132] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[133] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *Micro, IEEE*, 2010.

[134] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation (NSDI)*, 2006.

[135] J. Rice. *Mathematical statistics and data analysis*. Nelson Education, 2006.

[136] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

[137] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[138] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[139] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2006.

[140] B. Sharma, T. Wood, and C. R. Das. HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[141] K. Shen. Request Behavior Variations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[142] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 Multicore Server Processor. *IBM Journal of Research and Development*, 2011.

[143] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Mutlithreading Processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[144] L. Song and S. Lu. Statistical Debugging for Real-world Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014.

[145] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[146] Y.-Y. Su and J. Flinn. Automatically Generating Predicates and Solutions for Configuration Troubleshooting. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2009.

[147] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[148] L. Tang, J. Mars, and M. L. Soffa. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO)*, 2012.

[149] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[150] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. Reqos: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[151] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing Google's Warehouse Scale Computers: The NUMA Experience. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[152] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing Google's Warehouse Scale Computers: The NUMA Experience. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[153] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2009.

[154] Transaction Processing Performance Council. TPC Benchmarks.

[155] D. Tsafrir and D. Feitelson. Instability in Parallel Job Scheduling Simulation: the Role of Workload Flurries. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[156] D. Tsafrir, K. Ouaknine, and D. Feitelson. Reducing Performance Evaluation Sensitivity and Variability by Input Shaking. In *Proceedings of the 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2007.

[157] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[158] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, 1996.

[159] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995.

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

[161] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[162] A. Vega, A. Buyuktosunoglu, and P. Bose. SMT-centric Power-aware Thread Placement in Chip Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[163] A. Vega, A. Buyuktosunoglu, and P. Bose. SMT-Centric Power-Aware Thread Placement in Chip Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

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

[165] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with Peerpressure. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI)*, 2004.

[166] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench: A big data benchmark suite from internet services. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[167] W. Wang, J. W. Davidson, and M. L. Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[168] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging As Search: Finding the Needle in the Haystack. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI)*, 2004.

[169] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. Dynamo: Facebook?s Data Center-Wide Power Management System. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[170] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting Large-scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[171] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[172] X. Yang, S. M. Blackburn, and K. S. McKinley. Computer Performance Microscopy with SHIM. In *Proceeding of the 42nd Annual International Symposium on Computer Architecuture (ISCA)*, 2015.

[173] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[174] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010.

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

[176] S. M. Zahedi and B. C. Lee. REF: Resource Elasticity Fairness with Sharing Incentives for Multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[177] J. Zander. Final Root Cause Analysis and Improvement Areas: Nov 18 Azure Storage Service Interruption, 2014.

[178] W. Zhang, S. Rajasekaran, S. Duan, T. Wood, and M. Zhuy. Minimizing Interference and Maximizing Progress for Hadoop Virtual Machines. *SIGMETRICS Performance Evaluation Review*, 2015.

[179] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013.

[180] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[181] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.