Ph.D. DISSERTATION

Effective Performance Isolation for Consolidated Workloads

통합된 워크로드들을 위한 효과적인 성능 격리

August 2020

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yoonsung Nam

Effective Performance Isolation for Consolidated Workloads

통합된 워크로드들을 위한 효과적인 성능 격리

지도교수 엄현상

이 논문을 공학박사 학위논문으로 제출함

2020 년 7 월

서울대학교 대학원

컴퓨터 공학부

남윤성

남윤성의 공학박사 학위논문을 인준함

2020 년 7 월

위 원 장	염 헌 영	(인) [illegible]
부위원장	엄 현 상	(인) [illegible]
위 원	김 진 수	(인) [illegible]
위 원	이 재 욱	(인) [illegible]
위 원	이 재 환	(인) [illegible]

Abstract

As computer hardware advances, computing resources in a machine (e.g., the number of processing cores, cache sizes, and memory sizes) are increasing in various domains from the datacenter to embedded devices. To utilize these resources efficiently, it is necessary to execute multiple workloads in parallel by consolidating them for sharing system resources. However, the resource sharing may cause severe contentions for the shared resources, thereby the performance of workloads can be degraded significantly. Moreover, workloads may have different service level objectives (SLOs) from latency-critical to best-effort, which complicates the consolidation of workloads. To deal with the contentions, many OSs and hardware vendors have provided diverse resource isolation techniques. However, they have been used in perspective of fairness, not performance. Besides, the existing schemes do not provide or perform isolations efficiently and effectively. This dissertation presents feedback-based performance isolation optimizations that adaptively mitigate resource contentions by exploiting the available isolation interfaces in the existing OS and hardwares. To enable this, an efficient online profiling which estimates resource contentions is necessary. Also, isolations should be performed dynamically guided by the workloads' profiles.

In the dissertation, we propose performance isolation optimizations guided by online profiling for three systems. First, we present a performance isolation scheme that considers the charac- teristics of hardware and software isolation techniques for multicore systems. Second, we present an adaptive isolation optimization to mitigate mobile edge devices where resource are constrained and contentions may shift unexpectedly and frequently. Lastly, we propose a hierarchical contention-aware scheduling optimization for clusters, where provisions resources in fairness-centric manner, to mitigate contention when resource provisioning for latency-critical virtual machines (VMs). We have evaluated the proposed online profiling scheme and dynamic performance isolation scheme on diverse system environments (i.e., multicore server, edge devices, and clusters). Evaluation results show that our adaptive proposed approach can effectively track and mitigate resource contention for consolidated workloads, and thus can attain lower execution time (and latency) while achieving higher resource efficiency compared to the existing schemes under dynamic and significant resource contention.

Keywords: Performance Isolation, Resource Management, Resource Contention, Adaptive System, Multicore System, Edge Computing, Cluster

Student Number: 2014-30324

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Importance of Contention-Awareness

With the advance of the computer hardware, the size of computing resources within a machine are increasing. The number of CPU cores in a machine are growing to tens of cores, and the sizes of cache and memory are getting larger accordingly. In this trend, it is critical to share these resources by running multiple workloads (or tasks) in parallel to reduce wasted resources and maximize efficiency. From datacenter server to embedded devices, resource sharing becomes inevitable for efficient resource management and multi-process environments.

However, resource sharing may incur significant resource contentions among workloads. The resource contention can occur, from resources that are easy to allocate, such as CPU and memory, to ones that are hard to allocate, like last-level cache (LLC) and memory bandwidth. These resource contentions typically yield the following problems.

First, resource contention may affect negatively to the performance of workloads. Because resources may not be available when one workload try to utilize resources due to the other workloads already occupy them. In fact, workloads are typically not designed with a consideration that resources are not available on a machine. Besides that, for some resources such as LLC and memory bandwidth, most of machines does not have any allocation interface, thus it makes high performance variation according to the resource contentions among applications.

Second, resource contentions may lead to low resource efficiency by fragmented resources when reserving resources to avoid degraded performance. Resource reservation is straight-forward approach that allows sufficient resource allocation for workloads and avoids resource contention fairly. Even though it avoids contentions, it easily result in the low resource efficiency when resource demands are dynamically changed.

Third, resource contentions may increase the violation rate of service level objectives (SLOs) for workloads. Workloads generally have their own SLOs that can be either strict ones requiring low latency as much as few milliseconds or be relatively relaxed ones seeking best-effort performance. If resource contentions are significant, it may be impossible to meet strict SLOs without mitigating contentions. Although contentions are successfully reduced by using isolation techniques, if contentions are frequently changed, workloads may experience the frequent SLO violations frustrating their users. As a result, it leads to huge revenue loss in case of production services.

Obviously, resource sharing will be necessary and prevail in many domains to achieve higher resource efficiency. Therefore, effective and efficient isolation schemes that can detect resource contention and mitigate them is crucial for computer systems.

1.1.2 Problems

There have been many resource allocation and isolation techniques for handling resource contentions at OS or hardware level. However, most resource isolation techniques focus on resource fairness. Exploiting these isolation techniques for performance needs several optimizations. Because profiling contentions and estimating how these contentions affects the workload's performance is challenging. Identifying contentions is also related to choosing which isolation technique is appropriate for higher performance as well.

In this dissertation, we present several challenges to be addressed for effective performance isolation. First, existing isolation techniques should be fully considered for performance isolation. Many isolation techniques have different characteristics and configurations to improve isolation. Therefore, it is crucial to understand the difference among isolation techniques to choose proper isolation techniques. In addition, it is necessary to profile resource contention online and perform isolation techniques adaptively. Contentions may shift dynamically during the workload's execution and it may complicate isolation enforcement at runtime. Also, performance-feedback without contention estimation can lead to inefficient isolations. Thus, contentions should be handled in online manner to track and mitigate dynamic contentions. Finally, whenever resource provisioning is performed in cluster-level, it is necessary for frameworks not only improve resource efficiency, but also to consider contention-awareness for mitigating performance interference. Nevertheless, many cloud frameworks have not designed for mitigating contentions for performance of workloads.

These three challenges are related to optimizations for performance isolation and also related to the memory, one of the most contentious resources.

Multiple Isolation Techniques. Various resource isolation techniques have been exploited for performance isolation such as throttling CPU cycles, CPU core allocation, hardware cache partitioning, and so on. Nevertheless, there is no any consideration on which isolation technique is more effective in terms of attaining the higher performance. Typical multicore machines have several isolation techniques, and these isolation techniques have some tradeoffs in terms of strictness, responsiveness, and flexibility depending on the technique is software and hardware. For example, software-based throttling CPU cycles can be less effective than hardware-based per-core dynamic voltage frequency scaling (DVFS), because hardware isolation can directly control its hardware at low overhead as well as provide much faster isolation. Also, enforcing the right isolation technique guided by profiling can avoids inefficient isolation enforcements when multiple isolation techniques are used. Yet, current isolation schemes do not provide proper guide for the efficient isolation enforcement.

Dynamic Resource Contention. Resource contention is closely related to the patterns of resource usages that workloads consume. If they change continuously during the execution, the proper isolation technique to mitigate contention also should be changed accordingly. Therefore, online profiling and tracking contentions is necessary to know which contention affects negatively to the performance of workloads at runtime. Moreover, it is also crucial to perform isolation techniques adaptively and efficiently by the workload's profile. Edge computing environments including mobile edge devices have relatively constrained resources than clouds and are exposed to unexpected external events which incur high load variations for workloads. It is important how isolation scheme adaptively and effectively works under high and dynamic contention.

Contention-unaware Resource Provisioning. Cloud management frameworks are generally designed for maximizing resource efficiency in terms of resource capacity by bin-packing VMs or containers as much as possible. It seems reasonable in that reducing wasted resources achieves higher resource efficiency. However, the bin-packing approach may be harmful for performance of workloads due to resource contention. To avoid performance degradation while improving resource efficiency, frameworks should incorporate contention-awareness into their two-level resource provisioning that filters and weights nodes based on contentions, and mitigates contention within a node.

1.2 Contribution

The contributions are summarized as follows:

- We explore the tradeoffs between hardware isolation techniques and software ones in terms of the strictness, responsiveness, and flexibility. We present how inefficient isolation decisions affect performance degradation when consolidation. We design and implement a hybrid isolation system which adaptively isolates workloads considering the characteristics of workloads and tradeoffs in the isolation techniques. We evaluated that our system can improve the performance significantly compared with recent isolation scheme for the selected set of benchmarks.

- We identify the key challenges of multitasking on edge devices that have integrated SoC architecture and limited resources. We describe a profiling technique the measures dominant resource contention and also describe a practical phase change detection mechanism for the resource contention on edges. We describe the design and implementation details of *EdgeIso*, which uses several isolation techniques for effective performance isolation between tasks having diverse levels of SLOs. We evaluated *EdgeIso* on an NVIDIA Jetson TX2 device to compare with other alternative approaches using the selected set of benchmarks.

- We described the problem of inefficient resource provisioning mechanism in OpenStack and VMWare vSphere. Since cloud resource management is fairness-centric, we present a performance-centric contention-aware resource provisioning scheme that mitigates resource contention and improves performance of latency-critical in-memory VMs. We present the design and implementation of the hierarchical contention-aware scheduler and evaluated our scheme on OpenStack and VMware vSphere.

1.3 Dissertation Structure

- Chapter 2 introduces backgrounds for several isolation techniques supported by OSs and hardwares, architectural characteristics and challenges of various systems (e.g., multicore systems, edge devices, and clusters).
- Chapter 3 introduces `HIS`, our hybrid isolation scheme for multicore systems. We briefly presents the related work for multiple isolation techniques and describes the tradeoff between hardware and software techniques. We describe the design and implementation of our scheme and evaluate our scheme on consolidated multiple threaded workloads.
- Chapter 4 introduces `EdgeIso`, our effective performance isolation scheme for edge devices. We illustrate the challenges of resource-constrained edge devices (i.e., NVIDIA Jetson TX2). We describe the design and implementation of online profiler and adaptive isolation scheme, evaluate the scheme for data processing and latency-critical workloads, and compare ours with alternative approaches.
- Chapter 5 proposes a hierarchical contention-aware isolation scheme for clusters. We illustrate the problems of contention-unaware resource provisioning for various clusters (i.e, OpenStack and VMWare vSphere). We describe the method for measuring resource contention of in-memory latency-critical VMs (i.e., memcached). We describe the design and implementation of a contention-aware VM scheduling algorithm that provisions VM resources considering memory contention hierarchically at the intra node and inter node level. We evaluated our optimization for cluster by using mutilate, designed for testing various throughput and tail latency, and testing dynamic memory contentions.
- Chapter 6 concludes the dissertation and describes the future work.

Chapter 2

Background

In this chapter, we describe what is resource contention, what makes the contention, and why it should be controlled. We also describe the service level objectives which is a key criteria for how workloads are categorized and how aggressively workloads are consolidated. After that, this chapter provides backgrounds for the various isolation techniques supported by Linux kernel and hardware. Lastly, we explain some architectural characteristics and challenges of various systems including multicore systems, edge devices (integrated GPUs), and clusters.

2.1 Resource Contention

Resource contention can occur anywhere as long as resources are shared. However, the definition of resource contention in the dissertation is contention for resources shared by all cores, assuming a typical multi-core environment. Resource contentions typically occur in last-level cache, memory bandwidth, and memory shared by all cores. These shared resources are critical to memory-intensive workloads, which prevails in big data and AI domains.

In addition, another cause of resource contention usually stems from the fairness-centered resource allocation policy of the OS. Here, fairness is related to the capacity of the resources allocated. For example, Linux OS and OpenStack determines how many CPU cores or memory sizes can be allocated to processes, virtual machines, and container, based on the number of workloads that are consolidated in a machine.

However, this allocation approach is not proportional to LLC or memory bandwidth that are difficult to allocate. Because their allocation interfaces are not explicitly exposed so that one workload can saturate all resources depending on the characteristics of consolidated workloads, even if cores and memory are allocated fairly among workloads.

This is why contentions on shared resources lead to poor workload performance, and also to over-allocation of resources, creating resource inefficiency, to avoid performance degradation. In order to mitigate these resource contentions at the OS or hardware level, various resource allocation techniques are provided to enable isolations. Nevertheless, using these resource isolation techniques for performance isolation, it is necessary to measure contention accurately and provide resource isolation accordingly.

2.2 Service Level Objectives

Each workload typically has its own service level objective. While there are several ways to define the SLO, achieving these service level objectives is critical. SLOs can be defined by the number of instructions per cycle (IPC) or query per second (QPS) for measuring throughput, 99th percentile and 95th percentile tail latency for measuring worst case latency. Based on the criteria of these SLOs, workloads can be largely divided into two categories. One is latency-critical workloads which require low latency, and the other is best-effort one without any specific latency or throughput required.

Table 2.1: Comparison of the existing hardware and software isolation techniques

	Hardware isolation techniques		Software isolation techniques		
	Intel CAT [3]	Per-core DVFS [4]	CPU Cycle Limit [5]	CPU Allocation [6]	Thread Migration [6]
Type	Partitioning	Throttling	Throttling	Scheduling	Scheduling
Latency (ms)	3	2	40∼50	3	90
Configurations (Xeon E5-2683v4)	# of ways (20 per LLC)	# of available freq. (10 per core)	quota / period (100)	# of cores (16)	# of sockets (2)
Strictness	High	High	Medium	Medium	Low
Responsiveness	High	High	Medium	High	Low
Flexibility	Low	Low	High	High	High

Best-effort workloads are typically co-located with latency-critical ones to utilize idle resources which are not used by latency-critical ones to improve resource efficiency. When consolidating best-effort ones and latency-critical ones, it is essential to meet SLOs of latency-critical ones because of other co-located best-effort ones saturate shared resources.

2.3 Isolation Techniques

This section briefly describes the existing resource isolation techniques available in modern multicore systems. Resource isolation techniques can be categorized into two groups; one is software and the other is hardware. We will describe software and hardware isolation techniques and illustrates tradeoffs between these isolation techniques.

Table 2.1 shows the existing hardware and software isolation techniques. Most schedulers utilizes the software isolation techniques and hardware isolation techniques in the table. All isolation techniques can be categorized by three types; `Throttling`, `Scheduling`, and `Partitioning`.

2.3.1 Software Isolation Techniques

Software techniques reduce contentions among workloads by using software interfaces. `Throttling` and `Scheduling` are the representative types of software isolation techniques. `Throttling` is a broadly used to minimize performance interference by controlling the execution rates of contentious workloads among co-located ones. For example, Google CPI^2 throttles CPUs of background workloads to protect the performance of co-located production workloads [7]. Memguard restricts the memory accesses of the memory-intensive workloads based on assigned memory budget throttling CPU cycles [8]. Limiting CPU cycles is an efficient software isolation technique which throttle the execution of specific workloads [7–9]. The technique mitigates the contention for shared resources by limiting the number of cycles to *quota* within the configured *periods*. If the assigned cycles are exhausted during a period, the core will remain idle until the new period begins.

Another technique is mitigating contentions via `Scheduling`. Two techniques are mostly used for `Scheduling`. One is *CPU allocation*, and the other is *thread migration*. *CPU allocation* is simple, yet the effective software technique to isolate workloads. It works purely in software manner, and easily reduces the contention of shared resources. It allocates dedicated CPU cores to each workload to minimize resource contention among workloads. When allocating cores to workloads, it is critical to consider which workloads will be colocated with each other [10–14]. Because resource contention among workloads can grow or not depending on which workloads are co-located. When resource contentions can not be resolved by other isolations in a socket, *thread migration* can be helpful by migrating the most suffered workload to the less contentious socket (or machine). This can be helpful where exist severe contentions that `Throttling` can not mitigate. In contrast, in the cases of all the possible schedule pairs can not relax the contention, `Scheduling` may result in poor performance due to the unnecessary overheads as it would fail to find better workload pairs.

2.3.2 Hardware Isolation Techniques

Hardware techniques physically allocate resources to mitigate contentions among workloads or exploits specific hardware features equipped on recent multicore machines. Hardware techniques can provide fast and strict isolation compared with the software ones, because they directly control hardware interfaces. Besides, hardware techniques have lower latency than software ones. Because they have a fewer number of available configurations, which makes configuration search faster when enforcing isolations. There are two types of hardware isolation techniques; `Partitioning` and `Throttling`. `Partitioning` is a representative hardware isolation technique which strictly segregates resources for multiple workloads. For hardware partitioning, there are Intel Cache Allocation Technology (Intel CAT) for LLC way-partitioning [3] and Intel Running Average Power Limits (Intel RAPL) for limiting power consumption [15].

Another hardware isolation technique is `Throttling`-type one using dynamic voltage frequency scaling (DVFS). DVFS is originally designed to perform power management, however, owing to the advance of DVFS, voltage regulators on recent CPUs can adjust a voltage of each core in the CPUs. This enables faster and low-overhead controls for specific operations [16], thus this can enable fine-grained isolation for latency-sensitive workloads [17].

2.4 Architectural Characteristics and Challenges

The degree of resource contention or the type of contentious resource may vary depending on the system architecture. The following sections describe where the resource contention occurs by the architectures and explain the specific problem caused by resource contention in their environments and requirements for workloads.

2.4.1 Multicore Systems

As illustrated in figure 2.1, a multicore machine consists of CPU cores, L1/L2/L3 caches, memory controllers, prefetchers, and memory nodes [10, 13, 18]. Among these resources, as the number of CPU cores is increasing, a processor can have almost 20 CPU cores in a socket. Also, the number of processors is also increasing, and multicore systems generally have multiple processors. Proportional to the increasing number of CPU cores, the sizes of last-level cache and memory bandwidth are gradually growing, yet they have not grown as much as the number of cores [13].

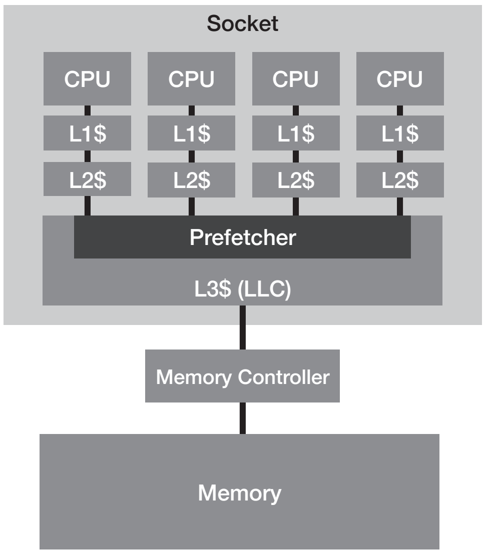


Figure 2.1: Multicore System Architecture

Therefore, if memory-intensive workloads are executing on these many CPU cores, it puts high memory loads on LLC and memory bandwidth, thereby degrading overall system performance. Consequently, the fact that memory resources are more demanded and memory-intensive workloads increase means that several indicators, showing the degree of resource contention, and isolation techniques to solve the problem are more critical than before.

2.4.2 Edge Devices

Latency-critical workloads rely on the caching of hot data for providing low latency or accelerating their services via accelerators. These make unpredictable latency for tasks when shared resource contention occurs and lowers the utility for the allocated resources such as CPU and memory. The resource contention becomes more significant and more critical in the world of edge devices. Figure 2.2 illustrates the resource contention on the NVIDIA Jetson TX2. It has four CPU cores, two superscalar (Den-

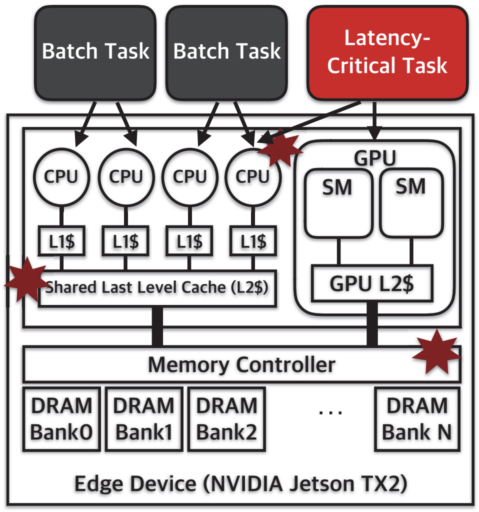


Figure 2.2: Shared resource contention between tasks on an edge device with integrated CPU-GPU architecture.

ver) CPU cores, and 256 GPU cores. As Jetson TX2 has an integrated CPU-GPU architecture, GPU shares memory and memory bandwidth with CPUs in the node. The integrated architecture worsens resource contention and also are found in other edge devices like NVIDIA Jetson Xavier [19] or Intel edge devices [20].

2.4.3 Clusters

As cloud computing has become popular and the demand for constructing the private or public clouds has been increasing, cloud platforms have evolved and many cloud projects have been suggested. Although these projects have been performed, the technologies for dynamic infrastructure reconfiguration or flexible resource sharing, which are the keys to Software-Defined Data Center (SDDC), have insufficiently been developed and used. To address this problem, we have designed and developed a Software-Defined Compute (SDC) framework based on the OpenStack [21], which is one of the most popular open-source cloud platforms. The framework performs profiling and scheduling to control the shared resources such as CPU and memory. For this reason, we focus on the SDC architecture. Figure 2.3 illustrates the overall high-level SDC architecture. In this architecture, we fill the gap between the traditional datacenter resource management framework and ideal SDC. For more flexible and efficient resource usage, it is important to capture the behavior of workloads and schedule the workloads based on the behavior more dynamically. In an SDC framework, all computing resources, including CPU and memory, are reorganized in a flexible manner or coordinated by the centralized/decentralized controller in order to achieve the specific operational goal such as guaranteeing the SLOs. In contrast to the architecture of traditional datacenter, an SDC one requires the components, such a performance monitoring tool and a coordinator, providing the SDC functionality. The subsequent subsections will describe the profiler and scheduler components of our SDC framework.

Profiling for SDC. Identifying the behavior of workloads is inevitable and necessary to support the SDC. The profiler should monitors the behavior of workloads, and if the performance degradation is detected, then a recovery action should be taken. However,

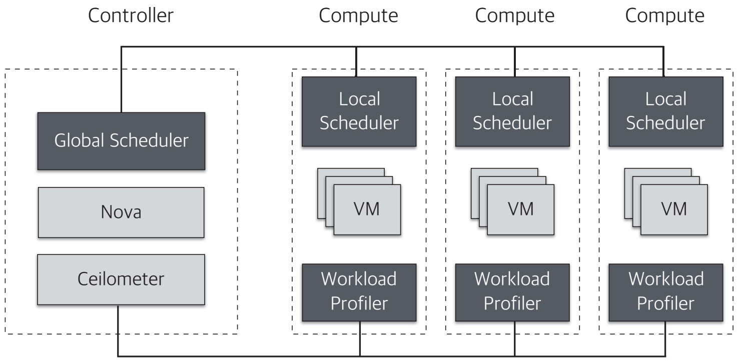


Figure 2.3: Proposed Software-Defined Compute Architecture. The components colored in black are the ones newly added for workload profiling and scheduling. The components colored in grey are the OpenStack compute (Nova), resource monitor (Ceilometer) and VMs.

there is no profiler to detect the performance anomaly in OpenStack. There exists the component for resource metering called Ceilometer, but this component simply counts the resource availability for billing. Currently, Ceilometer provides only alarm services which report the resource states to the user. The alarm services monitor the resource consumption of workloads and if the consumption exceeds the predefined threshold, then the services notify the user of that situation or trigger the additional actions such as autoscaling. Figure 2.4 shows the high-level architecture of the proposed workload profiler. It consists of two components; one is the performance metric monitor which collects the resource consuming stats such as memory contention information, and the other is the interference estimator, which evaluates the interferences among VMs and hosts, and propagates the evaluated information to other services. With the profiler, the cloud platform can monitor the performance degradation and leverage the performance metrics to schedule workloads.

Resource Management for SDC. Existing resource management frameworks have

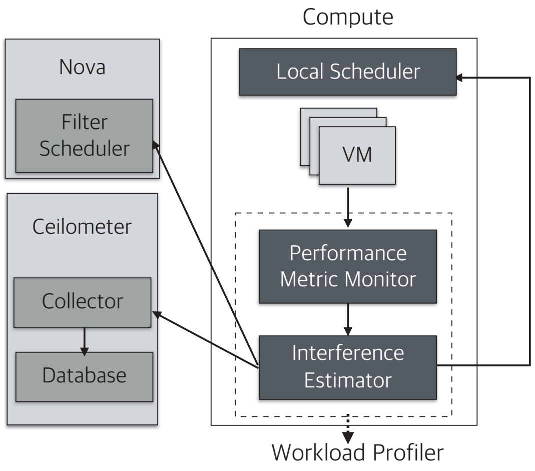


Figure 2.4: Proposed High-Level Workload Profiler Architecture. The workload profiler consists of two components, which are performance metric monitor and interference estimator. Each workload profiler is in each compute, and it delivers the profiled information to other services.

focused on the resource availability. To the current frameworks, ”how much of each resource is available in the datacenters?” is an important question. In the case of OpenStack, the numbers of allocatable virtual cores & network IPs and available RAMs are regarded as important resources. For this reason, the OpenStack compute (Nova) and OpenStack resource monitor (Ceilometer) consider the resource availability first, and Nova distributes the workloads to the hosts as evenly as possible.

VMWare vSphere, another production-level cloud solution, uses DRS (Distributed Resource Scheduler) as a cluster scheduler, and DRS works based on the resource entitlements, which are weighted resource usage based on CPU and memory utilizations [22]. Figure 2.5 depicts the architecture of VMWare vSphere DRS. It dynamically reconfigures the pools of resources, which are allocated to the VMs based on their requirements. A set of resources in a datacenter are represented by a resource pool tree, controlling the resource by the three different policies, which are reservation, limit, and shares. DRS is different from the Filter scheduler in terms of the granularity of controlling the resources, but it is similar to OpenStack in that it evaluates the cluster imbalance by using the standard deviation of host resource entitlements and dynamically schedules the workloads. These resource management strategies have limitations, specifically ”each of them does not care the performance goal such as the SLO.”

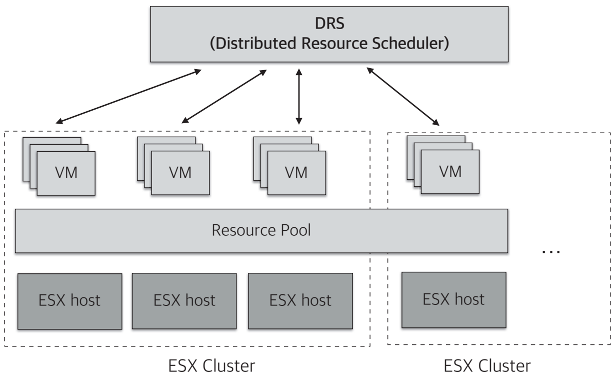


Figure 2.5: VMWare vSphere DRS. DRS collects the stats of VMs and hosts and dynamically schedules the VMs possibly by migrating them to other hosts.

To overcome their limitations, the datacenter frameworks require flexibility and efficiency. The flexibility means that the frameworks should prioritize the workloads based on their performance goals and allocate the resources accordingly. The efficiency means that the frameworks have to maximize the performance of workloads by allocating the resources properly. To gain the flexibility and efficiency, we propose a resource manager, consisting of the local scheduler and global scheduler. These schedulers are different from the existing datacenter schedulers such as the OpenStack Filter scheduler, in that they focus on the performance goals. In the case of the global scheduler, it periodically predicts the changes in latency, and if the predicted value exceeds the predefined threshold, it triggers the performance recovery procedure. In the case of the local scheduler, it classifies the workloads based on their characteristics, and allocates the resources considering their service goals. For example, the local scheduler allows latency-critical workloads to use the dedicated resource pool, but gives other batch workloads to the shared resource pool, for which the latency does not matter. The local scheduler also adaptively reacts to the resource contentions on the host, and thus it can mitigate the performance degradation of workloads and improve the performance isolation for the workloads.

Chapter 3

HIS: Towards Hybrid Isolation for Shared Multicore Systems

3.1 Introduction

A variety of applications from the simple web server to the complicated machine learning are running in the modern data centers. In the data centers, these applications are typically running on the multicore servers, sharing the computing resources such as CPU and memory to improve resource efficiency. Sharing resources on a machine is essential to reduce the total cost of ownership (TCO) of the data center; however, it causes contentions for the shared resources leading to performance degradation [23]. The performance degradation may result in user complaints and tremendous revenue loss [24]. To meet the service level objectives (SLOs) of multiple applications while improving resource efficiency in a machine, it is necessary to enforce isolation techniques appropriately to mitigate resource contentions.

There are two types of isolation techniques for multicore systems, that is, software and hardware ones. Software techniques are isolation techniques that allocate resources such as CPU and memory by controlling interactions among threads and resources in a software manner. They are broadly used in various platforms because it is relatively easy to adopt software isolation techniques [7, 9]. Moreover, software techniques are flexible in terms of performance, allowing multiple configurations for maximizing performance [11, 14]. On the other hand, software techniques are relatively loose isolation than hardware ones since they do not directly segregate or manipulate resources contrary to hardware ones. It makes software isolations less strict and less responsive than hardware ones, which may result in relatively slow isolation enforcement and high-performance variations [17]. Further, compared with hardware isolation, software one may have a larger search space for configurations due to considerable available combinations. For example, hardware cache partitioning provides strict isolation for last-level cache, and per-core dynamic voltage frequency scaling (DVFS) is useful when boosting latency-critical operations [16, 17].

Several research works have utilized software and hardware isolation techniques. First, some works use software techniques, such as core allocation, cycle throttling, and thread placement [7, 8, 11, 14]. Software approaches focus on efficient, portable, and flexible isolation. However, their approach is less strict in terms of providing predictable performance, and less responsive in that latency to isolation may be relatively high. Second, a few works utilize hardware techniques, such as hardware cache partitioning and per-core DVFS [16, 17]. Hardware approaches are strict and fast because they directly control the hardware feature for performance isolation. Their approach allows stable performance for workloads by segregating resources completely or quick response time for rapid changes in workloads. However, the approach may use a few hardware configurations that may not be enough for achieving maximum performance. Third, some research works use both hardware and software techniques for the isolation of multiple resources [25, 26]. Their works are in line with ours in terms of using multiple types of isolation techniques. However, we focus on the tradeoffs in hardware and software techniques, which they have not fully explored, and we effectively perform isolations by profiling the sources of performance degradation among various resources, which they have not focused.

In this chapter, we investigate the characteristics of isolation techniques in terms of strictness, responsiveness, and flexibility. We explore the tradeoffs lying between hardware and software techniques and further evaluate a prototype that combines software and hardware isolation techniques to overcome the shortcomings of each isolation technique. The proposed scheme considers the tradeoffs mainly caused by the isolation mechanism which is either strict and low-latency hardware techniques or loose but flexible software ones to mitigate the contentions dynamically according to the workloads' resource demands and execution patterns.

To realize the hybrid isolation scheme, we developed a profiler and a user-level scheduler that uses four isolation techniques. Profiler conducts online profiling stages that collects performance counter samples during no contention for estimating resource contentions. It is useful to track the changes in contentions and identify which resource is the most contentious. Scheduler uses two hardware isolations, which are hardware cache partitioning and per-core DVFS, and two software isolations that allocate cores and perform thread placement. Using these techniques, the scheduler can perform isolation strictly, fast, and flexibly to consolidated workloads. We have evaluated our scheme for latency-critical workloads with different levels of SLOs when colocating best-effort workloads having diverse characteristics in terms of resource contentions. Our results show that the proposed scheme can improve the performance of foreground workloads by from $1.4-76\times$ compared with Heracles [25], the recent performance isolation frameworks.

The contributions of our work as follows:

- We have explored the tradeoffs between hardware isolation techniques and software ones in terms of the strictness, responsiveness, and flexibility.

- We have designed and implemented a hybrid isolation system that profiles the sources of contentions in an online manner and adaptively isolates workloads considering the characteristics of workloads and tradeoffs in the isolation techniques.

- We have evaluated that our system can effectively improve the performance compared with the state-of-the-art isolation scheme for the latency-critical workloads with diverse SLOs.

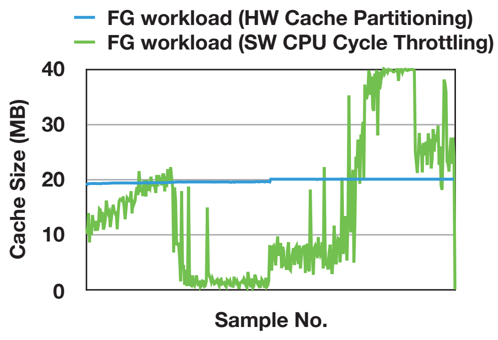
The rest of this chapter is organized as follows: Section 3.2 describes the tradeoff between hardware and software techniques, and Section 3.3 shows problem of ineffective isolations; different isolation effects and inefficient isolation decisions. Section 3.4 describes the design and implementation of our hybrid isolation scheme. Section 3.5 shows the evaluation results. Section 3.6 covers the related work. Finally, Section 3.7 concludes this chapter.

3.2 Trade-offs between Hardware and Software Techniques

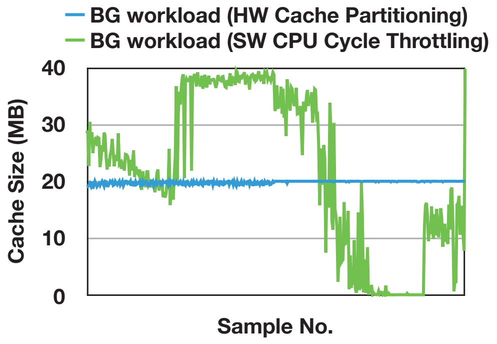
In this section, we describe the trade-offs between hardware and software isolation techniques. Also, we present the effects of isolation techniques by the characteristics of workloads such as resource demands.

To describe the trade-offs, we ran two workloads, each is a multi-threaded process and ran on a single socket while enforcing performance isolation. The test machine has 32GB of RAM, and its CPU is a Xeon E5-2683v4 (2.1GHz, 16-cores). We turned off the hyper-threading feature. For baseline, we used static software isolation (i.e., *Core Allocation*). We used `cgroups cpuset` [6] to allocate 8 cores (16 cores) of one socket equally to each workload and allocate local memory. We chose several benchmarks for foreground workloads that show a diverse range of memory and LLC access pattern; `streamcluster` and `canneal` of PARSEC [27], and `kmeans` and `nn` of Rodinia [28], and Apache benchmark (`ab`). For background workloads, we used `SP` of the NASA parallel benchmark [29] because it shows high LLC and memory bandwidth usage enough to stress memory subsystem.

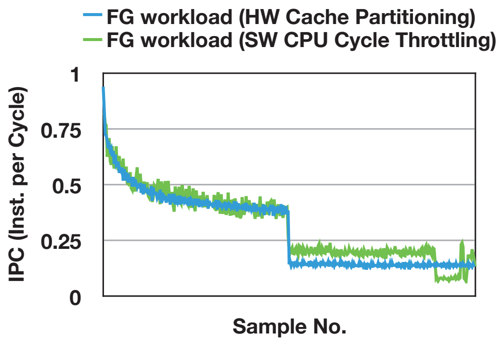
Strictness. To show the strictness of hardware techniques and software ones, we compared *hardware cache-partitioning* and *software cycle-limiting* by running two workloads concurrently on a socket. We ran `canneal` as a foreground and `SP` as a background by allocating the equal number of dedicated cores. For hardware isolation, we allocated the equal amount of LLC to each workload, and for software isolation, we limited the CPU cycles of a background workload to use only 50% of assigned CPU cycles to restrict LLC accesses to its half.



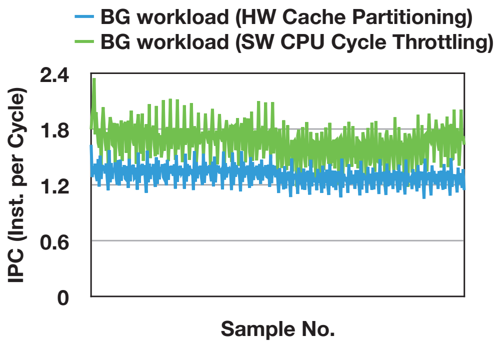
(a) Changes in FG's LLC



(b) Changes in BG's LLC



(c) Changes in FG's IPC



(d) Changes in BG's IPC

Figure 3.1: Comparing the strictness of the hardware and software isolations, necessary for predictable performance, the software one shows high variation of performance. Workloads are `canneal` (foreground) and `SP` (background). *x-axis* represents the number of samples and *y-axis* represents LLC allocation and IPC of workloads.

Figure 3.1 shows the changes in LLC usage and instructions per cycle (IPC) of foreground and background workloads. As shown in Figure 3.1a and 3.1b, when using the hardware isolation technique, the LLC allocations are equally divided all the time due to the direct and strict segregation of hardware isolation. On the other hand, when using the software isolation technique, the LLC allocations are changed dramatically over times, because the software isolation does not guarantee the physical segregation of resources.

The difference between isolation techniques makes the performance of workloads unpredictable. Figure 3.1c and 3.1d show the performance variations of the software isolation. Software CPU cycle limiting shows a larger variation compared with the hardware cache partitioning in the case of foreground workload. Even worse, in the case of background workload, those variations are getting much bigger, showing more unpredictable IPCs when software cycle limiting. As a result, we find that the hardware isolation technique provides better predictable performance than the software isolation technique by enforcing strict isolation.

Responsiveness. We also compared responsiveness of the hardware and software technique to find which technique can provide more fine-grained contention control by performing isolation quickly. We define responsiveness of isolation technique as the latency to its effect. We chose per-core DVFS as a hardware isolation technique and CPU cycle limiting as a software one to demonstrate the difference in terms of the responsiveness. Per-core DVFS can adjust the core frequencies at 0.1GHz granularity. On the other hand, CPU cycle limiting can change the cycle at 1% granularity. Even though the control granularity of software is more fine-grained, the speed of enforcing isolation is faster when enforcing hardware isolation. Enforcing core frequency takes a couple of milliseconds. Meanwhile, enforcement of cycle limiting takes 40-50 milliseconds which is 13-25$\times$ longer than DVFS as shown in Table 2.1.

To illustrate the responsiveness of the hardware and software isolation technique, we ran two workloads in a socket, and each runs on the eight dedicated cores; one is apache web server and the other is `SP` that shows high LLC and memory bandwidth demands. We evaluated the responsiveness of isolation techniques by running the apache benchmark (`ab`) which sends the requests to the web server. While running two workloads, we increased the request load of the web server, and also throttled the

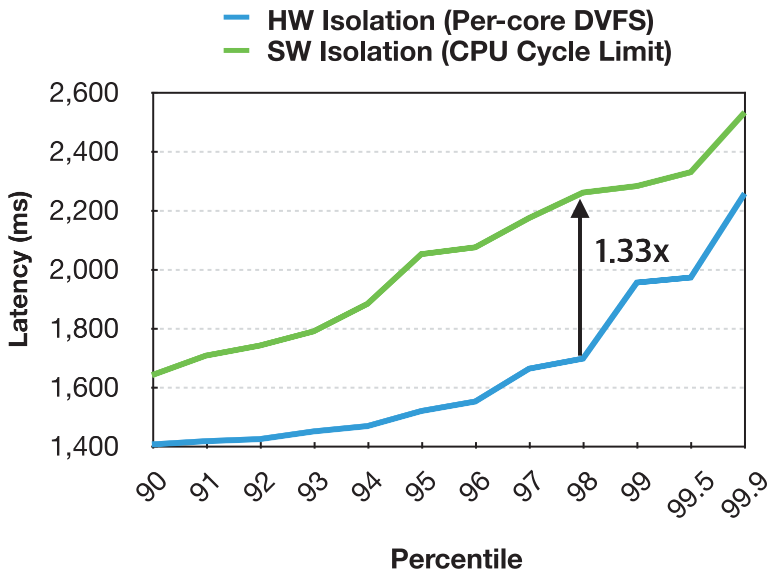


Figure 3.2: Comparing the responsiveness of the hardware and software isolations. The graph shows the responsiveness can affect the performance. Workloads are apache web server (`ab`) (foreground) and `SP` (background). *x-axis* represents the percentile of web server request and *y-axis* represents the latency of web server

execution of the background workload. To compare hardware and software techniques, we conducted the experiments twice; first with per-core DVFS, and second with CPU cycle limiting. In both experiments, we throttled the CPU cores of the background workload by increasing the degree of isolation by a step at every 200 milliseconds, and we increased ten steps. For per-core DVFS, we changed the CPU frequency of the background workload from 2.1GHz to 1.2GHz by 0.1GHz. In the same way, for CPU cycle limiting, we also changed the allowed CPU cycle percentage from 100% to 57%, which is the same degree as DVFS. Figure 3.2 presents how the hardware isolation technique responds more quickly. When performing the software isolation, 98th percentile latency can be 1.33$\times$ higher than hardware isolation. This latency difference in tail-latency comes from fast isolation speed thanks to low overhead of hardware technique. The effect of fast isolation may be more important where the resource contention changes frequently or fine-grained control matters. Consequently, we find that the hardware isolation technique is more responsive than software one.

Flexibility. We investigated the flexibility of the hardware and software isolation technique. Flexibility means the ability to choose better scheduling options by mapping threads to resources (e.g., CPU cores and memory nodes) or grouping workloads which minimize the contentions and improve the throughput for the workloads. To describe the effectiveness of flexibility, we grouped four workloads, which shows high LLC-intensity or memory bandwidth intensity, into two groups. And, two workloads are paired in each group, and scheduled each group to the separate sockets. We performed different isolations to the same four workloads; the first with the hardware cache partitioning and the second with scheduling by regrouping the background workloads. Figure 3.3 shows scheduling is more effective than hardware cache partitioning, so that the performance of `canneal` and `SP` improves by up to 1.6$\times$ and 1.3$\times$ than the hardware one. Some workloads show performance degradations, but their performance loss is reasonable considering the other workloads' performance benefit. The results indicate that software isolation can be useful when the resource contention can not be reduced by the hardware isolation, which have a few isolation options. In this experiment, hardware cache partitioning can only solve resource contention in a socket. However, software isolations such as migration enable more options for enhancing performance and improving resource efficiency.

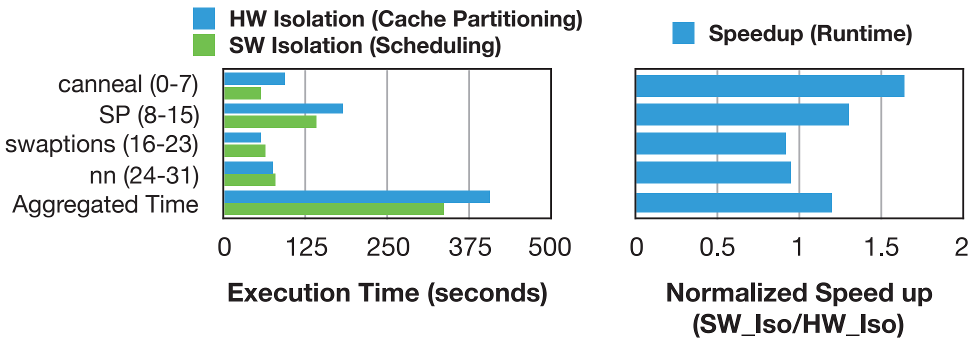


Figure 3.3: Benefits of the flexible software isolation. `canneal` and `SP` show high LLC and memory contentions. However, `swaptions` and `nn` are relatively not. In case of performing the hardware isolation, the contention is still high. On the other hand, the software isolation can effectively mitigate the contention significantly. *x-axis* shows the runtime and speedup of workloads and *y-axis* shows their name and CPU affinities. The ranges in parenthesis indicate the range of CPU IDs where workloads run.

3.3 Ineffective Isolations

3.3.1 Different Isolation Effects

In addition to the trade-offs between isolation techniques, the isolation effects depend on the characteristics of workloads such as resource demands. The same isolation technique can deliver different impacts according to the workloads. We present a simple example of multiple isolations are performed on the different foreground under the high LLC and memory contention.

To demonstrate the effectiveness of each isolation, we tested all isolation techniques which are `Partitioning`, `Throttling`, and `Scheduling`. We manually divided LLCs evenly to workloads using Intel CAT for `Partitioning`. We

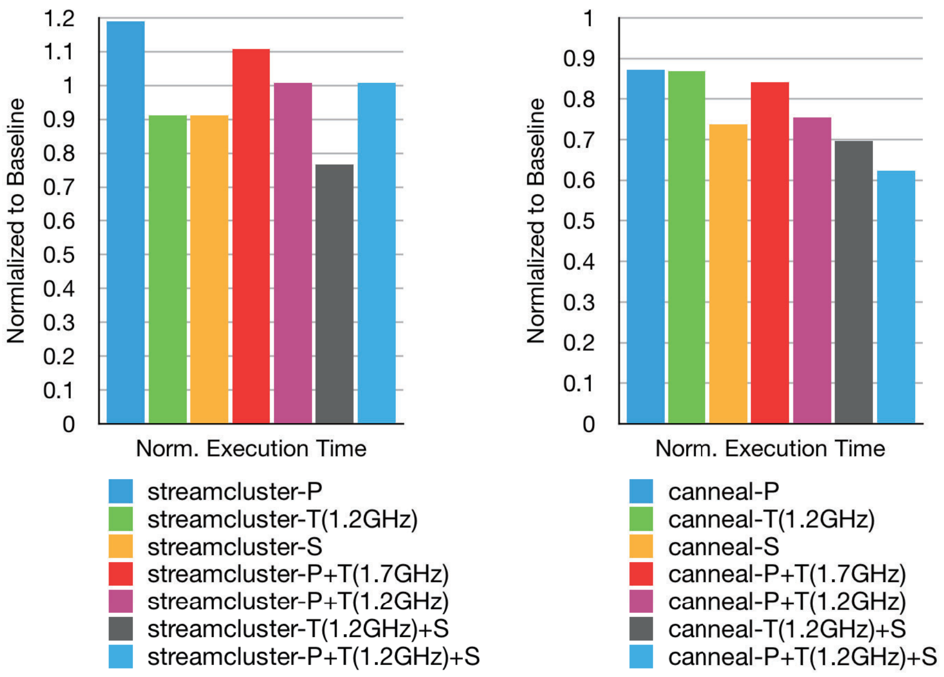


Figure 3.4: Enforcing multiple isolations to `streamcluster` and `canneal`, each colocated with `SP`. The execution time is normalized to the performance of a workload running on the dedicated cores on the default system (P: `Partitioning`, T: `Throttling`, and S: `Scheduling`).

also changed the execution rate of background workload by setting the frequency of core as the highest frequency(2.1 GHz), the middle frequency(1.7 GHz), and the lowest frequency(1.2 GHz) for `Throttling`. Finally, we changed the number of cores of the background workload to the half of allocated cores, which is four cores, to describe the effect of mitigating memory contention via `Scheduling`. The baseline is the case of when two workloads are running on its dedicated cores without performing any isolation.

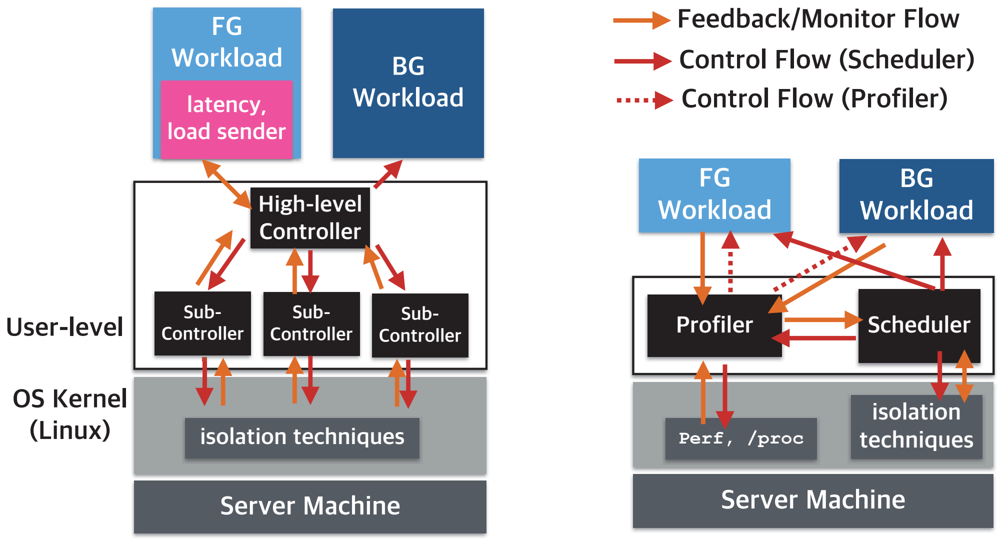
As shown in Figure 3.4, the performances of foregrounds vary according to the different isolation techniques. This is because the resource demands of the foregrounds are different from each other, and also isolation effects are different depending on the isolation techniques as well. In the case of `streamcluster`, partitioning LLC increased the execution time by 20% compared with the baseline, but `Throttling` or `Scheduling` reduced the execution time by 10% compared to the baseline. This is because the `streamcluster` is a memory bandwidth intensive workload, so restricting LLC makes its performance worsen. However, `Throttling` or `Scheduling` could increase the memory bandwidth of `streamcluster` by reducing background's memory access. In the case of `canneal`, it uses less memory bandwidth than `streamcluster`, but it is an LLC intensive workload with high LLC hit ratio. For `canneal`, all three isolations can reduce the execution time significantly.

However, in the case of `streamcluster`, when both techniques (`Throttling` and `Scheduling`) are used, the execution time is reduced by 24% compared with the baseline configuration. On the other hand, in case of `canneal`, the execution time is reduced by up to 38%, which is the highest performance improvement. In this way, we find that effective isolation techniques can be different according to the characteristics of the workload. Moreover, we realize that it is necessary and important to enforce appropriate isolation techniques adaptively considering the changed contentions.

3.3.2 Inefficient Isolation Decisions

Performing the "right" isolation technique is critical to the performance isolation. The quality of isolation decision can be determined by "which isolation is used" and whether "the chosen isolation technique can mitigate the most significant contention effectively". However, current recent isolation frameworks perform isolations based on 1) the offline analysis of contentions for workloads or 2) the online performance feedback such as tail latency and throughput. The former is regarded as expensive approach in that it requires offline analysis to accurately mitigate resource contentions. Moreover, it may need another offline analysis for every machine platform and workloads to get workload profile. On the other hand, the latter does not have to require offline data, instead it directly gets performance feedbacks periodically to decide isolations. However, when deciding isolations, it does not consider which isolation technique is the most contentious one. This can affect to the quality of isolation decisions when multiple isolation techniques are utilized to reduce performance interferences. There may be possible to allocate resources or set configurations sub-optimally without considering the most contentious one or coordinating isolation techniques.

Table 3.1 and figure 3.5 shows the comparison between recent isolation system (i.e., Heracles [25]) and our proposed system. Heracles is well-known performance isolation system that utilizes multiple isolation techniques to mitigate resource contention among a latency-critical workload and best-effort workloads while improving resource efficiency. It consists of multiple sub-controllers and one high-level controller. A high-level controller is responsible for monitoring tail-latency of a latency-critical workload and deciding whether the best-effort workloads can be allocated more resource allocations or not by the application-level performance data, which are the tail-latency and loads (i.e., 99th percentile latency and queries per second), every fifteen seconds. To obtain performance feedbacks from latency-critical workloads, applications need to be modified to send their end-to-end latency data to Heracles. Sub-controllers are responsible for different resource contentions including core & memory, power, and network bandwidth. These sub-controllers simultaneously perform isolations every two seconds according to the decision for resource allocation given by a high-level controller. Unlike Heracles, our scheme does not need application modification for isolation decision. We directly measure which resource is how much contentious and perform isolations for the most contentious one at the time. Although our scheme does not consider simultaneous isolations enforcement, we consider the most contentious isolation at a time, thereby we can avoid the sub-optimal isolation decision that worsen application's performance. Moreover, isolation interval is relatively short that is able to quickly adapt to the changes in dynamic resource contention compared with Heracles.



(a) Heracles Architecture

(b) HIS Architecture

Figure 3.5: Comparing the system architecture of Heracles and ours

Table 3.1: Comparison between Heracles and the proposed scheme

	Heracles [25]	**HIS (Proposed)**
Need for App. Modification	**O**	**X**
Contention Analysis	**X**	**O**
Simultaneous Isolation Enforcement	**O**	**X**
Isolation Decision	by application's **latency** and **loads**	by application's **contentions**
Isolation Interval	High-level Controller: 15s Sub-Controller: 1∼2s	100ms

3.4 HIS: Hybrid Isolation System

This section describes the overview of how our proposed system can deal with the trade-offs between multiple isolation techniques depending on the characteristics of workloads. To achieve this goal, we propose *HIS*, a hybrid isolation system that leverages hardware and software isolation techniques to mitigate contentions and improve the performance of workloads.

Figure 3.6 illustrates our *HIS* architecture. As described in the figure, our system consists of a profiler, isolation techniques, and a scheduler. We divide workloads as foreground workloads and background workloads. The foreground is a latency-critical or high-priority batch workload and the background is the best-effort workload. *HIS* groups these two types of workloads and performs isolations on *workload group*s and places a group on a socket to improve resource efficiency. We also assume there is one foreground workload in the workload group like other clouds do [25].

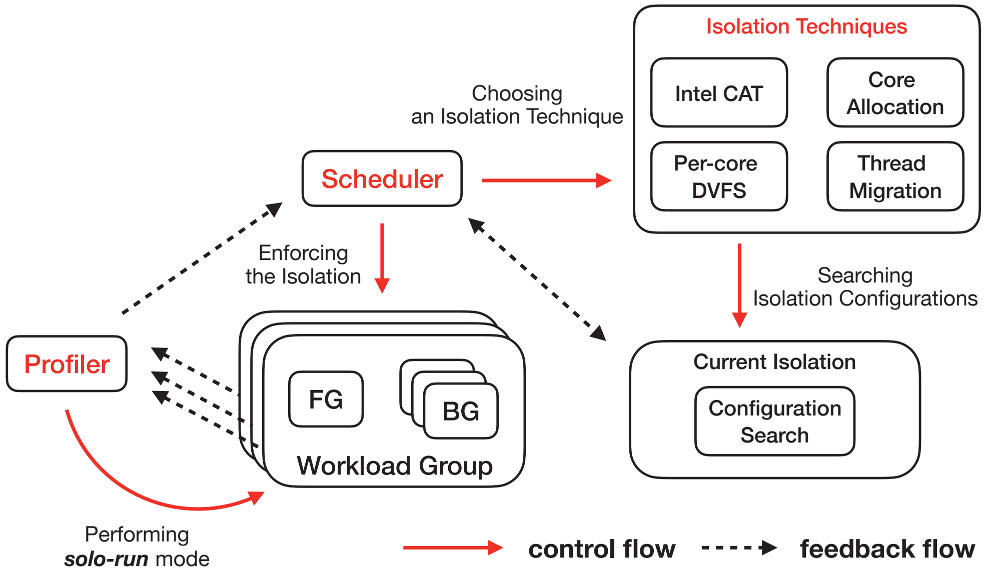


Figure 3.6: *HIS* architecture. It consists of a *profiler*, *isolation techniques*, and a *scheduler*. The redline shows control flow and black dotted line shows the feedback of workload profiles, performances, and isolation decisions. The scheduler uses four isolation techniques; two hardware isolations (i.e., Intel CAT and per-core DVFS) and two software isolations (i.e., core allocation and thread migration).

The profiler collects the performance counters from workloads, and then profiles resource contentions from the collected counters. To profile resource contention online, the profiler performs *solo-run mode*, which enables for a foreground workload to run alone, to obtain the performance counters of each workload when no contention exists. After profiling *solo-run* data, the profiler collects performance counters of consolidated workloads to estimate how contentions affect resource usages. We define the performance counters of workloads when workloads co-executes as *co-run* data. Note that, we currently consider the *solo-run* data for the foreground workloads. We will describe more detail of *solo-run mode* at the Section 5.1.Both the obtained *solo-run* data and the *co-run* data are used to calculate the resource contention. After that, it sends the information of resource contention to the scheduler. Then, the scheduler checks which resource contention is the most contentious and decides an isolation technique considering the types of isolation technique and resource contention. Once an isolation technique is selected, the scheduler searches for a configuration of isolation and enforces isolations until the contention is minimized to below the pre-tuned threshold (i.e., 5% for each contention). In other words, the scheduler adjusts isolations to reduce the resource contention for a foreground workload close to when the workload runs alone. The scheduler repeats this procedure until the foreground workload finishes.

For isolations, *HIS* checks which isolation technique is the most appropriate one among multiple isolation ones; *HIS* considers multiple hardware and software isolations, and applies isolation techniques incrementally to improve the performance of a foreground workload while maximizing that of background one. This approach is useful because the scheduler reflects the subsequent resource contention and can enforce the corresponding isolation technique. For enforcing a proper isolation, the scheduler should know the dominant contention and decide appropriate isolations. Following sections will describe how the profiler profiles contention and how scheduler chooses isolation configurations in detail.

3.4.1 Profiling Contention

Profiling contention is essential for performance isolation. Our scheduler receives the resource usages of workloads from the profiler to estimate the contention on the system. To profile the contention, the profiler measures the per-workload performance counters such as LLC misses and LLC references in every profile interval (i.e., 100ms). It calculates the resource contention by the difference of resource usages between when all workloads run concurrently (*co-run*) and when a workload runs alone (*solo-run*).

We used the differences of *co-run* and *solo-run* data, because it presents resource sensitivity of a workload that how much the performance of workloads is degraded by the contention compared with no contention exists [12, 17, 30].

The profiler maintains the *solo-run* data for each foreground workload to calculate the resource contention, thus the scheduler checks whether the *solo-run* data exists or the execution phase has changed at every scheduling interval using already sampled data. If there is no data to calculate resource contention or the profile sample data is outdated, then the scheduler dictates to collect the new samples for *solo-run* data by stopping other background workloads. We call this procedure *solo-run mode*. We used two signals to enable *solo-run mode*; `SIGSTOP` for stop running workloads and `SIGCONT` for resume stopped workloads. To enable the *solo-run mode*, the profiler stops all current isolations and also pauses other background workloads during the successive profile intervals (e.g., one or two seconds). During the *solo-run mode*, only a foreground workload runs alone, and after finishing, the profiler stores all collected performance counters during the mode and resumes all previously paused isolations and background workloads.

The profiler classifies the workloads by their mostly used resources and also classify them by the type of the workload provided by users (e.g., FG and BG). We focused on the LLC and memory bandwidth to mitigate the contention on the memory subsystem. To measure the LLC contention, we used the `LLC misses` and `LLC references`, obtained by performance counters, and calculate the LLC hit ratio reflecting how much workload reuses the LLC. In addition, `local_mem_bytes`, obtained by Intel *Resctrl*, is used to estimate the memory bandwidth contention. The metrics can be added to consider more contentions and complicated execution patterns. With these metrics, the profiler can determine the dominant resource by comparing them, and also classify a workload as one of which CPU-intensive, LLC-intensive, or memory bandwidth intensive at every scheduling interval.

3.4.2 Hybrid Isolation

In this section, we will detail the trade-offs of isolation techniques and describe how our scheduler leverages them to mitigate the contention.

Isolation Mechanisms

HIS considers four isolations to mitigate contentions. In Table 2.1 of Section 2.1.1, HIS uses four techniques, which are hardware cache partitioning, per-core DVFS, core allocation, and thread migration, in hybrid isolation system.

Hardware Isolations. For hardware isolations, we used the Intel Cache Allocation Technology (Intel CAT) and per-core dynamic voltage frequency scaling (DVFS). With Intel CAT, *HIS* can allocate a LLC by the unit of a way. In our machine (i.e., Xeon E5-2683v4, 16-cores per socket), a socket has 40MB of LLCs and each consists of 20 ways. Intel CAT provides strict isolation for the LLC in a socket, because it partitions LLCs physically by masking the ways in *Resctrl*. We also used the per-core DVFS to throttle the execution of workload. Per-core DVFS is used to improve power efficiency of processors as well as mitigate the contentions and enable fine-grained control to improve performance of workloads. Using per-core DVFS, the scheduler can rapidly mitigate the memory contention, generated from contentious background workloads by adjusting the frequencies of cores running backgrounds. For enforcing core frequencies, we used the `CPUFreq Governor` of Linux.

The hardware isolations perform strict and quick isolation compared with the software isolations. Hardware cache partitioning provides the strict isolation which affects more predictable performance for the workloads. They generally take few milliseconds to reflect their effects to the workloads' performance. As shown in Table 2.1 (in Sec. 2.1.1), we observed $2-3$ms of latencies, and this low latency is beneficial to meet the SLOs of the latency-sensitive workload when the execution patterns of workloads changed frequently or the load of latency-sensitive workload shows high variation.

Software Isolations. For software isolations, we used the `cgroups cpuset` to allocate CPUs and memory nodes to workloads. To mitigate the contentions on the multicore systems, two software isolations are used in scheduling; core allocation and thread migration. Core allocation performs the allocation of CPU cores for workloads to isolate core resources by their CPU demands. For example, latency-sensitive workloads such as the web server can show high load variation by the user patterns, so the CPU demands can vary by their loads. Therefore, core allocation should be performed according to the CPU demands to improve resource efficiency and meet the SLO of foreground workloads.

Unlike core allocation which manages the contention of a workload group, thread migration detects the performance imbalance between workload groups, then it regroups those workloads by migrating workloads to the other socket. The thread migration is effective when the contention on a workload group is too large to be mitigated by other isolations (e.g., hardware isolations) on the single socket. However, too frequent thread migrations may be harmful to the performance because the cost of the memory migration over the sockets is expensive [31]. Therefore, we designed that thread migration is triggered only (1) if the performance benefit is estimated to exceed the threshold or (2) if the phase changes in a workload group is detected.

The software isolations provide flexibility compared with the hardware isolations. Core allocation treats CPU demands as well as mitigates memory contention according to the type of contention of workloads. They typically take more times than the hardware isolations to reflect isolation impacts on the workloads' performance (e.g., tens to hundreds of milliseconds).

Hybrid Scheduler

The hybrid scheduler periodically (1) chooses a proper isolation technique and (2) searches isolation configurations to improve the performance of foreground workloads within the workload groups. Before the hybrid scheduler initiates isolations, the profiler sends the information about current active workload groups, such as pid, workload type (FG or BG), and profiled resource contention to the scheduler. By using the workload group information, the scheduler initiates the isolations for the workload groups in parallel. While performing isolations, the scheduler checks whether the workloads in the group need *solo-run* data to calculate contentions, and if yes, requests for the profiler to perform *solo-run mode* to collect the new *solo-run* data.

Choosing an isolation technique. The hybrid scheduler chooses an isolation technique based on the mostly contentious resource, identified by the profiler. For the resource contention, at first, the scheduler checks whether the hardware isolation is available for the resource or not, and chooses the isolation if the isolation is possible and has not been tried. Between software and hardware isolations, the scheduler prioritizes hardware isolations for the strict and fast isolation. If all hardware isolation has tried before, the scheduler checks whether the software isolation technique are available for the resource or not and if it is possible then chooses the software isolation. If all the hardware and software isolations are used, the scheduler reconsider all techniques to reuse them. We implemented our policy to consider hardware isolation as much as possible. However, the policy for choosing an isolation technique can be changed to meet SLOs of the workloads.

There are two cases that the software isolations are chosen rather than hardware one. The first case is wrong invocation for an isolation technique. The scheduler often fails to search a better configuration due to the a few errors of profile data. For instance, the profiler may identify CPU contention as major factor when the actual contention is

LLC contention. In this situation, the scheduler may perform hardware cache partitioning by its profile results. To minimize this case, we may choose isolation techniques more conservatively by not changing techniques until successive contentions are detected. The second case is when the scheduler exploits all hardware techniques, but still fails to reduce the contention because of their lower number of available configurations. For example, while the per-core DVFS may be not enough for mitigating severe memory contention due to its small configuration ranges, restricting the number of cores may be more beneficial to mitigate memory contention.

Enforcing isolation. After choosing the isolation, the scheduler searches isolation configurations that minimize the resource contention by enforcing the various configurations repeatedly and incrementally. Whenever before enforcing isolation, the scheduler decides whether it allocates more resources to the foreground workload or not, based on resource contention. For example, if the dominant resource contention for the foreground workload is LLC contention, and also if the LLC hit ratio of the foreground one during *co-run* is lower than that of *solo-run*, the scheduler allocates more LLC ways to foreground workload. Because lower LLC hit ratio than the *solo-run* typically means that foreground workload can be improved if the workload is assigned more LLC ways.

Once the isolation is performed, the scheduler waits until the effect of enforcing an isolation is reflected, and then it repeatedly checks the degree of the contention. We empirically find that 100ms is the most effective time to feedback contentions, yet the wait time can be tuned depending the target workloads. The scheduler finds there is no severe contention, or it can not perform the isolations further (e.g., searching all possible configurations), then the configuration search ends. Finally, the scheduler enforces the configuration for chosen isolation.

3.5 Evaluation

This section describes the preliminary experimental setup and results. We evaluated the hybrid isolation system for the batch and latency-sensitive workloads compared with the default Linux system using static software isolations. Here, we define the baseline as the case of *co-run* where the foreground and the background runs together on a socket. Both workloads share memory subsystem such as an LLC and a memory controller, but have their own dedicated CPU cores.

3.5.1 Experimental Setup

We evaluated the *HIS* on a dual 16-core Intel Xeon E5-2683 v4 server. The LLC size of the server processor is 40MB and can be allocated to the workload in 2MB units (per a way) using Intel CAT. The nominal frequency is 2.1GHz and the configurable core frequencies are 10 steps from 1.2GHz to 2.1GHz. We turned off Turbo-boost and Hyper-threading. Our test machine is equipped 32GB of RAM with each socket. The maximum bandwidth of the socket is measured to 68 GB/s by Intel *VTune* and we used Linux kernel 4.19.0.

We used various benchmark applications from four different suites. For batch foregrounds, we used PARSEC (`bodytrack`, `canneal`, `streamcluster`, `dedup`, `facesim`, `ferret`, `fluidanimate`, `swaptions`, and `vips`) and Rodinia (`cfd`, `nn`, `kmeans`, and `bfs`). For latency-sensitive foregrounds, we used the apache web server and `ab` (apache benchmark), three workloads (xapian, moses, and sphinx) in Tailbench [32]. In the case of latency-sensitive foreground, the scheduler should respond quickly to deal with the load spikes of the web server. We chose the `SP` from NPB as the background for comparing static naive approach and ours, because SP shows high memory bandwidth and LLC usage than other benchmarks. For comparison with Heracles, the-state-of-art isolation system, we chose three different levels of background workloads in terms of memory intensiveness; for CPU workloads, `bfs` and `swaptions`, for LLC-intensive workloads, `canneal` and `facesim`, for memory bandwidth-intensive workloads, `SP` and `UA`.

3.5.2 Experimental Results

Batch Workloads

We show the performance results for the batch workloads running as the foreground in Figure 3.7. In the figure, *HIS* isolates foreground workload effectively, so that the performance of batch workloads are improved significantly compared to the *co-run*. In case of *canneal*, the performance is improved more than 1.7× than *co-run* with simple core isolation that the workloads run on their dedicated cores, and the scheduler improves the performance of benchmarks on average 1.22× than *co-run*. On the other hand, the performance of the background workload is degraded, because our scheduler restricts the resource usage of the background workload to improve the performance and the responsiveness of foreground.

Latency-sensitive Workloads

Figure 3.8 presents the performance of latency-sensitive workload running as the foreground. In order to evaluate the performance of latency-sensitive workload, we modified the `ab` which uses the *Pareto* distribution to reproduce situations where a few users are connected during most of the time and the connections are bursty. We measured the percentile latencies of requests.

In the figure, *HIS* can reduce the tail-latencies of web server below the performance of *solo-run* (8 cores) until 99.9th percentile, because the scheduler considers changes in dynamic load of the web server as well as the dominant resource contentions, and enforces various isolation techniques according to them. We also plot the tail-latencies

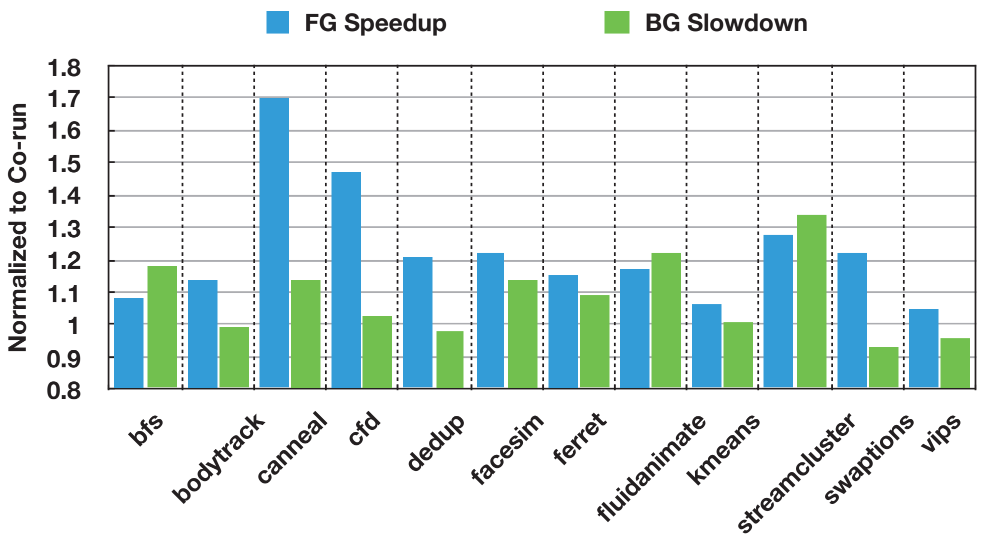


Figure 3.7: Performance improvement of the batch foreground workload with *HIS* compared to the *co-run*. Each workload is initially allocated eight dedicated cores (background workload: `SP`).

of *solo-run* (12 cores) to compare with the proactive approach that reserves CPU cores as much as the maximum CPU cores that *HIS* allocates under the experiment. The latencies of *HIS* are higher than *solo-run* (12-cores), because *HIS* begins by allocating fewer cores to workload and increase the number of cores assigned to the workloads.

Compared with the *co-run*, *HIS* achieves the performance up to 2.14× speedup (for 99.9th percentile latency), while the performance of background workloads is slow down by 1.47×. We observed that the main reason for the performance improvement of foreground is due to fast and strict hardware isolation, core isolation which allocates more cores depending on the CPU demands, and adaptive isolations.

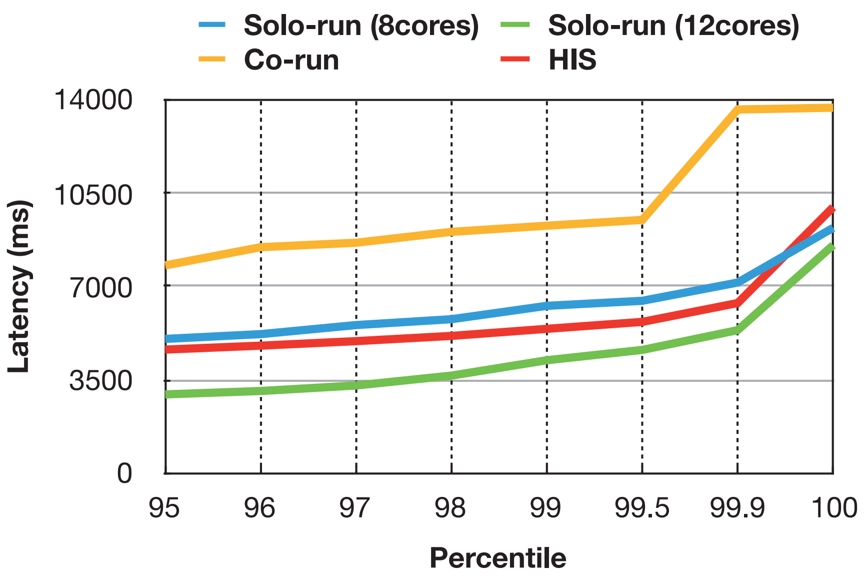
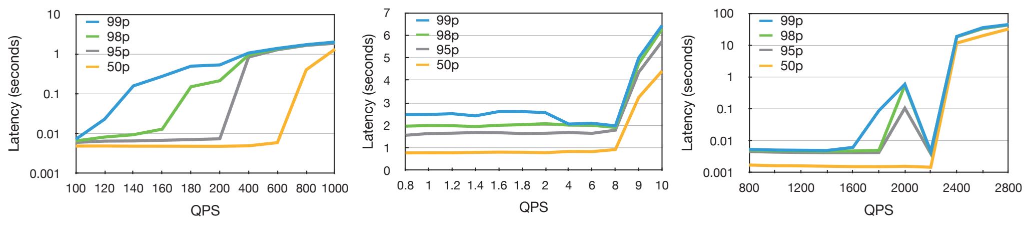


Figure 3.8: Performance improvement of the latency-sensitive foreground workload (Apache web server) with *HIS* compared to the *co-run*. Each workload is initially allocated eight dedicated cores (background workload: `SP`).

Comparison with the recent work

We compared ours with the recent isolation scheme, Heracles, using several workloads in Tailbench [32] to show the performance improvement under dynamic contentions. We chose three latency-critical workloads (`moses`, `sphinx`, and `xapian`) from Tailbench that can show various levels of SLOs from seconds to milliseconds. `moses` is a latency-critical statistical machine translation workload that translates one language to another one. `sphinx` is a latency-critical speech recognition workload, processing natural languages, which is very widely used in AI speakers. `xapian` is a latency-critical online search workload that requires very low latency to return the output of users' search. To identify the characteristics of benchmarks for experiments, we conducted experiments under no contention by increasing the levels of loads in terms of



(a) moses (Peak QPS: 200) (b) sphinx (Peak QPS: 8) (c) xapian (Peak QPS: 2200)

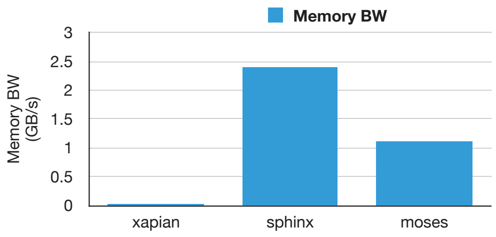
Figure 3.9: The performance profile of latency-critical workloads

QPS. We followed evaluation setup as described in Tailbench. Figure 3.9 shows the performance profile of latency-critical workloads.

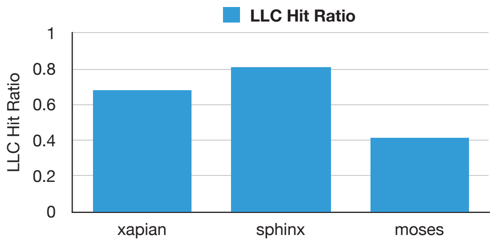
As shown in the figure, we found that the loads that abruptly increase 95th percentile tail-latency. Using these loads, we defined *peak loads* and also set the *high load configurations* of these workloads as 70% of *peak loads*. Each high load configurations is as follows; For the configuration of `xapian` is 1540 QPS, for `moses`, 140 QPS, for `sphinx`, 5.6 QPS. For 99th percentile tail-latency at the high load configurations, `xapian` takes under 10ms, `moses` takes from 10ms to 1s, and `sphinx` takes more than 2s.

Further, we analyzed the characteristics of selected workloads for evaluation. Figure 3.10 shows the various resource usages of selected workloads. As shown in figure 3.10a, we can easily figure out that `sphinx` is the most memory bandwidth-intensive one, `moses` shows moderate bandwidth usage, and `xapian` barely uses memory bandwidth. However, as the figure 3.10b and 3.10c show, `xapian` shows higher last-level cache hit ratio and IPCs than `moses`.

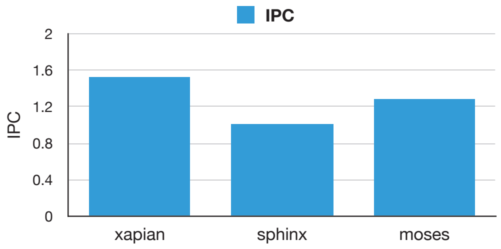
With these workload configurations, we conducted experiments that show how our proposed scheme performs well compared with Heracles. Note that, scheduling interval of HIS is 100ms. Therefore, for fair comparison with Heracles, we adjusted the scheduling interval of all controllers (both high-level controller and sub-controllers) in



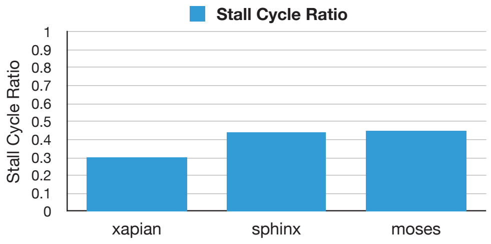
(a) Memory Bandwidth



(b) Last-Level Cache Hit Ratio



(c) Instructions Per Cycle

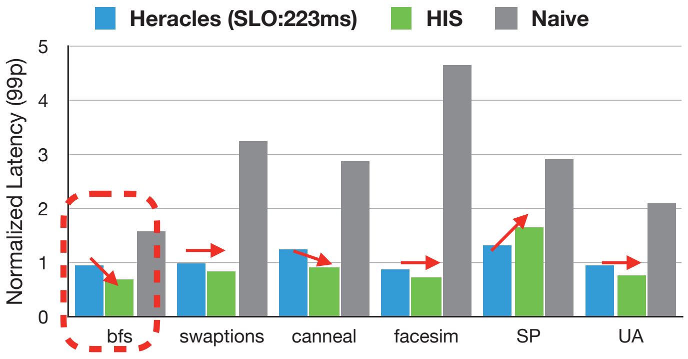


(d) Stall Cycle Ratio)

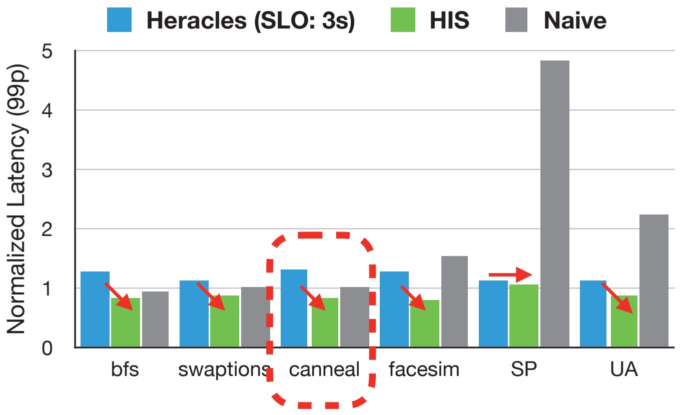
Figure 3.10: The characteristics of latency-critical workloads

Heracles as one second. In addition, we implemented three sub-controllers for Heracles, focusing on memory contentions; sub-controllers for Cores, LLC, and CPU frequencies. All sub-controllers measure current memory bandwidth before performing isolation based on collected performance counters (i.e., `local_memory_bytes` from *Intel Resctrl* and number of CPUs, CPU frequency) to decide allocate more resources or not. On the other hand, *HIS* collects `local_memory_bytes`, `LLC_misses`, `LLC_references`, `instructions`, `cycles`, number of CPUs, `number_of_threads`, and CPU frequency.

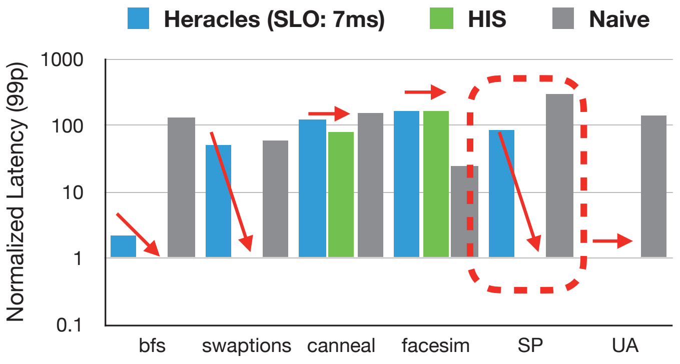
We also set the SLOs of each workload as 1.5× the 99th percentile tail-latency. (i.e., `xapina`: 7ms, `mesos`: 223ms, `sphinx`: 3s). Figure 3.11 shows the evaluation results of Heracles, a naive isolation, and ours. Naive isolation is naive resource allocation that statically partitions CPU cores evenly and does not manage the memory contention.



(a) moses (QPS: 140)



(b) sphinx (QPS: 5.6)



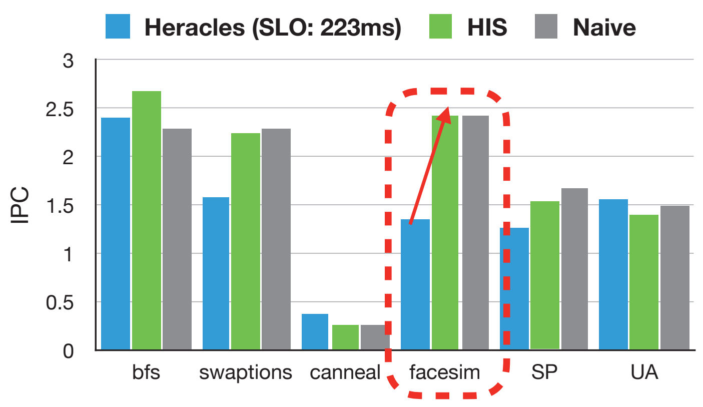
(c) xapian (QPS: 1540)

Figure 3.11: Comparing the performance of latency-critical workloads

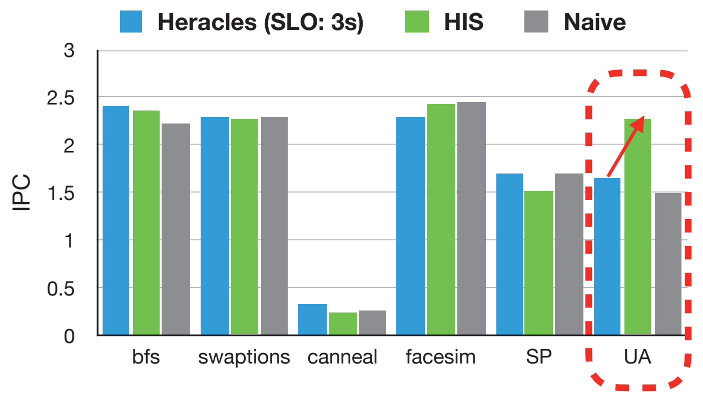
As shown in the figure, although Heracles significantly improve the tail-latency than naive approach, *HIS* shows similar or latency improvement compared to Heracles in terms of 99th percentile tail-latency. Especially a red-dotted circle in each graph shows the most improved case for each workload; `xapian` improves 76×, `moses` improves 1.4×, and `sphinx` improves 1.6× than Heracles. Even though Heracles uses conservative resource allocation policy for best-effort workloads that use latency slack and loads, it is not enough for improving performance. These improvement can be achieved by their unawareness of the most contentious resources that leads to the ineffective isolation enforcement. As Heracles does not have any coordination mechanisms for sub-controllers, isolation enforcements are performed sub-optimally.

For `xapian`, the performance improvement is significantly higher than Heracles, because the SLO of `xapian` is very low and small sub-optimal contention management can be harmful for the performance of the workload. Heracles may easily perform wrong isolation for resource contentions that results in the high latency (76×). On the other hand, *HIS* continuously tracks all resource contentions and mitigates the most significant one effectively as well as chooses isolation techniques considering their characteristics.

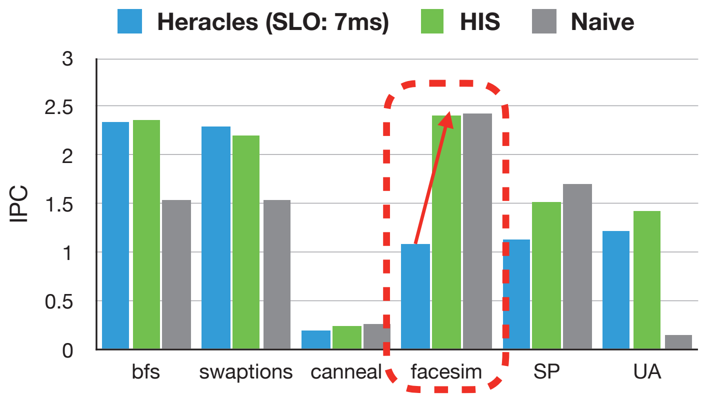
Besides latency-critical workloads, we also compared the throughput of best-effort workloads. Figure 3.12 shows that both *HIS* and Heracles achieved similar performance for the best-effort workloads. For the most improved cases, marked red-dotted circles in graphs, we observed that latency-critical workload's performance is not improved and same as Heracles. It means *HIS* performs isolations more effectively than Heracles and achieves higher resource efficiency, thus achieving higher background workloads while maintaining the similar performance of co-located latency-critical workloads.



(a) moses (QPS: 140)



(b) sphinx (QPS: 5.6)



(c) xapian (QPS: 1540)

Figure 3.12: Comparing the performance of best-effort workloads

We also tested the sensitivity of scheduling intervals for Heracles. Heracles has relatively high scheduling interval to collect meaningful tail-latency feedbacks. Therefore, when it decides wrong decisions, large interval may be harmful for latency of latency-critical workloads. We adjusted interval configurations of high-level controller from fifteen seconds to 100 milliseconds and also adjusted that of sub-controllers from two seconds to 100 milliseconds correspondingly.

Figure 3.13 describes the how scheduling interval affects to the Heracles. In this experiments, we observed that shorter intervals than one second results in higher latency. Heracles allocates more resources to best-effort workloads until any SLO violation or high loads, thus it conservatively allocates resources to the best-effort ones. In case of shorter isolation intervals (i.e., 0.1 & 0.5 seconds), sub-controllers aggressively increase resource allocations for best-effort ones which is likely to make more frequent SLO violation or higher latency. In addition to that issue, tail-latency feedbacks may be not enough for successful isolation decisions for high-level controller.

As illustrated in the figure, *HIS* outperforms various settings of Heracles, because *HIS* effectively finds which contention is major and rapidly adapts to the contention fluctuation which may lead to performance degradation.

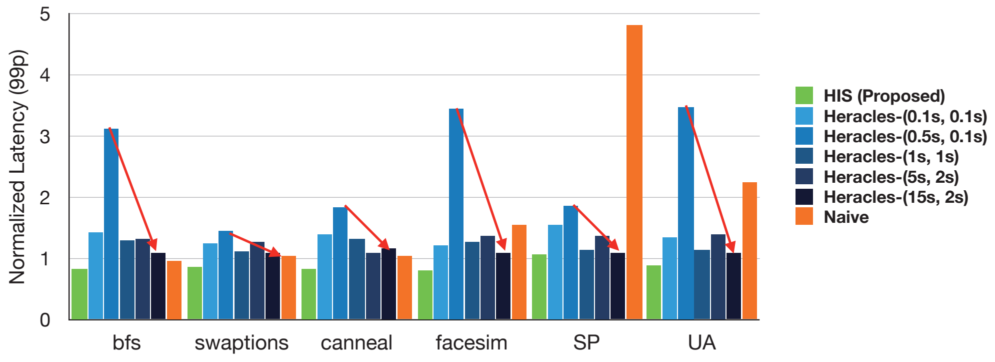


Figure 3.13: Scheduling Interval Sensitivity Tests for Heracles. (Latency-critical Workload:Sphinx, QPS:5.6, and SLO:3s)

3.6 Related Work

There have been many studies on isolation approaches used in multicore systems. Software isolation is widely used in most multicore systems. CPI^2 [7] detects the performance anomaly and identifies the suffered a *victim* workload using statistics of CPI(Cycles Per Instruction), and throttles the CPU usage of the *antagonist* for performance isolation. Their work is inline with ours in terms of throttling background workloads with software isolations. However it only uses software techniques which provide less strict isolation, thereby needs harsh CPU hard-capping for antagoist for strictness (i.e., 0.01 CPU-sec/sec).

Memguard [8] isolates the memory bandwidth contention based on its memory budget. It utilizes a software isolation that throttles memory access of each workload by restricting CPU cycles, thus each workload's memory bandwidth can not exceed the assigned memory bandwidth. Similar to our work, it isolates memory resources by reserving memory bandwidth, but it does not utilize hardware isolation technique, so there is no guarantee for strict isolation. However, our work uses hardware isolation techniques to supplement strictness. Both CPI^2 [7] and Memguard [8] isolate workloads by throttling CPU using a software technique, and they can mitigate memory contention easily. However software techniques may result in unintended interferences under the co-location of workloads showing bursty behaviors.

Dirigent [17] is a fine-grained isolation runtime system which partition an LLC and throttle CPUs. Similar to ours, it exploits hardware isolation techniques such as hardware cache partitioning and per-core DVFS to meet the SLOs of a latency-sensitive workload while backfilling batch workloads to improve resource efficiency. Our work is in line with their work [17] in terms of providing fine-grained isolations for considering the characteristics of workloads. However, we focus on the adaptive enforcement of multiple isolation techniques according to the characteristics of workloads, thus we can take more options for better performance isolation.

Quasar [11] utilizes a machine learning algorithm to infer which colocation mostly mitigates the shared resource contention, and uses scheduling and thread migration, which is the software approach, for isolation of consolidated workloads. Their work is inline with ours in terms of multiple isolation techniques In contrast, they only uses software isolation techniques for higher flexibility which can not provide strict and fast isolation.

Heracles [25] and PARTIES [26] isolate workloads by partitioning and throttling resources using both hardware and software isolation schemes to meet SLOs of production workloads while increasing resource efficiency. Similar to ours, their works are inline with ours in terms of using multiple isolation techniques for multicore systems. However, their works do not consider the tradeoffs between isolation techniques which can be harmful for the strictness and flexibility.

3.7 Conclusion

We developed a hybrid isolation system that utilizes hardware and software isolation techniques in a hybrid manner by the characteristics of the workloads. We have explored the tradeoffs between hardware and software isolation techniques, and illustrated how these properties affect performance of consolidated workloads. We have proposed an algorithm for isolation to use isolation techniques mutually complementary through characteristics analysis of workloads and comparison of each isolation technique. Our experimental results show that our approach can improve the performance of foreground workloads in terms of execution time than the static software isolation by from $1.7\times - 2.14\times$, and also improves tail-latency from $1.4\times - 76\times$ compared to the-state-of-art isolation framework while improving resource efficiency under the different levels of contentions.

Chapter 4

EdgeIso: Effective Performance Isolation for Edge Devices

4.1 Introduction

As the Internet of Things has evolved, numerous types of applications have emerged increasingly. Virtual realities [33], wearable devices [34], smart factories [35, 36], autonomous drones [37], and self-driving cars [38, 39] are such emerging applications, and these require low latency along with efficient data processing. However, the existing clouds present relatively high latencies, which are not enough to meet the service level objectives (SLOs) of those services [40]. For this reason, edge computing is getting attention as a complementary solution.

Although edge reduces the latency by placing services close to users and data sources, it still has a potential problem that might violate SLOs; *resource contention* caused by multitasking for the tasks with different resource demands and SLOs on edge. For example, the latency-critical tasks which interact with users (or sensors) can be colocated with the batch ones to process data from them to improve the quality of services [37, 41]. As batch tasks typically tend to consume lots of resources, it can result in unpredictable and degraded performance for the latency-critical tasks. Moreover, the problem can be worse by the edge's *integrated architecture* and *limited capacity*. Edge devices, such as NVIDIA Jetson TX2, have integrated CPU and GPU architecture sharing memory and memory bandwidth. These architectural characteristics may impose frequent and severe contention for memory when multitasking on edge [42].

There have been several studies to handle these challenges on edges. An approach is offloading heavy computations to clouds [43–45]. Similar to ours, it can provide low latency for tasks on edges, but it does not consider resource contention caused by multitasking and requires additional resources for offloading. Another approach is resource reservation for recurrent tasks based on resource capacity such as CPU and memory [46,47]. Our study is in line with these studies in terms of achieving higher resource efficiency by consolidating tasks on nodes. However, they assume the recurrent characteristics of cloud tasks that may not adapt to the edge's dynamic environment. Resource isolation can adjust the degree of accessing the shared resources by using several isolation interfaces [9, 17, 23, 25, 26]. This approach is used in many cloud systems to isolate performance among tasks. Our study is in line with these studies in terms of providing isolations for latency-critical tasks. However, these cloud frameworks do not consider multiple latency-critical tasks scenario, which is common in edge nodes [9, 17, 25], or have insufficient capabilities to support more strict SLOs and adaptive isolations for edges [23, 26].

In this chapter, we propose *EdgeIso*, a lightweight user-level scheduler that effectively isolates the performance of tasks while maximizing resource efficiency by (a) profiling the dominant resource contention for tasks, (b) detecting the phase changes of tasks such as load fluctuations, and (c) performing isolation techniques incrementally and adaptively. To identify the dominant resource contention, we present an efficient online profiler that utilizes the fact that insufficient allocation for the dominantly used resource of the latency-critical task affects its performance negatively. The profiler monitors performance counters of both the latency-critical tasks' *co-run* and *solo-run* and calculates the sensitivities for resource contentions to determine the dominant resource contention. Also, we developed *EdgeIso* to perform an isolation technique incrementally. Although each isolation affects the subsequent shape of resource contentions, this design allows the scheduler to search the best configuration for performance isolation effectively. We focus on one of the most crucial resource contentions for edges, including CPU, last-level cache (LLC), and memory bandwidth. For these resource contentions, we utilize three isolation techniques; CPU core allocation, CPU cycles throttling, and GPU core frequency scaling. Although *EdgeIso* mainly deals with computation and memory contention, it is applicable to cover other types of resource contentions (e.g., network and storage I/O). Lastly, we implement a phase detection mechanism that adapts to the load fluctuation of tasks or changes in resource demands, thereby avoiding performance degradation and oscillatory behaviors.

We evaluate our proposed scheduler on an NVIDIA Jetson TX2, which has four CPU cores and 256 GPU ones using batch data processing and latency-critical object detection tasks. Our preliminary experimental results show that *EdgeIso* can effectively mitigate resource contention, achieve the performance of the latency-critical and foreground data processing tasks comparable to that of its *solo-run*, and meet diverse levels of SLOs of latency-critical tasks. We also find that our scheduler can meet SLOs of latency-critical tasks compared with the other alternative schemes, such as offloading to cloud and resource reservation.

This chapter makes four contributions as follows. First, it identifies the key challenges of multitasking on edge devices that have integrated SoC architecture and limited resources. Second, it describes a profiling technique the measures dominant resource contention and also describes a practical phase change detection mechanism for the resource contention on edges. Third, it describes the design and implementation details of *EdgeIso*, which uses several isolation techniques for effective performance isolation between tasks having diverse levels of SLOs. Fourth, it evaluates *EdgeIso* on an NVIDIA Jetson TX2 device to compare with other alternative approaches using the selected set of benchmarks.

4.2 Motivation and Related Work

4.2.1 Motivation

To show how much resource contention affects the performance of the latency-critical task, we ran object detection task on GPU and a single core, and also ran each STREAM [2] as a batch task, which widely known as memory bandwidth benchmarks, on a single and its dedicated core. Figure 4.1b shows the performance of a latency-critical task under memory contention. As increasing the number of batch tasks, the tail latency of object detection is degraded by up to 3$\times$ compared with when the no batch task runs. The results show that shared resource contention can be critical to latency-critical tasks on edge devices.

Diverse SLOs. Edge applications are different from the cloud ones in that they have relatively more strict latency requirements (e.g., $<$300ms) [34, 40]. Like cloud applications [26], SLOs of edge ones are diverse and ranging from seconds to milliseconds [34, 48]. Edges usually run multiple latency-critical tasks and also run batch tasks such as data processing and retraining tasks [41]. The diverse SLOs complicate performance isolations in edges. In order to meet SLOs on edges, accurate and careful resource management is necessary because wrong decisions lead to significant violations of the SLOs, which may affect user satisfaction or accidents [48].

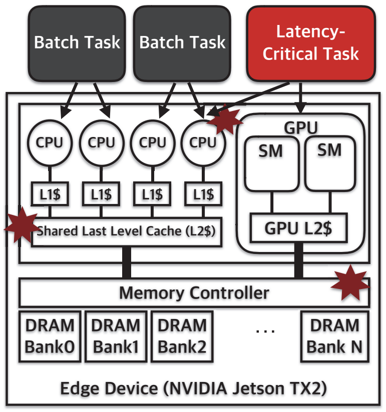
High Load Fluctuations. Edge applications often suffer from high fluctuations in offered traffics caused by the dynamic environment such as user mobility and the correlated events [49]. This high fluctuation indicates the frequently changing resource demands of tasks. To deal with the load fluctuations, some cloud schedulers provide reservation schemes for resources in advance [46, 47]. However, edge devices have constrained resources; thus, it is not a viable solution. Even though there exists adaptive runtime for SLOs, it is critical for the runtime to quickly detect the fluctuations and deal with them effectively for edge devices.

4.2.2 Related Works

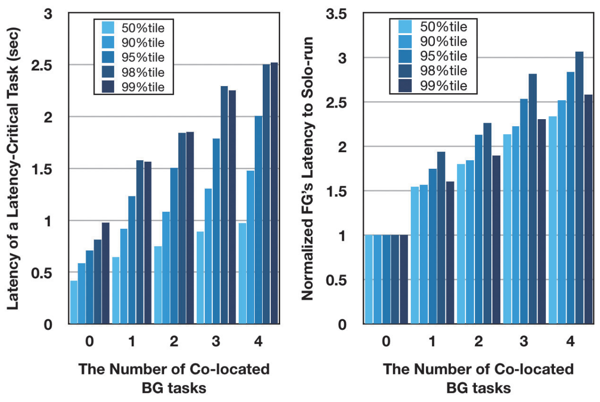
There are several techniques for solving these challenges. Figure 4.1c illustrates the difference between the existing approaches and *EdgeIso*. Table 4.1 highlights the comparison of benefits between the existing techniques and *EdgeIso*.

Offloading. Edge applications can mitigate resource contention by offloading the heavy tasks to the clouds. This approach provides low latencies for latency-critical tasks as well as high throughput for batch tasks. However, it can waste the resources of an edge node during the idle time of a latency-critical task.

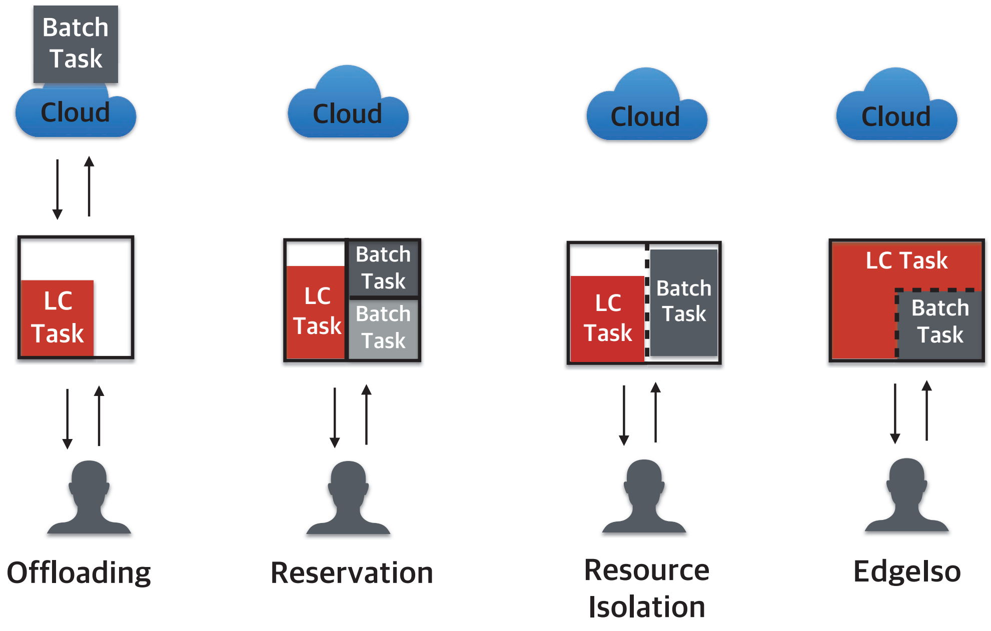
Steel [43] provides the interface for developers to make and deploy edge applications easily. It also can provide low latency for latency-critical tasks similar to ours and quickly moves edge tasks to other nodes by using their interface. However, it has yet no consideration for the resource contentions between multiple latency-critical tasks. Neurosergeon [45] suggests the offloading scheme which performs layer-wise partitioning of a neural network to improve performance or energy efficiency when running a DNN workload on edge-cloud. Their work is in line with ours in terms of improving the latency of the latency-critical task. However, their work also does not consider the multitasking scenario, which can lead to the resource contention on which we focus. Semantic Cache [44] offloads inference tasks to clouds and performs caching the results of them on edges to achieve low latency. However, it also uses a small neural network as an encoder to learn and classify the similarity of features between incom-



(a)



(b)



(c)

Figure 4.1: (a) Shared resource contention between tasks on an edge device with integrated CPU-GPU architecture. (b) Performance degradation of the latency-critical task (object detection; Single Shot Multi-box Detection [1]) by increasing the number of co-located memory intensive batch ones. We used STREAM benchmark [2] as a batch one. (c) Comparison between existing isolation techniques and *EdgeIso*.

ing images and cached ones, which may need more resources to improve its accuracy. Therefore, their work is in line with ours in terms of providing low latency. However, it also has a potential resource contention problem.

Reservation. Another technique is resource reservation for tasks based on the resource capacity. This approach already adopted in many clouds to meet SLOs of various tasks. As illustrated in Figure 4.1c, the reservation scheme carefully allocates resources to tasks by their priorities or resource usage history. It can guarantee certain levels of SLOs and resource efficiency by consolidating tasks as much as possible based on the resource utilization such as CPU and memory. However, it does not consider the contention for shared resources, which can degrade the performance of tasks. Moreover, it may not adapt quickly to edges where execution patterns or resource contention change continuously according to the edge's dynamic environment.

Rayon [46] runs on top of YARN [50] and provide the *reservation* scheme for tasks based on resource capacity and the deadlines of tasks. Rayon guarantees the tasks' deadline by re-ordering the execution and allocating CPU and memory resources according to the submitted resource requests. However, it does not consider the contention for shared resources such as LLC and memory bandwidth. Our work is similar to prioritize latency-critical tasks to meet their SLOs. Besides, we also consider the contention for shared resources. TetriSched [47] proposes a *plan-ahead* reservation scheme to know when the resource is available for other tasks based on the history of recurrent tasks. Using history data, It helps to schedule tasks to wait for resources of a node or allocate to other nodes. However, similar to Rayon, it does not mitigate the contention for shared resources, and more importantly, they assume the recurrent execution patterns aiming to cloud tasks. Also, Tetrisched is in line with our work in terms of improving higher resource efficiency. However, our work can not only handle shared resource contention but also adapt load fluctuation more effectively.

Resource Isolation. There are several resource management runtime for clouds ex-

Table 4.1: Comparison between existing isolation techniques. ((✔) shows references partially working on the feature)

	Isolation Techniques			
	Offloading [43–45]	**Reservation** [46,47]	**Resource Isolation** [9,23,25,26]	**EdgeIso**
Low Latency (for Latency-Critical Task)	✔	✔	✔	✔
High Resource Efficiency		✔	✔	✔
Quick Adaptation	(✔) [43]		(✔) [9]	✔
GPU Task Isolation				✔

ploiting resource isolation techniques. Leverich et al. [23] solve the performance degradation when consolidating multiple latency-critical tasks by adjusting virtual runtime and task awakening mechanism in Linux scheduler by giving some awakening slack to prioritize the task. They utilize kind of throttling CPU bandwidth. The resource isolation is similar to ours; however, they do not focus on the shared resource contention and quick adaptation, which is essential to edges. PerfIso [9] proactively allocates CPU cores for latency-critical tasks to meet their SLOs and quick adaptation for the load fluctuations. Similar to ours, it achieves low latencies and quick adaptation, however, it only focuses on CPU contention and not on shared resources such as memory bandwidth and LLC, which can be more critical to edge tasks. Both Heracles [25] and PARTIES [26] isolates shared resources for the latency-critical task to meet their SLOs by monitoring the task's current latency. They are inline with our works in that they use multiple resource isolation techniques, however, unlike ours, Heracles does not support multiple latency-critical tasks, and PARTIES randomly chooses resource isolation techniques exploiting resource fungibility which can lead to long convergence time for latency-critical tasks.

4.3 Design and Implementation

For effective isolation, it is necessary to identify which resource is the most contentious one. In order to identify this, we utilize the resource sensitivity, the degree of changes in resource usage under the contentions. The more resource usage changes, the higher resource sensitivity is. We will call the resource usage (or data) when a task runs alone as *solo-run* data, and when multiple tasks run together as *co-run* data. Calculating resource contention with the difference between *solo-run* data and *co-run* data is the simple but effective metric for measuring shared resource contention [18]. To utilize this approach, we develop the online profiler measuring these sensitivities. Also, we design and implement *EdgeIso* to perform isolation to reduce those changes in resource usage for the latency-critical tasks.

Figure 4.2 shows the architecture of *EdgeIso*. It mainly consists of a profiler and scheduler. Profiler periodically monitors the resource usage of latency-critical tasks and its contention running on the edge ($\textcircled{1}$ in Figure 4.2). Using profiled data, the phase detector evaluates whether the calculated resource contentions are larger than the predefined threshold or not ($\textcircled{2a}$). We define a *phase* as the degree of resource usages for a task during the successive profile intervals. If the changes exceed the predefined threshold, the profiler regards it as a change for the phase, which needs to update the task's resource usage to calculate the resource contention appropriately. Once a phase change is detected, the profiler updates the *solo-run* data to accurately calculate the resource contention.

Before monitoring *solo-run* data of a latency-critical task for its changed phase, the profiler stops all other tasks and run the latency-critical task alone during the predefined time ($\textcircled{2b}$). If there exist multiple latency-critical tasks, the profiler profiles each task in a round-robin manner by switching tasks in a fine-grained time slice not to affect the latency of tasks negatively. The profiler finds the dominant resource con-

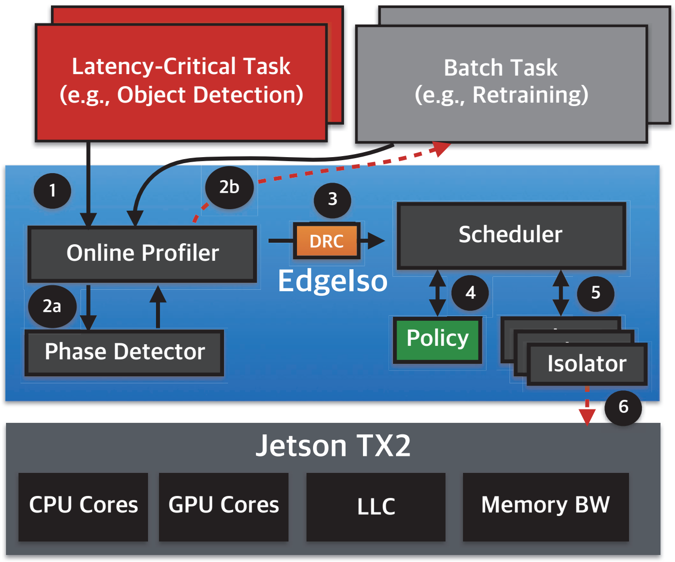


Figure 4.2: EdgeIso system architecture. It mainly consists of the *profiler* and *scheduler*. The black line shows data flow and the red dotted line shows enforcement (e.g., suspending tasks for profiling & performing isolations).

tentions (DRCs), which is the contention showing the most substantial change in resource usage due to its sensitivity to contention and sends it to the scheduler (③).

For isolation, the scheduler tries to find a latency-critical task that violates mostly its SLO. After that, it decides which isolation should be performed at the next interval considering the contention information and an isolation policy (④). After all, the scheduler invokes a selected isolator to perform isolations (⑤), and the isolator performs the isolation incrementally (⑥). In the following sections, we describe in detail how the profiler and scheduler work on edges to achieve our goals.

4.3.1 Profiling

When multiple tasks are running simultaneously, resource contention among tasks can be changed continuously according to the loads, behaviors, and interaction among the tasks. Thus, profiling the resource contentions for multiple tasks is difficult during scheduling the tasks.

To solve this challenge, we devise a profiling technique that effectively measures the resource contention for a task by suspending other tasks and monitoring the resource usage of a task alone for a predefined period in an online manner. By using the collected data, the profiler can measure the resource contention for the task by calculating the difference of resource usage between *solo-run* and *co-run*. We abbreviate the difference to *diff* in the rest of this section.

Figure 4.3 shows how the online profiler works for profiling the *solo-run* data. The profiler uses signals such as `SIGSTOP` and `SIGCONT` to pause and resume the execution of tasks. By doing this, the profiler can perform the profiling for a task's *solo-run* data for a predefined period whenever a phase has changed; in other words, the consuming pattern of resource usage changes. In case of where multiple latency-critical tasks are running on an edge node, the profiler profiles each latency-critical task during the predefined period and switches to another latency-critical one in a round-robin manner. In the worst case, all latency-critical tasks require entering the profiling stage. This approach reduces the latency penalty by amortizing their penalty of suspension periods at the cost of loss of accuracy. Even though the profiler inaccurately obtains profiled data at first, the subsequent profiling stages can minimize the inaccuracy and correct the profile. This approach is practical in terms of minimizing the overhead of profiling while collecting *solo-run* data for a new phase. Given the profiled data, the profiler calculates the resource contentions (*diff*s). It is one of the most effective methods for profiling resource contention in an online manner [30].

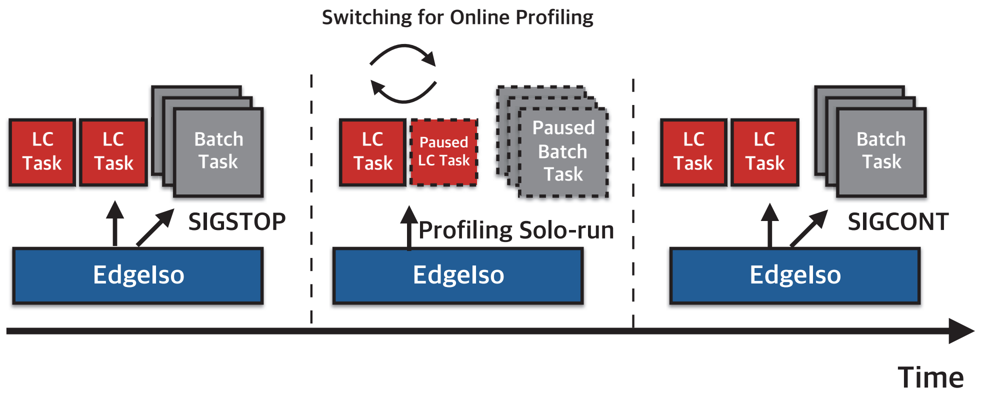


Figure 4.3: Profiling the resource contention for the latency-critical task by monitoring resource usages when the task runs alone.

Depending on the values of *diff*s, the profiler finds which resource is contentious at the time. If the value of *diff* is below 0 (i.e., $-1 < diff < 0$), it means that the resource is currently under contention, otherwise, the resource contention does not exist for the resource and more resource is used compared with resource usage of the task's *solo-run* (i.e., $0 < diff < 1$). The definition of *diff*s for various resource types and how DRCs are determined using *diff*s will be described in the dominant resource contention section. The profiler collects and tracks the changes in *diff*s of the latency-sensitive task and periodically informs that to the scheduler.

With the *diff*s, the profiler checks whether the phase has changed. If it occurs significant changes in *diff*s repeatedly (e.g., one of the abs(*diff*s)>1 for three times), then the profiler considers the current phase has changed and triggers profiling stages. We set three for the threshold to avoid false positives, but it is a tunable one. As the profiling stage is triggered, the profiler informs it to the scheduler to stop all the other tasks, and profiles the *solo-run* data of the phase-changed latency-critical task during the predefined period.

Metrics. Currently, we focus on the resource contention for CPUs, last-level cache

(LLC), and memory bandwidth. However, we implement a monitor interface in the profiler, which can be easily extensible to incorporate other types of resource contentions. We use Linux *Perf* and `/proc` to monitor the values of metrics. For measuring LLC contention, we use the LLC hit ratio, which reflects the reuse of LLC. For memory bandwidth, we use memory bandwidth utilization by using the LLC misses, the bytes of a cache line, and a maximum bandwidth of the edge devices. For CPU contention, we use the number of active threads (for detecting contention) and instructions per core (for allocating cores).

Dominant Resource Contention. Identifying the dominant resource contention of latency-critical tasks is essential for efficient and effective performance isolation. We define the *dominant resource contention* (DRC) for a task as the resource having the largest negative value of *diff*, except CPU contention. Specifically, for deciding the dominant resource contention, the profiler monitors *diff*s of multiple resource metrics; LLC hit ratio (LLC_HR) for LLC, memory bandwidth (Mem_BW) for memory BW, the instructions per second (IPS) and the number of active threads per allocated core (NumTh/cores) for CPU cores. Note, the number of active threads is used for detecting contention and IPS used for allocating cores, which will be described in the *scheduling* section. *diff*s are defined as follows:

$$diff_{LLC} = LLC\_HR_{co\_run} - LLC\_HR_{solo\_run}$$

$$diff_{Mem\_BW} = (MemBW_{co\_run}/MemBW_{solo\_run}) - 1$$

$$diff_{IPS} = (IPS_{co\_run}/IPS_{solo\_run}) - 1$$

$$diff_{NumTh/cores} = NumTh_{cur}/cores_{cur} - NumTh_{prev}/cores_{prev}$$

Before comparing multiple *diff*s, *EdgeIso* checks if the number of active threads exceeds the threshold (e.g., $> 2\times$ allocated cores), and if yes, then CPU contention is regarded as the DRC. Otherwise, *EdgeIso* compares other *diff*s, which are LLC hit ratio and memory bandwidth. Unlike other contentions, we deal with CPU contention differently, because it gives For example, if there is no change in the number of active threads, and *diff* s are -0.251 (LLC), 0.449 (Mem_BW), -0.126 (IPS), and 1 (NumTh/-cores), respectively, then *EdgeIso* determines the LLC contention as the DRC based on the given *diff* s.

Phase Change Detection. We consider the changes in the phase of the latency-critical tasks. Without the phase detection, the calculation of *diff* s can become wrong. Because the calculation is based on the sampled *solo-run* data during a short time, it can not represent the metrics during the whole execution. Therefore, the profiler keeps tracking of sampled data and checking whether a new phase comes in every one second. As shown in Figure 4.4, the profiler tracks *diff* s of resources for detecting the changed phase. To avoid oscillations, we only admit the values of exceeding the threshold (one of the abs(*diff*)s$>$1) for three times as the phase change. We choose three for detecting an actual phase change because more than three mismatches between isolator and DRC increase latency significantly. Additional optimizations for detecting phase also can be applied to ours [51, 52].

4.3.2 Scheduling

The objective of *EdgeIso* is to mitigate resource contention for meeting SLOs of the latency-critical tasks while improving maximizing resource efficiency by co-running batch tasks. In this section, we describe isolation techniques used to mitigate resource contention and illustrate how *EdgeIso* utilizes isolation techniques incrementally and dynamically.

Isolation Techniques. To mitigate the contentions for the shared resources, we use three resource isolation techniques: core allocation, cycle throttling, and GPU frequency throttling. GPU cores and CPU ones on the edge device share the memory bandwidth due to its inherent integrated architecture. For memory bandwidth con-

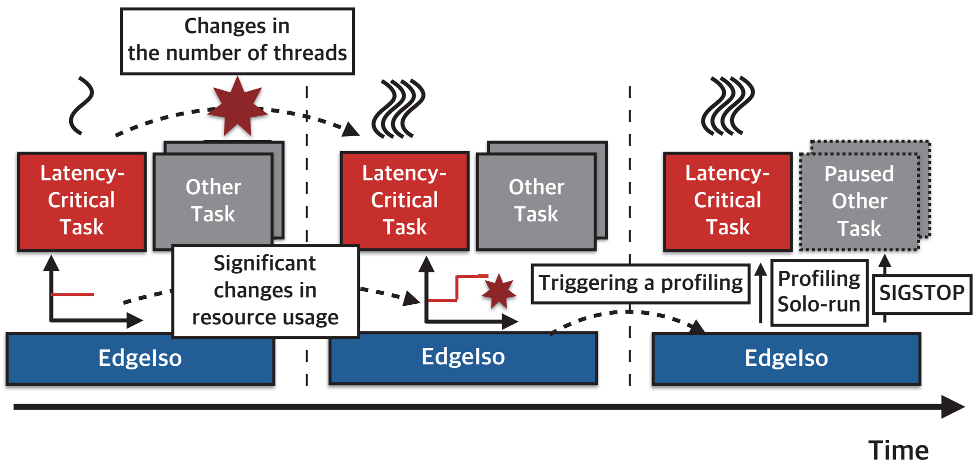


Figure 4.4: Illustration of detecting a phase change. *EdgeIso* triggers profiling for solo-run data when it detects either the significant changes in the resource usage or changes in the number of threads.

tention, we use the GPU frequency throttling and core allocation. Since there is no explicit isolation interface for memory bandwidth, we throttle the execution of the most contentious task, which accesses the memory frequently. For memory access of CPU cores, we control the number of cores allocated to the most contentious task by using `cgroup:cpuset`. For memory access to GPU ones, we utilize the interface for GPU power management. The range of possible GPU frequencies is from 140MHz to 1300MHz, and the number of possible steps is fourteen.

Jetson TX2 uses ARM processors, which do not support any LLC isolation techniques such as Intel *Cache Allocation Technology* (CAT) [3]. Therefore, in order to throttle LLC contention, instead of using LLC partition, we limited the cycles of tasks by using `cgroup:cpu`. However, for the edge devices may have such LLC isolation techniques (e.g., Intel Fog Reference Design (FRD) equipped with a Xeon processor [48, 53]), *EdgeIso* can exploit the technique to isolate the allocation of LLC. For CPU contention, we also utilize `cgroup:cpuset` to change the mapping between threads to CPU cores.

Algorithm. *EdgeIso* incrementally and adaptively performs isolation techniques to mitigate the contention between the latency-critical tasks and batch ones. We implement *EdgeIso* (1) to deal with dynamically changing contention by tracking phase changes and (2) to perform isolation incrementally based on the contention and current isolation. Reflecting the phase change is necessary to meet the SLO of latency-critical tasks while improving resource efficiency by running batch tasks more aggressively. Also, performing incremental isolations is important to find the best configuration for the isolations (e.g., parameters for each isolation).

In order to find the proper parameters, we implement a pluggable policy that decides the next isolation techniques. The policy dictates the next isolation technique and tasks to be isolated corresponding to the DRC of the most SLO violated latency-critical task. It also helps to avoid oscillations of choosing isolators by counting mismatches between isolators and DRC.

Figure 4.5 illustrates how isolation is selected based on the dominant resource contention. In the figure, DRC is memory bandwidth at *N*th interval. Therefore, the scheduler selects memory bandwidth isolator and proceeds isolations. Depending on the contention and the results of performed isolation, DRC may change to other resource types such as CPU or LLC. If DRC is memory bandwidth at the *N+1*th interval, the memory bandwidth isolator will proceed to perform isolation. Otherwise, DRC has changed to other types, and then the scheduler checks the counters of mismatches between isolators and DRC to comply with the policy. If the number of counters exceeds the threshold (e.g., three for our policy), the scheduler changes the current isolator to an isolator, which corresponds to the resource type, described in the isolation technique section.

Algorithm 4.1 shows the pseudo-code of the scheduler. Initially, *EdgeIso* evenly

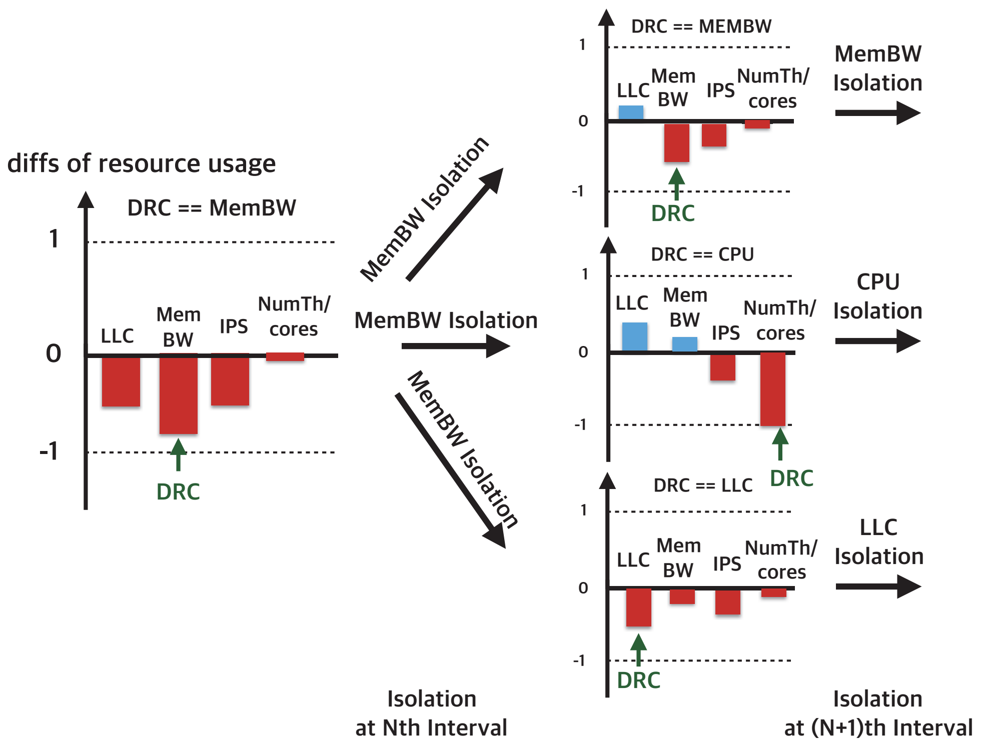


Figure 4.5: Illustration of incremental isolation. Performing isolation at the previous interval can lead to the other types of isolation at the very next interval. According to *diff*s, dominant resource contention (DRC) can be determined differently and the corresponding isolator can be invoked.

allocates the resources to tasks. In the case of two tasks, *EdgeIso* allocates half of the CPU cores to the tasks, respectively. The scheduler periodically gets all *diff*s from latency-critical tasks (line 2). For appropriate isolations, the scheduler checks whether there is any profiled data (metrics of *solo-run*) for the latency-critical task to decide to trigger profiling (lines 3-4). If no previous profiled data exists or phase change has detected, by suspending other batch tasks, the profiler measures the resource demands of the latency-critical tasks for a short period and updates from them (lines 5-6). After that, the scheduler finds the victim, which is the most SLO-violated latency-critical task, DRC of the LC task, and checks the current isolator (line 7-9). Then, the scheduler determines which task is the most contentious task based on DRC (line 10). After deciding target task, current isolator and chosen contentious task(`cont_task`), the next isolation technique to mitigate the contention is decided (line 11). To avoid oscillations when enforcing isolations, *EdgeIso* counts the number of mismatches between the current isolation technique and the DRC. When the mismatches occur over three times, the scheduler recognizes it as the real changes of DRC and decides to use the other isolation techniques for the subsequent isolation. Otherwise, the scheduler considers it as the temporal changes of resource contention and ignores them.

Once the scheduler decides an isolation technique, it is necessary to determine how much isolation will be performed for the tasks (lines 12-19). To determine whether the stronger isolations or weaker one for the latency-critical tasks, *EdgeIso* uses *diff*s of the tasks. If the contention is getting smaller by performing the isolation, the performance interference for the tasks is also decreased. *EdgeIso* searches for the isolation configurations for a resource contention until *diff* is reduced below 0.5%, which can be tunable (lines 12-19). Although the scheduler fails to reduce *diff*, the search process can stop if it reaches the boundaries for the configuration (lines 18-19). For example, when using GPU DVFS for throttling memory access for GPU cores, configurable minimum core frequency is 140MHz. As the search process reaches to 140MHz, then it ends to find more configurations.

Algorithm 4.1: Pseudo-code of the EdgeIso Algorithm

```
while True:
    diff_info = get_all_diffs_of_LC_tasks()
    phase_change = phase_change_detection(diff_info)
    if phase_change:
        // Profiling solo-run data for the changed phase
        solo_run_data = profiling_solo_run()
        update_data(solo_run_data)
    victim = get_SLO_violated_LC_task()
    drc = dominant_resource_cont(victim)
    cur_iso = get_cur_isolator()
    cont_task = choose_iso_target_task(drc)
    // Deciding the next isolator
    next_iso = decide_next_isolator(drc, cur_iso, cont_task)
    // Monitoring contention (diff_info) and determining the
       next isolation step
    next_step = next_iso.monitor_contention(diff_info)
    switch next_stepdo
        case strengthen
            next_iso.strengthen()
        case weaken
            next_iso.weaken()
        case stop
            next_iso.set_idle()
    sleep(0.2) // Sleep during predefined interval
```

4.3.3 Overheads

The overhead of *EdgeIso* comes from the profiler, phase detection, and scheduler. The profiler monitors the resource usages in every 200ms by attaching a monitor thread to every co-running task for reading performance counters. We find that they consume less than 5% CPU utilization per task. In our prototype, the profiler performs asynchronous I/Os, and it does not affect the performance of tasks as well. The overhead from phase-detection consists of the overhead of profiling *solo-run* data for the latency-critical tasks. Every detection of a phase change takes a two-seconds delay for all batch tasks. We empirically choose two seconds because it is the minimum interval for capturing changing contention for our experiments. However, this is tunable parameters depending on the platform and the types of tasks. Although batch tasks' overhead, the latency benefit for the latency-critical tasks is much more significant, considering batch tasks' SLOs. They mostly run in a best-effort manner with low levels of SLOs. In the experiment, we find that the total number of invocation of phase detection is two times in the case of Figure 4.9a, which is acceptable. The overhead of the scheduler itself is not as much as 10% in terms of the CPU utilization. We run the scheduler on the cores that execute the batch task. Thus, the overhead of the scheduler does not affect the performance of the latency-critical task.

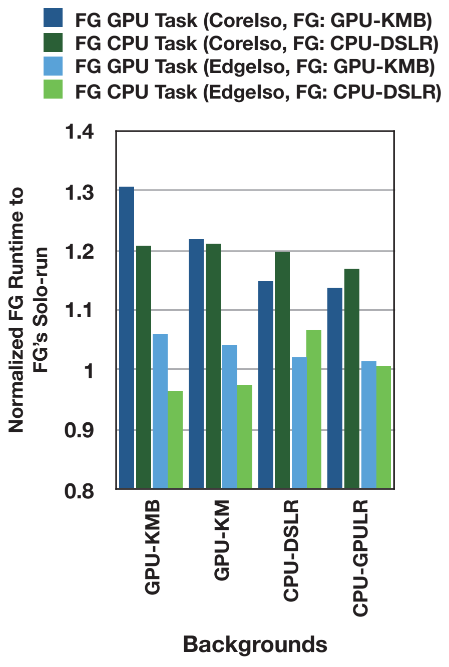
4.4 Evaluation

We performed experiments on a Jetson TX2, equipped with four ARMv8 cores, a Pascal GPU(256 GPU cores), and 8GB RAM installed Samsung 860 Pro SSD 512GB. The maximum memory bandwidth of Jetson TX2 is 50GB/s. We disabled two Denver cores in the following experiments and only uses ARM cores and a GPU. In Jetson TX2, Denver CPUs are disabled as a default due to their high energy consumption. Enabling these cores gives powerful processing capabilities but also consumes power significantly. Denver cores are useful running some tasks in isolation since they have dedicated caches separated from caches connected to the other four ARM cores. However, they also share memory bandwidth with other ARM cores and accelerators. In order to evaluate data processing tasks, we used the four SparkGPU benchmarks [54]; SparkDSLR (CPU-DSLR), SparkGPULR (CPU-GPULR), GpuKmeansBatch (GPU-KMB), and GpuKmeans (GPU-KM). The prefix indicates the types of cores on where the workload runs, and the suffix means the name of the SparkGPU benchmark. For example, GPU-KMB is a benchmark, which name is KmeansBatch and runs on GPU For object detection, we chose Single Shot Multi-box Detection (SSD) [1] for the inference task and its retraining one. We used the pre-trained VGG models and carefully adjust parameters not to exceed the memory capacity to avoid the crash of tasks by the out-of-memory (e.g., retraining tasks train runs two images in a batch, and learning rate 0.01). For input datasets, we used the subset of VOC-2007 [55], and each dataset has 200 images.

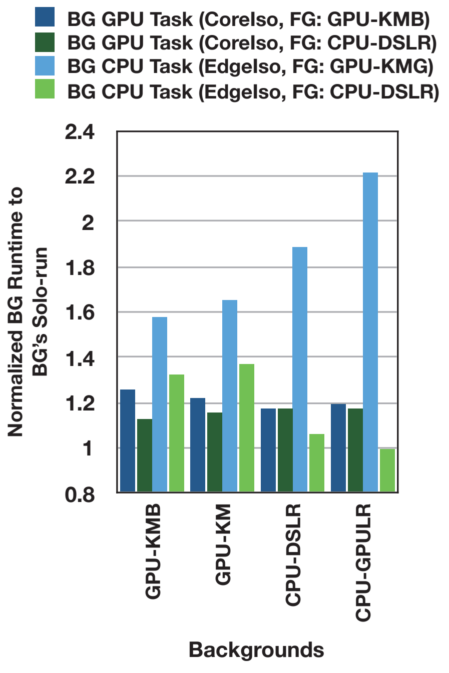
4.4.1 Data Processing Task

In the first experiment, we show that *EdgeIso* can successfully isolate the performance of the foreground task, even though the background one is co-running. As shown in Figure 4.6a, the performance of foreground tasks is improved by 1.2$\times$ compared with that of their *co-run*s, thus reducing the memory contention completely. Moreover, *EdgeIso* effectively isolates the performance of foreground tasks and improves the performance as much as their *solo-run* (*No Contention*). Some tasks (i.e., GPU-KMB and CPU-DSLR in Figure. 4.6a) run a little bit faster than the case of *solo-run* because the false sharing in batch tasks is eliminated by allocating memory in a misaligned way [30].

This improvement can be attained since our scheduler periodically profiles the dominant resource contention and enforces multiple isolation techniques adaptively to



(a) Execution time of FG tasks



(b) Execution time of BG tasks

Figure 4.6: Performance comparison of *No Contention* (i.e. solo-run), *Core Isolation* (i.e. simple core isolation using `cgroup:cpuset`), and *EdgeIso* on a Jetson TX2.

reduce the resource contention aggressively and incrementally. To improve the performance of the foreground task, *EdgeIso* throttles the background task. Figure 4.6b shows the performance of background tasks. It shows the degraded performance by up to 2.2× that of the background task's *solo-run* in case of co-running GPU-KMB and CPU-GPULR. Although two data processing tasks are competing, their characteristics of resource demands and SLOs can differ. If certain batch tasks have higher priority than other batch tasks, *EdgeIso* can give those tasks higher priority and runs at least contention.

4.4.2 Latency-critical Task

For the latency-critical task experiments, we used Single Shot Multi-box Detection (SSD) [1] for the inference task and its retraining one. We ran the SSD task on a GPU and two CPU cores as the foreground and also run retraining one on the rest of two CPU cores as the background. In order to test the performance of the latency-critical task when different input images are incoming, we prepared four groups of image datasets from VOC2007 by the number of objects in an image; small, medium, large, dynamic. There is only one object in an image of the small dataset. The medium dataset has 9 or 10 objects, the large dataset has from 25 to 42 objects, and the dynamic dataset made by combining three datasets.

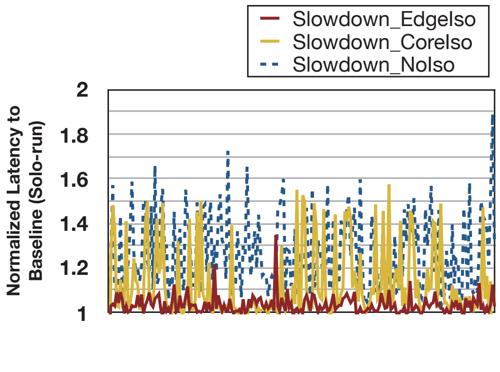
We evaluated three isolation schemes when an object detection task runs with a retraining task. We chose *small* and *dynamic* as input dataset to test stable or dynamic situation, respectively. First one is `NoIso`, which allocates CPU and memory resources without any isolation techniques, second one is `CoreIso` only pins dedicated CPU cores using *cgroup*, and last one is `EdgeIso`. In the case of `NoIso`, both tasks can run on all four CPU cores, but the object detection task uses GPU additionally.

As shown in Figure 4.7a and 4.7b, latency of `EdgeIso` is much stable and lower than other schemes. Although `CoreIso` isolates CPU resources, it still shows high latency variation compared with the baseline (*No contention*). It indicates that memory contention itself can increase latencies significantly. Interestingly, although the small object images show low latency and low variation without contention, it becomes higher latency variation under the resource contention, as shown in Figure 4.7b. We investigated those images and find that not only the number of objects in an image but also the size of objects affects the latency variation. From our observations, it turns out that the insufficient LLC resources to perform caching an object at that time makes latency spikes.

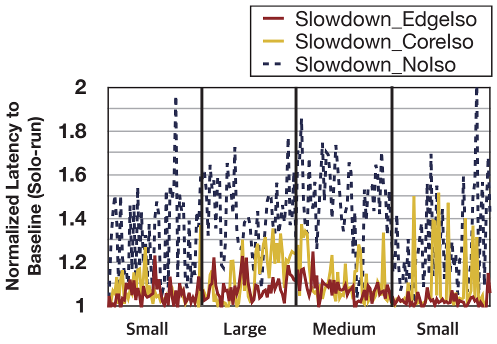
To show the SLO compliance, we set two different SLOs and compare the SLO violation ratio of `NoIso`, `CoreIso`, and `EdgeIso`. We defined an SLO as the degree of a slowdown from *solo-run*, considering the latency of object detection can vary depending on the number of objects. Figure 4.7c shows the SLO violation ratio when threshold is set to 10% and 20%, respectively. `EdgeIso` violates SLOs less than 10% for all dataset when the threshold is 20% (S-20% and D-20%). Even for *small* dataset, `EdgeIso` just violate 1%. When the SLO threshold is 10%, the ratio of SLO violation can become around 18% for `EdgeIso` when *dynamic* case (D-10%) at the worst case, but it is still lower than that of other schemes.

Figure 4.7d shows the normalized throughput of batch tasks for three schemes. We chose the instruction rate (IPS; instruction per second) to measure batch tasks' throughput. Unlike latency-critical task, `EdgeIso` shows the lowest throughput for the batch task, because it throttles the execution of batch ones to mitigate the memory contention caused by them. Despite its degraded throughput, the batch task usually runs in a best-effort manner. Also, this performance degradation can be acceptable in the edge environment where the more strict latency is required for the latency-critical task.

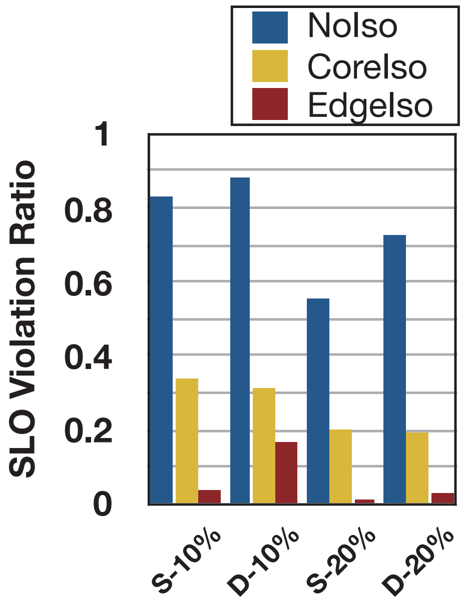
We also used Tailbench [32] to evaluate various types of latency-critical tasks running on CPU cores along with several batch tasks using GPU and CPU. In order to evaluate them on Jetson TX2, we follow the network benchmark setup in [32], and we used the Xeon-grade server as a client to generate loads enough to Jetson device. We set the QPS of each task as 300 for *img-dnn*, 1 for *sphinx*, and 1000 for *xapian*. We chose the GPU and CPU version of KMeans workloads used in Figure 4.6, and SSD retraining task used in 4.7. Figure 4.8 shows the normalized latency of each benchmark to *solo-run*. From the results, we can find that *EdgeIso* can deal with different levels of resource contention effectively. As shown in Figure 4.8a, *EdgeIso* achieves much lower latency than other schemes by mitigating memory contention. Because *img-dnn*



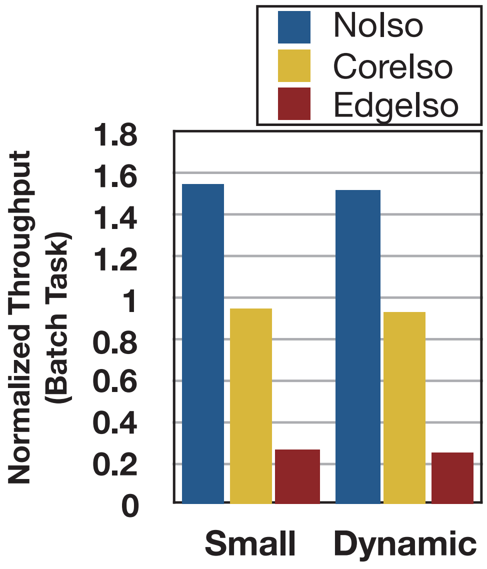
(a) SSD small dataset



(b) SSD dynamic dataset

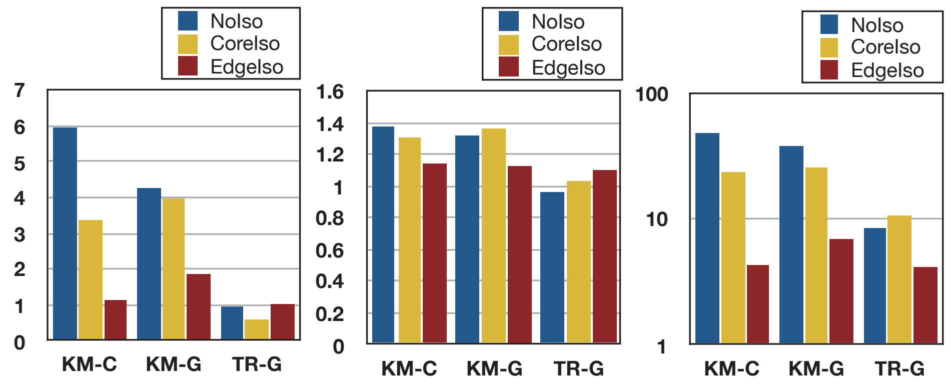


(c) SLO violation ratio



(d) Batch task

Figure 4.7: The normalized latency of object detection using Single Shot Multi-box Detection (SSD) on Jetson TX2. The *x-axis* represents the incoming images in each dataset (time goes from left to right), and *y-axis* represents normalized latency to its *solo-run* (*no contention*.) We set SLO thresholds as the 10% and 20% of slowdown from *solo-run*.



(a) img-dnn (b) sphinx (c) xapian

Figure 4.8: The normalized latency of Tailbench tasks on Jetson TX2. We evaluate three schemes. Each Tailbench task run as a foreground with co-running several batch tasks as background. (KM: kmeans, TR: SSD-training, C: CPU, and G: GPU)

task is memory-intensive one that consumes a lot of memory bandwidth, memory contention was a major factor of the performance degradation of latency-critical tasks. Figure 4.8b also shows similar or a little higher latency when using *EdgeIso* than the *NoIso* case, but in overall, we achieve lower latency. In the *xapian* case, we observed significant latency improvement compared with others, but this benchmark is so CPU intensive one with strict SLOs. Therefore, we obtained much higher latencies than the *solo-run* case.

4.4.3 Comparison with Alternative Approaches

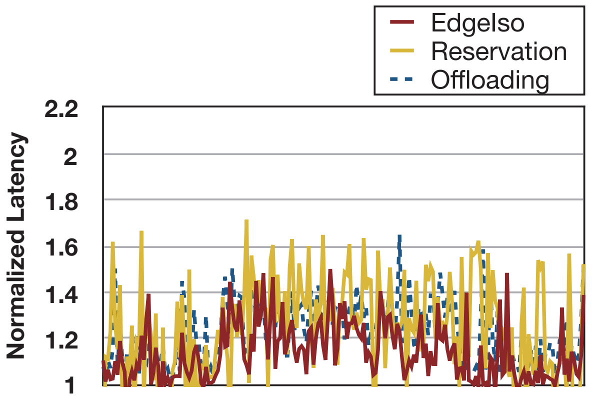
We also evaluated how much the *EdgeIso* is beneficial compared with the existing alternative methods of mitigating resource contention. We ran three tasks for evaluation; one latency-critical task and two batch tasks consume a different level of memory bandwidth. For the latency-critical task, object detection, used in the previous experiment, is selected, and k-means clustering (from SparkGPU Benchmarks) and retraining task of the object detection is selected as a batch task, respectively. We chose `dynamic` dataset which has high load fluctuations. Then, we placed and launched these tasks on the Jetson TX2 in the following way to show the effects of using alternative methods. In all scenarios, the latency-critical task runs on a GPU, and batch tasks run on the CPUs.

Offloading. It refers to the scheme that places or migrates a single heavy batch task from the edge node to the cloud and running the latency-critical task with a light batch one on the edge [44]. In the experiment, we place the latency-critical task and k-means batch task on the Jetson TX2 by allocating the dedicated CPU cores to them evenly. For a heavy batch task, we placed a retraining task to the Xeon E5-2683 v4 server, which is a typical type of server for the clouds.

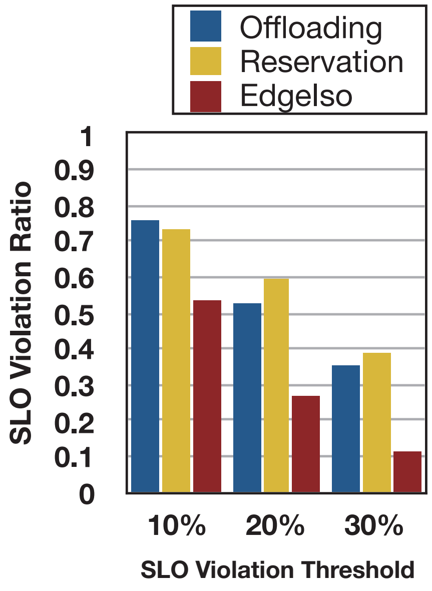
Reservation. This scheme allocates the user-requested amount of resources in advance and re-order the tasks' execution by their deadline [46, 47]. To mimic the reservation system, we allocated all tasks in the edge node carefully, maximizing resource utilization. Therefore, we placed the latency-critical task with the k-means batch task and retraining task in a Jetson TX2 and allocated the different cores to all tasks in the experiment.

Figure 4.9a shows the normalized latency of each alternative scheme and *EdgeIso*. Comparing with `Offloading` and `Reservation` in terms of request latency, EdgeIso shows 58.5%, 94.8% lower peak latency in the best case and shows 27.7% and 18.2% higher latency in the worst case, respectively. On average, EdgeIso shows 8.2% and 26.1% lower latencies than `Offloading` and `Reservation` one, respectively. For standard deviation, EdgeIso shows 9.8% and 33.4% lower standard deviation, respectively.

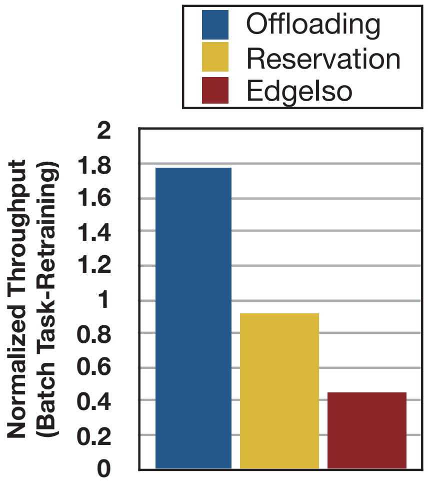
We can see that `Offloading` has similar latency for the latency-critical task compared with the `EdgeIso` in terms of peak latency. Since `Offloading` runs only two tasks on the Jetson, there is only one contender for the latency-critical task. In this case, the latency of `Offloading` can be better. Nevertheless, there is a slowdown



(a) Normalized latencies



(b) SLO violation



(c) Batch Task

Figure 4.9: Comparison with alternative approaches. The normalized latency of object detection using Single Shot Multi-box Detection (SSD) on Jetson TX2.

of latency due to the co-running batch task(i.e., *kmeans*), resulting in performance degradation.

In the case of `Reservation`, we mimic the reservation scheme by place all three tasks in Jetson node. The `Reservation` approach places tasks considering the resource capacity by using simple core allocation such as `cgroup:cpuset`. It shows the high latency variation and also shows the highest latency compared with others. Especially, we can see that when dataset with the small number of objects (for first 50 images of Figure 4.9a) are processed, `Reservation` shows very unpredictable latency, because the images in the dataset are sensitive to resource contention and other co-running batch tasks easily evict them.

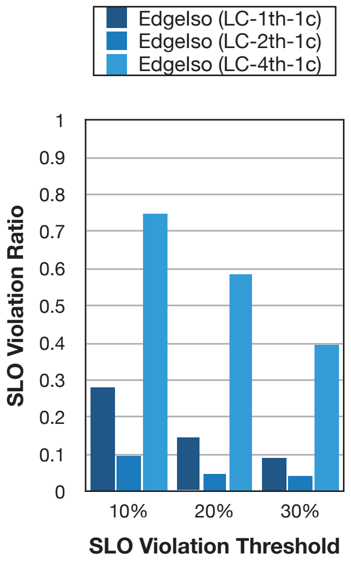
In the case of `EdgeIso`, the latency is similar to `Offloading`. However, the latency variation is much smaller. Even though `EdgeIso` runs three batches on a Jetson node, it shows the lowest median latency and low variation. From the results, we can find that `EdgeIso` can improve latency under high resource contention while maximizing resource efficiency by running batch tasks together. As shown in Figure 4.9b, we also find that all SLO violation ratios of `EdgeIso` are lowest compared with other approaches under all different SLO thresholds.

Figure 4.9c shows the throughput of re-training task when `Offloading`, `Reservation`, and `EdgeIso`. `Offloading` achieves the highest throughput in the experiment. Because we run the re-training task on the Xeon server, which is not suffered from any resource contention, the throughput can be higher than other approaches. However, there may exist resource contention in most cloud datacenters. This may result in performance degradation as well. `Reservation` attains similar performance to that of its *solo-run* at the latency penalty for the latency-critical task. `EdgeIso` shows the lowest throughput for the batch task. This performance penalty is due to the `EdgeIso`'s isolation policy which prioritizes the latency-critical task and reduces its resource contention from other tasks. Even though its performance is degraded, *EdgeIso* shows only half performance compared with when the no contention exists.

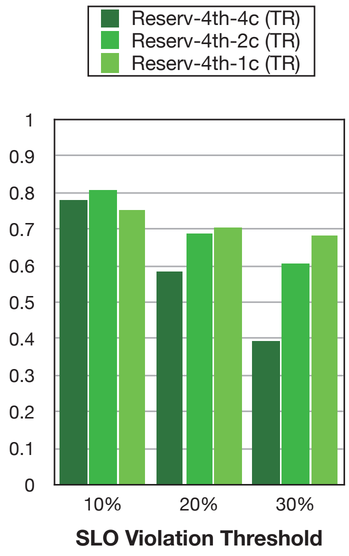
For more detailed comparisons, we conducted other experiments with diverse configurations for different thread-to-core mapping and different reservation schemes. We make two variations for reservation scheme. One is `Time Reservation (TR)` and the other is `Space reservation (SR)`. `Time Reservation` reserves certain percents of CPU cycles for workloads, and any other workload can not use the CPU cycles, which is similar to reservation scheme in TetriSched [47]. `Space Reservation` reserves a number of CPU cores for workloads which is used in the previous experiment in Figure 4.9.

We conducted experiments using a different workloads set used in previous experiments. While foreground task is same as object detection, background tasks are a linear regression task (SparkDSLR) and a kmeans task (CPU version of GpuKmeansBatch) in SparkGPU. Each thread setting is described in legend of each graph.

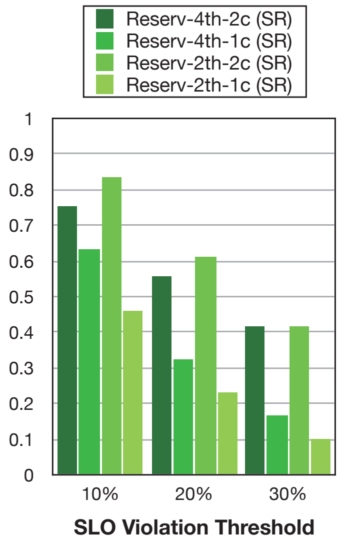
As shown in Figure 4.10, *EdgeIso* shows lower SLO violation ratio for various SLO violation thresholds than other schemes. Especially, `EdgeIso` achieves best performance when two CPU threads are used for data loaders by Figure 4.10a. It is because four threads case makes too much context switching for assigned single core and one thread case shows lower parallelism. In these experiments, we used 4 threads for best-effort tasks.



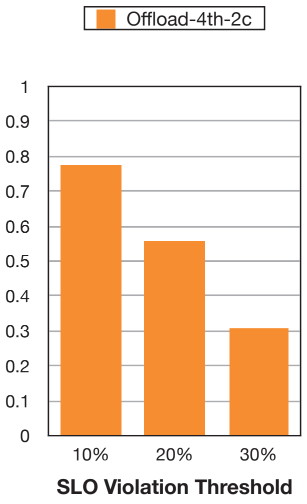
(a) EdgeIso



(b) Reservation (Time)



(c) Reservation (Space)



(d) Offloading

Figure 4.10: Comparison of SLO violation ratio for a latency-critical task (EdgeIso, Time Reservation, Space Reservation, and Offloading)

`Time Reservation` shows relatively worse performance than `Space Reservation`, because `Time Reservation` has to do more context switches than `Space Reservation` and the number of allocated CPU cores for best-effort tasks is smaller that it less access to the memory concurrently. Note, the number of threads and cores described in legends means thread and core settings for latency-critical tasks. For example, ”Reserv-2th-2c (SR)” in Figure 4.10c means `Space Reservation` and the number of threads and cores are two for latency-critical tasks, respectively. `Time Reservation` achieves similar but slightly worse performance to `EdgeIso` when the latency-critical task uses four threads, which is the worst case of `EdgeIso`. On the other hand, `Space Reservation` shows similar but slightly worse performance to `EdgeIso` when the latency-critical task uses one or two threads, which is the best case of `EdgeIso`. In summary, `EdgeIso` shows higher performance (lower latency) $2\times$ and $4\times$ than `Space Reservation` and `Time Reservation`, respectively.

For `Offloading` scheme, we moved more heavy task (kmeans) to another Jetson TX2 node, and ran two tasks (object detection for a latency-critical task, and linear regression for a best-effort one). Comparing with ours, `Offloading` is slightly better when threshold is $20-30\%$ in terms of SLO violation ratio. However, `Offloading` show higher SLO violation ratio compared with the best case of `EdgeIso` (i.e., cases of one or two threads). In summary, `EdgeIso` shows $3-6\times$ higher performance (lower latency) than `Offloading` scheme.

We also compared the performance of best-effort tasks running on background by aggregating instructions per cycle of all two best-effort tasks. As shown in Figure 4.11, we found that `EdgeIso` allows best-effort tasks running at high throughput as much as `Offloading` that needs additional hardware resources to run the offloaded task. For both `Reservation` schemes, they show lower throughput for best-effort tasks, because they degraded the throughput of best-effort tasks due to smaller resource allocation or larger context switching.

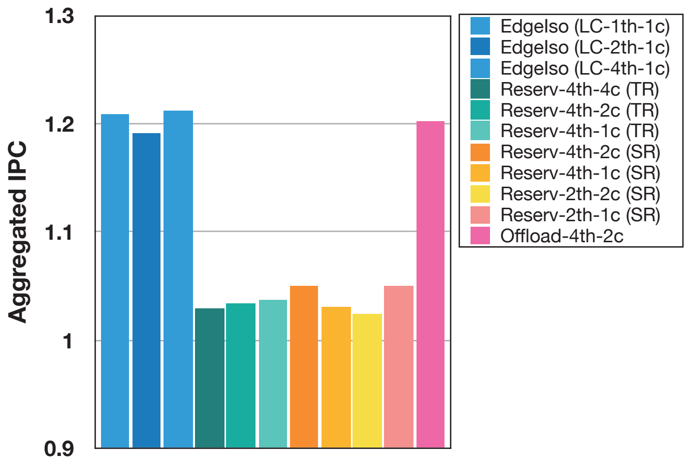


Figure 4.11: Performance comparison for best-effort tasks (EdgeIso, Time Reservation, Space Reservation, and Offloading)

4.5 Conclusion

In this work, we have illustrated the challenges of multitasking on edge devices, which is critical to the performance of tasks. To address these challenges, we present *EdgeIso*, an effective edge scheduler that mitigates resource contention on edges and isolates the performance of latency-critical tasks while running background tasks. It periodically profiles which resource contention is the dominant one for the tasks running on the edge, tracks the changes in the contentions efficiently, and performs isolations adaptively and dynamically by enforcing the appropriate isolation techniques incrementally. We have evaluated *EdgeIso* on an NVIDIA Jetson TX2 using several benchmarks of diverse SLOs and resource demands. The evaluation results show that *EdgeIso* can effectively mitigate resource contention, reduce the SLO violation ratio significantly for tasks of diverse SLOs compared with existing schemes under dynamic resource contention.

Chapter 5

Workload-aware Resource Management for Software-Defined Compute

5.1 Introduction

As cloud computing industry has been growing rapidly and becoming mature, more and more diverse and heterogeneous workloads have been running on the datacenters. There are a variety of workloads such as big data analytics, scientific workloads, social networks, and other web service workloads in the datacenter. These workloads have their own service goals, and thus it is important to meet these service objectives for user satisfaction [24].

However, it is difficult to efficiently and effectively run these workloads in datacenters because of their different resource demands for the resources. If the demands conflict, the contention for the limited shared resources increases. In order to minimize the contention, it is important to know workloads' resource consuming patterns. Especially, CPU and memory are most intensively shared resources considering the multicore architecture that most of servers have, and in-memory workloads being widely used, such as in-memory cache [56] and in-memory database [57].

Even though the different demands for the resources in running datacenter workloads including memory intensive or latency-sensitive ones need to be considered, the current datacenter frameworks [21, 22] do not take them into account sufficiently. Traditional frameworks have focused mainly on the allocation of resources to the workloads; they have not considered the resource consuming patterns such as memory access or CPU interrupt handling. These metrics are also important compared to traditional ones including the allocated memory size, because they reflect the execution behavior of the workloads, and thus they can be directly associated with their performance. Using the metrics for scheduling workloads, the datacenters can handle the dynamic changes in the behavior of workloads. The key to Software-Defined Compute (SDC), which is one of the emerging trends for the datacenters, is to consider the dynamic behavior of workloads in performing the allocation of resources to the workloads.

SDC is one part of Software-Defined Data Center (SDDC), where all datacenter functions are controlled by the software controller [58]. It is originally from the concept of Software-Defined Network (SDN), which decouples the control and data planes. Similar to the SDN, the SDDC aims to reconfigure and reorganize the datacenter infrastructures by the software controller. Through the software-defined components such as the SDN and SDC, the datacenter can be more flexible and cost-effective. For example, if the SDC is realized, the datacenter infrastructures can be optimized, targeting the specific workloads easily, and flexibly reconfigured responding to changes in the behavior of workloads.

We introduce an effective workload-aware resource management framework for SDC. For the workload-aware resource management, we monitor the behavior of workloads in the datacenter and place or schedule the workloads based on the behavior, not based on the resource availability only. The workload profiler and workload-aware schedulers are the key components for the workload-aware resource management, which are used for profiling the workload characteristics and scheduling workloads to avoid performance interferences. In particular, we focus on the performance isolation with respect to the CPU and memory resources for the latency-sensitive workloads, most commonly used in the datacenter, when demonstrating the effectiveness of the framework. The workload profiler continuously monitors the resource usage pattern of latency-sensitive workloads, and if the performance anomaly is detected, then the workload-aware schedulers handle the performance problem by dynamically scheduling workloads or migrating the workloads to other hosts.

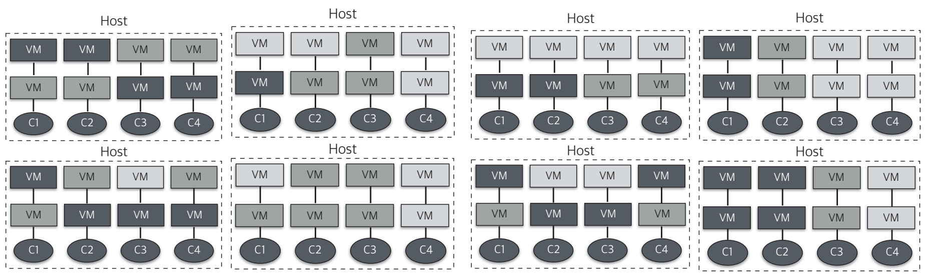
We implemented our schedulers in an OpenStack testbed and made it run in the VMWare one, and evaluated the scheduling algorithms in the OpenStack and VMWare testbeds. By performing experiments, we found that we can improve the performance of latency-sensitive workloads; specifically, we can achieve twice higher throughputs and lower the tail latency by up to 95% compared to the existing frameworks.

The rest of chapter is organized as follows. Section 5.2 shows the motivational scenario for contention-unaware scheduling. Section 5.3 explains the new performance metrics for profiling the behavior of workloads. Section 5.4 explains the methods for mitigating the contention for shared resources, describes workload-aware scheduling algorithms in the proposed framework, and provides the details of our scheduler implementation. Section 5.5 shows the results of our experiments performed to compare the proposed algorithms with the existing scheduling ones. Section 5.6 discusses the applicability of our proposed approach in practice. Section 5.7 presents the related work. Finally, Section 5.8 concludes the chapter.

5.2 Motivation

We consider the contention for memory in a motivating example. Suppose that VMs are placed on the hosts and have various resource demands as shown in Figure 5.1a and 5.1b. Even though there are various types of workloads in the datacenter, we assume only two types of workloads exist in the hosts in the simple scenario. In this case, some hosts may have lots of memory intensive VMs and others may not. If there are an equal number of VMs in each host, there may be no difference of resource utilization among the hosts, because the resources are allocated based on the types of VM instances. However, although the allocated resources are similar between two hosts, there may be a large difference in the performance of VMs; for example, the intensity of memory contention is different among the hosts due to the difference in the number of the simultaneous memory accesses. Currently the existing datacenter VM management is based on the strategy which maximizes the overall resource utilization of datacenter. Therefore, the current VM scheduling algorithms evenly distribute the VMs among the hosts in the datacenter. This strategy may be good for resource utilization, but it may not guarantee low latency or high throughput for the workloads.

Figure 5.1c and 5.1d show the different levels of VM memory intensity in a host depending on the VM-to-core mapping. Figure 5.1c shows that the memory intensive VMs spread across the cores in the host, and this placement can result in high memory intensity at once; in the worst case, all cores might handle memory requests at the same time, and thus the VMs accessing the memory subsystem in the same socket could suffer from performance degradation. However, if the VMs are placed and scheduled as shown in Figure 5.1d, then simultaneous memory requests could be reduced and thus the performance degradation would also be decreased. Furthermore, to avoid the performance degradation, it is essential to schedule workloads across the cluster considering the resource states of both hosts and cluster. Because the resource



(a) High Memory Intensity w/o Scheduling (b) Low Memory Intensity w/o Scheduling (c) High Memory Intensity w/ Non-Restricted Scheduling (d) Low Memory Intensity w/ Restricted Scheduling

Figure 5.1: Different Level of Memory Intensity (a-b), Different VM-to-Core Mapping (c-d). The darker colored is a VM, the higher memory intensive is the VM. The physical CPUs are depicted as the black circles. We assumed that each VM has only one vCPU to show the problem clearly.

demands in the hosts change so dynamically over time that load imbalance can easily occur.

5.3 Workload Profiling for Performance Isolation

This section presents how our proposed profiler works and which metric is used for profiling. We used the metrics to detect the contention for resources to mitigate the contention and those to present the behavior of latency-sensitive workloads in order to maximize the levels of resource utilization while meeting the SLOs.

5.3.1 Performance Metrics for Workloads Behavior

This section presents metric used by the workload-aware schedulers for estimating the memory contention. In addition to the memory contention metric, it also shows the metric that measures the the performance of latency-sensitive workloads. In the following subsections, we answer the following questions: “which metric is useful and effective for understanding the characteristics of workload such as the memory intensity or latency sensitivity?” and “how can the workload-aware schedulers use these metrics?”

Memory Intensity

The memory intensity is different from the memory utilization in that it reflects the memory access behavior of the workloads. For example, there may be workloads that have high memory utilization but low memory intensity and vice versa. Therefore, when balancing the load in the datacenter, it might be ineffective to consider only the memory utilization as a metric for load balancing, without taking the memory intensity into account. The memory contention is critical to not only the memory intensive VMs, but also other colocated VMs. High memory contention, caused by ineffective consolidation, could affect the other VMs which access the shared memory subsystem. For this reason, we first investigated that which metric is effective to identify the memory intensity of VMs.

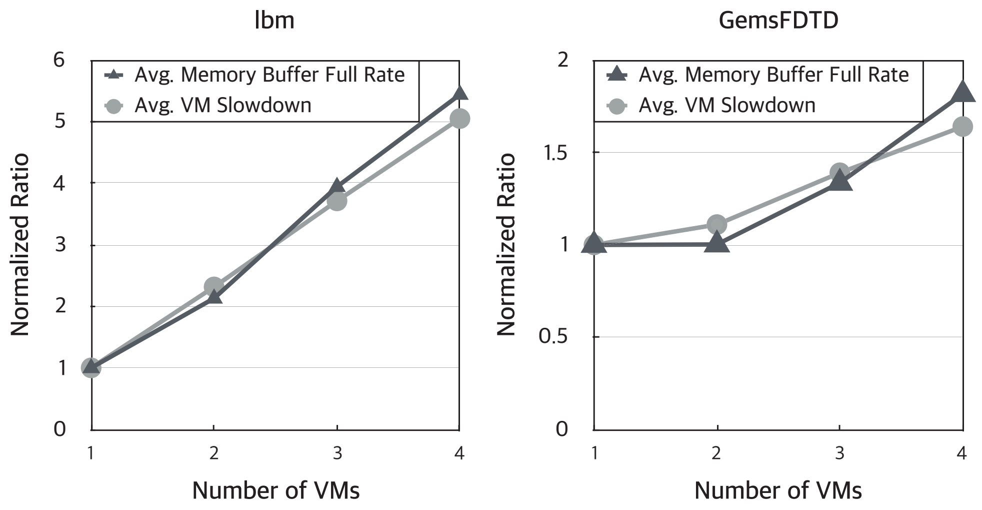
There are a lot of memory related components affecting the performance of workloads. However, among the various components such as L3 cache, prefetcher, and memory controller, there is no single dominant factor to influence the memory contention [10]. In the memory subsystem, there exist two parts; one part is called core part, which is dedicated to each core, such as L1 and L2 caches. The other part is called uncore part, which is shared across the cores. L3 cache and memory controller are in the uncore part. Because of the complexity of uncore part, we consider the entire uncore part as a blackbox. And we focus on the rejected requests to the uncore memory subsystem for deciding how much memory contention occurs. Between the L2 and L3 caches, there is an SQ (Super Queue) per core used for the buffer of uncore part [59].

We monitor the rejections for memory requests to the SQ. When a request is rejected because the SQ has become full, then it generates an SQfull event. We call the rate of the requests rejected from the SQ ‘Memory Buffer Full Rate’ or MBFR for short.

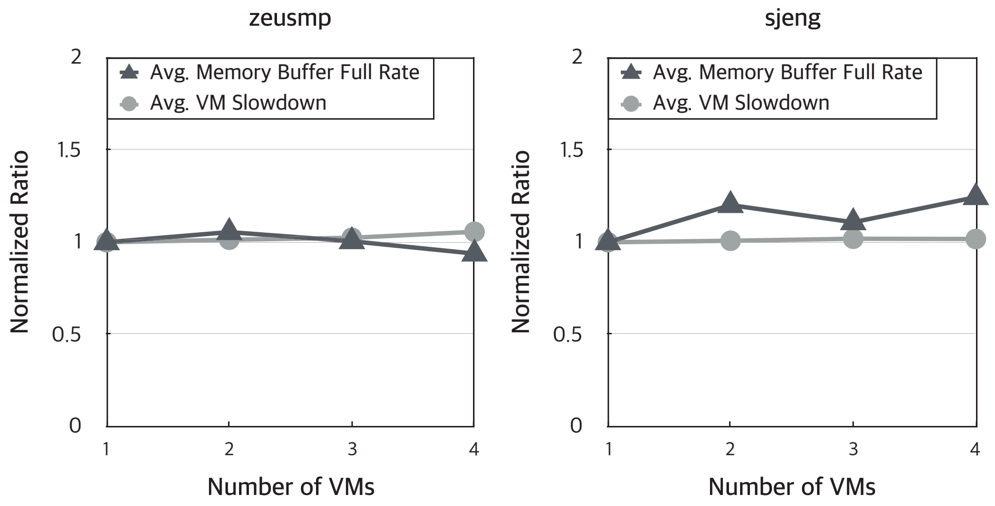
To check how MBFR reflects the memory intensity and affects the performance of VMs, we conducted the stress tests. We chose four workloads among the SPEC2006 benchmarks [60]. With two well-known memory intensive workloads, *lbm* and *GemsFDTD*, and other two well-known CPU intensive workloads, *zeusmp* and *sjeng*, we investigated the relationship between the MBFR and slowdown of the workloads while increasing the number of the same workloads in a host.

Figure 5.2 shows the experimental results for the four different workloads. We compared the execution time and MBFR with those of the solo execution. As shown in the figure, we can find a strong correlation between MBFR and the average VM slowdown. In the case of memory intensive VMs, the more VMs are colocated, the higher MBFR and slowdown happen. In contrast, in the case of CPU intensive VMs, no matter how many VMs are colocated, neither MBFR nor slowdown is changed much.

In order to confirm that the MBFR could be an appropriate metric which reflects the memory intensiveness, we compared it with the L3 miss rate, which might be regarded as a possible memory intensiveness metric to identify the memory intensity [10, 12]. Figure 5.3 shows the total L3 miss rate and total MBFR for each workload. As shown in the figure (*left*), we can figure out that some memory intensive workloads have low L3 miss rates which do not reflect their memory intensity. For example, *lbm* is known as a memory intensive workload and shows higher slowdown than the CPU intensive workloads as shown in the figure, but it has lower L3 miss rates than the other CPU intensive workloads, *zeusmp* and *sjeng*. However Figure 5.3 (*right*) shows that memory intensive workloads, *lbm* and *GemsFDTD*, have higher memory buffer full rates than the CPU intensive workloads. These results indicate that a single



(a) Memory Intensive Workloads



(b) CPU Intensive Workloads

Figure 5.2: Correlation between Average Normalized MBFR and Average Slowdown.

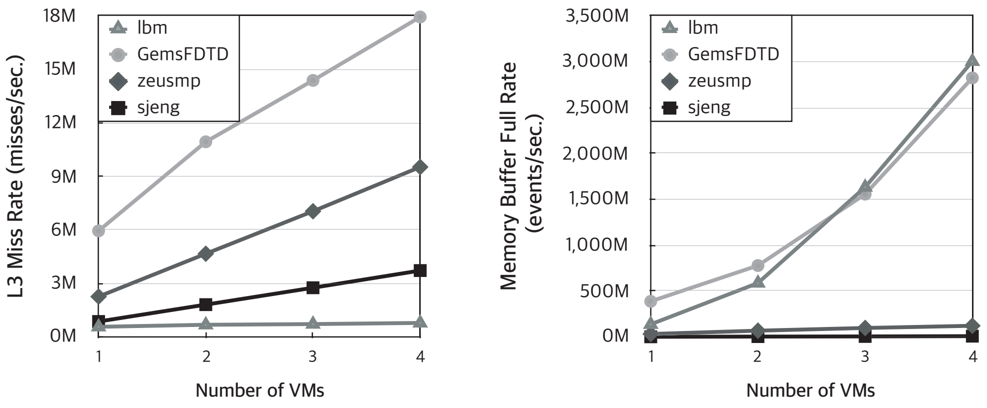


Figure 5.3: Comparison between L3 miss rate (left) and MBFR (right).

factor such as L3 miss rates may not show the memory contention in all cases. Via some stress tests and comparison experiments, we found that MBFR is an effective metric that shows the slowdown of VM.

SoftIRQs

To figure out the behavior of latency-sensitive workloads, we need to understand how the network I/O operations are performed. When the latency-sensitive workloads communicate with other clients, the network I/O occurs, and then the interrupts, as known as SoftIRQs (Software Interrupt Requests), are generated from the NIC of physical machines to process the incoming network packets. As shown in Figure 5.4, the number of queries processed per second (QPS) during the execution of memcached workload as the throughput, and that of generated SoftIRQs (y-axis) increase as the rate (x-axis) of the requests made by the client grows. Also we find that the more network traffic is incoming, the more packets are processed, and the more interrupts occur. Based on this result, we could conclude that there is a strong correlation among the network traffic, the number of interrupts, and the performance of latency-sensitive workloads.

Time Window for Collecting the Values of Metrics

To detect the change in the performance of workloads, we collect the MBFR and SoftIRQ samples for some intervals of time such as every 10 (for the local scheduler) and 30 seconds (for the global scheduler), considering these intervals as the time windows. The use of time window is necessary for estimating the slowdown of VMs or predicting the violation of SLOs. The local/global scheduler has its own scheduling interval, and the workload profiler monitors the consecutive samples for the intervals in order to provide the information to the schedulers. If the samples are collected, the information for scheduling is calculated from the collected samples and sent to the schedulers. For example, the averages of metric values are calculated and sent to the local schedulers in order to decide the memory intensities of the hosts. For global scheduling, the workload profilers send the samples of metric values collected every 30 seconds to the global scheduler, and the scheduler can decide which host has workloads causing SLO violations, and then perform migrations for the workloads.

Many other researchers recognized the importance of processing interrupts for the latency sensitive workloads, and thus they tried to improve the performance of interrupt processing [61–65]. Therefore, we decided to use the number of interrupts as an indicator to reflect the performance of the latency-sensitive workloads.

5.4 Workload-Aware Scheduling for Performance Isolation

Our approach is to solve the contention problem via VM scheduling. To highlight the effects of the workload-aware scheduling, we make some scenarios for placing and dynamically scheduling VMs.

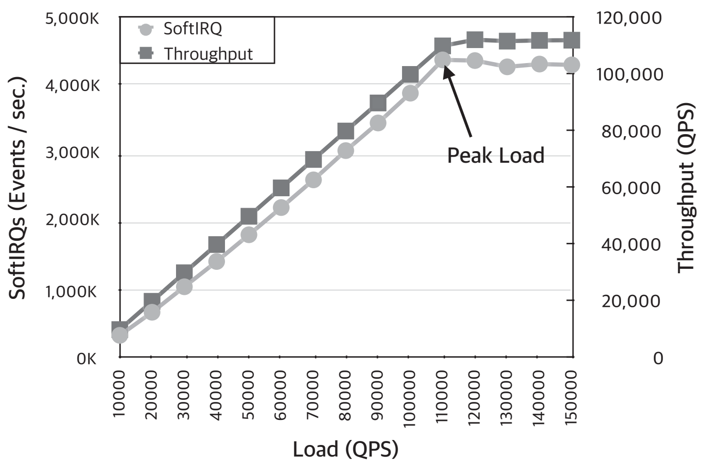


Figure 5.4: Relationship between SoftIRQs and Throughput. As the SoftIRQ is saturated, the throughput is also saturated. This indicates the SoftIRQ reflects the load in the queueing system.

5.4.1 Method for Mitigating Resource Contention

This subsection describes the method for mitigating the resource contention and thereby improving the performance. It presents the strategy of reducing the simultaneous memory accesses and also the method of predicting the latency for detecting the performance anomaly. To reduce the resource contention, we adjust the vCPU-to-core mapping and restrict the number of cores that memory intensive workloads can run on. This helps the performance of colocated workloads be not degraded. To meet the SLOs of latency-sensitive workloads, we make a prediction of latency by utilizing the information on the memory buffer full rates and software interrupts. We use a threshold to determine the violation of SLO, and thus if the predicted latency exceeds the threshold, the process of mitigating the contention is triggered.

Restricted Scheduling

It is important to minimize the memory contention for latency-sensitive workloads, especially in the case of in-memory latency-critical workloads such as in-memory cache services or in-memory database ones. These in-memory latency-critical services can be easily affected by the characteristics of co-located workloads with high memory intensity. Thus discreet scheduling is essential to meet the latency requirements of latency-sensitive workloads. To do this, it is desirable for these workloads to make dedicated access to the shared resources such as the cores and memory. Latency-sensitive workloads should be executed on the dedicated cores, because they are especially sensitive to the sharing of CPU resources [62]. To mitigate the access for the shared memory subsystem, it is necessary to restrict the number of memory intensive cores that have high memory intensiveness.

Without restricting the number of such cores, the memory intensity may be increased throughout all cores, resulting in high memory contention due to the contention for the limited resources of memory subsystem. Importantly, it is critical to schedule workloads considering the limitations of using the resources such as memory bandwidth. In this chapter, to show the effectiveness of our workload-aware resource management framework, we suggest a scheduling algorithm that dynamically restricts the maximum number of memory intensive cores, on which memory intensive workloads can run.

Predicted Latency

In cloud datacenters, latency-sensitive workloads such as web servers and in-memory database systems are common, and to these workloads, the tail-latency is considered as a critical performance metric. We suggest a simple, efficient model to predict the violation of SLO (Service Level Objective). To derive such a model, by using only server-side information, the trend of tail latency was predicted. We estimated the latency with some metrics by using a memcached that is well known as the representative latency-sensitive workload. The derived equation Eq.1 is as follows:

$$W = \frac{c \times MBFR_{vm}}{1 - R_{IRQ_{vm}}}, \tag{5.1}$$

where W is the average waiting time, $c \times MBFR_{vm}$ is the average service time of the VM (c is a constant), and $R_{IRQ_{vm}}$ is the utilization of queueing system s.t. $0 \leq R_{IRQ_{vm}} < 1$.

$$R_{IRQ_{vm}} = \frac{IRQ_{vm}}{IRQ_{max}} = \frac{IRQ_{vm}}{CPU_{vm}} \times k, \tag{5.2}$$

where $R_{IRQ_{vm}}$ is the ratio of SoftIRQs, IRQ_{vm} is the number of SoftIRQs, IRQ_{max} is the maximum number of SoftIRQs, and CPU_{vm} is the CPU utilization for the VM.

Eq. 5.2 was derived by using the Little's law [66] to predict latency. This law could be easily applied to any queueing system. To apply the law to our system, we assumed that our queueing system is based on M/M/1, which is commonly used for web server queueing model. In this M/M/1 model, W can be calculated with the average service time and utilization of queueing system. Considering that the MBFR approximates in the execution time of memory intensive workloads, MBFR can be substituted for the service time. From the ratio of SoftIRQ ($R_{IRQ_{vm}}$), we could calculate the utilization of queueing system as shown in Eq. refeq2. To obtain the maximum IRQ for the system, we simply used the fact that the more SoftIRQ is processed, the more CPU consumption occurs as previously shown in Figure 5.4.

5.4.2 Workload-Aware Scheduler

We introduce workload-aware VM schedulers that minimize the memory contention of latency-sensitive VMs while meeting the latency requirements for the VMs. Particularly, we have focused on in-memory latency-sensitive workloads such as memcached, the latency requirements for the VMs of which can be met by minimizing the memory contention of latency-sensitive VMs. The proposed schedulers perform the two-phase scheduling, which consists of the host-level and core-level one.

Global Scheduler

The objective of the global scheduler is to place and dynamically schedule VMs among the hosts by detecting the performance anomaly with the threshold for the LSVMs (Latency-Sensitive VMs) and meet the performance requirements for these VMs while minimizing the resource contention. First, the global scheduler should predict the latencies of LSVMs, and thus it receives information on the MBFR and SoftIRQ from the profiler.

The MBFR is used to measure the memory intensity, and the SoftIRQ is also utilized to predict the performance anomaly for the LSVMs. These measurements are used to check whether migration of VMs will be beneficial to the LSVMs, every 30 seconds (**Line 4**). We set the interval of global scheduler to 30 seconds, because we emprically found that the interval is adequate for the global scheduler to collect the performance information about the hosts. The global scheduler periodically checks the predicted latency of each LSVM and when it exceeds the threshold, `SLO_threshold`, then the scheduler starts the live migration procedure (**Lines 7-18**). We think that the use of the threshold should lead to meeting the performance requirement for a latency-sensitive workload as if it ran solely. Specifically, in the case of the memcached workloads, we determined the threshold to the predicted latency when a load of about 75 percentile of peak throughput was given, which kept it below sub-10milliseconds latency. If there are other latency-sensitive workloads, multiple `SLO_threshold`s could be used, and the `SLO_threshold` of each latency-sensitive workload should be determined as in the case of memcached workload which we used. If the live migration procedure is triggered, then the global scheduler finds

Algorithm 5.1: Global Scheduler

```
   /* Global scheduler is invoked every 30 seconds                  */
1  def GlobalScheduler():
2      for each (current) host:
3          collected_stats= collect_metrics();
           /* Collecting info on VMs and hosts from the profiler  */
4          for each lsvm in current host:
5              p = predicted_lsvm_latency(collected_stats);
6              if p > SLO_threshold:
7                  h = lowest_cont_host(collected_stats);
8                  if h is not current host:
                       /* If current (source) host is not the one with
                          the lowest memory intensity, migrating LSVM
                          to the least memory intensive host          */
9                      target_vm = lsvm;
10                     dest = h;
11                 elif h is current host:
                       /* If current (source) host is the lowest
                          memory intensive host, migrating the highest
                          memory intensive Non-LSVM to the lowest
                          memory intensive host other than the source
                          one                                         */
12                     target_vm = high_cont_nlsvm(collected_stats);
13                     dest = other_lowest_cont_host(h);
14                 live_migration(target_vm, dest);
```

the destination to migrate the LSVM to, based on the MBFR; the scheduler searches the lowest memory intensive host among all hosts except for the source host. If it succeeds in finding the lowest memory intensive host, then it migrates the LSVM to the destination host (**Lines 10-13**); it migrates a non LSVM that has the highest memory intensity to the second lowest memory intensive host (**Lines 14-17**).

Local Scheduler

The goal of local scheduler is to maximize the performance of latency-sensitive workloads by running them on the dedicated cores while executing the memory intensive VMs on a limited set of other cores. Every 15 seconds, the local scheduling procedure is triggered. We set the interval of local scheduling to 15 seconds, because we empirically found that information on about 15 to 20 VMs was collected in 10 to 12 seconds. Once the scheduler is invoked, it classifies the vCPUs in the host into three groups, which are latency-sensitive vCPUs (ls), memory intensive vCPUs (mem), and non-memory intensive vCPUs (nmem) (**Line 4**). After classifying the vCPUs, the local scheduler decides to allocate the dedicated cores to latency-sensitive vCPUs by pinning each vCPU to a certain dedicated core in order to meet their requirement of the low latency (**Line 6**). We allocate each dedicated core to each latency-sensitive vCPU, and have the other vCPUs (mem and nmem) be pinned to other non-dedicated cores. It is thus possible to allocate the dedicated cores to the latency-sensitive workloads. It is important to divide the resources into two groups, the dedicated and shared ones, for high performance of latency-critical workloads and for overall high server utilization.

To schedule the workloads other than the latency-sensitive ones, the scheduler decides the number of memory intensive cores for throttling the concurrent memory accesses (**Line 8**).

The scheduler can determine the number dynamically based on the memory intensity of the current host. Considering the fact that the MBFR of host increases quadratically as shown in Figure 5.3 (*right*), we empirically found that the MBFR value of host increases too rapidly, as memory intensive VMs run on more than approximately a half of cores in the host. Therefore we set the maximum number of memory intensive cores to the half of memory intensive workloads in the host. The local scheduler schedules memory intensive workloads on the limited set of memory intensive cores while it distributes the unscheduled non-memory intensive workloads on the rest of the shared cores. When the scheduler allocates the cores to the vCPUs, it considers the number of vCPUs running on each of the cores, which may be regarded as a kind of fairness, so that it can maximize the core utilization in the host (**Lines 12, 15 & 17**).

To maximize the utilization, the local scheduler finds which core is the one, on which the smaller number of vCPUs scheduled. For example, suppose that a vCPU is in the set of unscheduled, and that core 0 and core 1 are the candidate destination cores, which have one vCPU and three vCPUs, respectively. The local scheduler calculates the differences in the number of vCPUs between the cores in the `get_min_vcpu_core()` function. If the difference is 0, then the vCPU is scheduled on the higher memory intensive core; otherwise, this case means the imbalance between the cores occurs, and thus the vCPU is scheduled on the less loaded core.

Once the destination core is determined, the CPU affinity of the vCPU is decided to the destination core in the `schedule()` function. If the destination core is previously allocated to the vCPU, the function does not do anything in order to avoid unnecessary overhead (**Line 18**).

Implementation

We made a prototype of workload-aware schedulers with Python. The prototype is composed of three parts, the global scheduler, the local scheduler, and workload profiler, which are implemented as user-level programs to run without any modification of host and guest OSes. The global scheduler makes RPC calls in order to receive the

Algorithm 5.2: Local Scheduler

```
/* Local scheduler is invoked every 15 seconds */
def LocalScheduler():
    collected_stats = collect_metrics();
    mem, nmem, ls= classify_vcpus(collected_stats);
    candidate_vcpus = mem + nmem + ls;
    dedicated_cores = get_dedicated_cores(len(ls));
    unscheduled = len(candidate_vcpus);
    mem_cores = get_mem_cores(collected_stats);
    while unscheduled > 0:
        src_vcpu = get_src_vcpus(candidate_vcpus);
        if src_vcpu is latency-sensitive vcpu:
            dst = get_min_vcpu_core(dedicated_cores);
        elif src_vcpu is non-latency-sensitive vcpu:
            if src_vcpu is memory intensive vcpu:
                dst = get_min_vcpu_core(mem_cores);
            elif src_vcpu is non-memory intensive vcpu:
                dst = get_min_vcpu_core(other_cores);
        schedule(src_vcpu, dst);
        unscheduled = unscheduled - 1;
        candidate_vcpus.remove(src_vcpu);
```

values of the metrics about the host from the profiler that resides in each host. The global and local schedulers are implemented on the OpenStack Nova and KVM, and they schedules the VMs by using the Nova and Libvirt API. The workload profiler uses Linux Perf [67] to collect the values of the MBFR and SoftIRQs. To obtain these values, the profiler reads the PMU counter every second. For detecting LSVMs, we use the tracepoint events provided by the Perf. We utilize the MetricWeigher which is a part of the OpenStack Filter scheduler to receive the values of metrics, and decide whether the host is memory intensive or not.

5.5 Evaluation

To illustrate the benefit of our workload-aware scheduling, we show the result of executing the scheduling algorithms step by step. We present the performance improvements from initial placement of workloads to dynamic scheduling of them in the OpenStack and VMWare testbeds, respectively. We show how beneficial the proposed workload-aware scheduling could be to the latency-sensitive workload by considering the memory intensity and latency sensitiveness. We first show the benefit of local placement which assigns all the workloads (including the non-latency-sensitive ones) to the cores by considering the memory intensity of host, and then show the benefit of global placement which assigns the workloads to the hosts and subsequently the cores of the selected hosts by considering the difference in the memory intensity between the selected host and the other ones. Finally, we present how the proposed approach works well even in a workload-changing scenario where the memory intensity of host dynamically changes.

To create a high contention environment, we ran some SPEC2006 workloads as interfering ones. We chose the *lbm* as a memory intensive one and *sjeng* as a CPU intensive one, because the memory intensities of these two workloads are almost constant. We also used the memcached workload as the latency-sensitive one, which is one of the most popular in-memory cache services for the Web. Because the memcached should respond quickly to the requests from web services, the tail-latency is important during the execution of memcached.

We show benefit of our proposed mechanisms by applying them in each of the testbeds, and comparing them with each of the existing scheduling ones; OpenStack using KVM on top of Linux OS, VMWare vSphere uses ESXi that is the bare-metal hypervisor. To show the benefit of using our algorithms in the VMWare testbed, we manually performed live migration by setting the CPU affinity in the UMA architecture according to our algorithm.

5.5.1 Experimental Setup

We performed experiments in the OpenStack and VMWare vSphere testbeds. Each testbed consists of four compute hosts, running twelve VMs, and one controller host providing services such as networking and managing the compute hosts. Each host in each testbed has an eight-core Intel Xeon CPU E5-2650 v2 @ 2.6GHz with both Hyper-Threading and DVFS disabled. In each host, the CPU frequency is 2.6GHz and the size of main memory is 256GB. Each server is connected though 1Gbps links. For dynamic live migration, we used the Ceph filesystem as the OpenStack volume backend, and vSphere's VMFS filesystem for vMotion.

We considered the memory contention on the UMA system. To differentiate the memory intensity of each host in a scenario, we ran the memory and CPU intensive VMs at the different ratios as shown in Table 5.1. We chose the 'mutilate' benchmark to generate realistic memcached workloads [68]. Through the mutilate, we simulated Facebook 'ETC' workloads which are the representative Facebook ones with the lowest hit rate (81.4%) [69]. To generate the workloads, five clients were used, and each client made the requests by using sixteen threads on the sixteen-core client host. We

Table 5.1: Different Configurations for Experiments. We used the SPEC 2006 benchmarks, *sjeng* as a CPU-intesive workload and *lbm* as a memory intensive workload.

Host	H1	H2	H3	H4
CPU-int. VMs	4	6	8	12
Mem-int. VMs	8	6	4	0

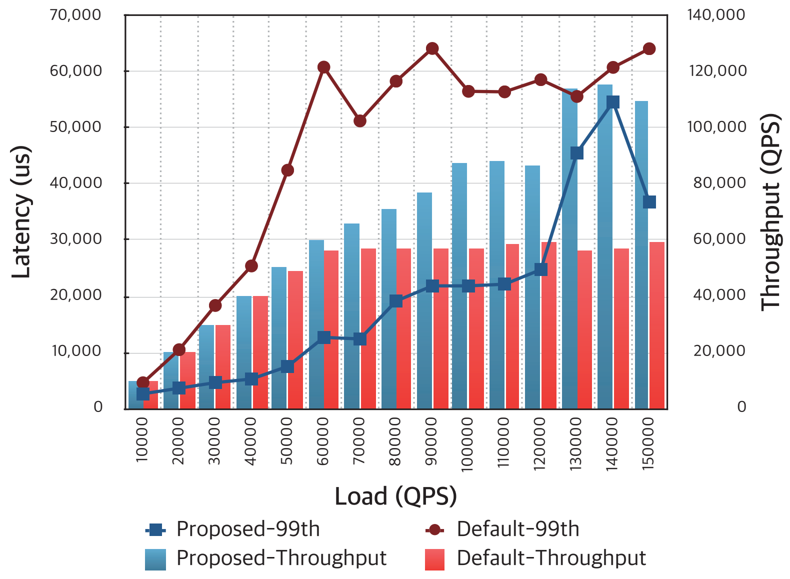
ran the memcached server in the VM instance with two vCPUs & 6G RAM, and two memcached server threads in total.

5.5.2 Experiment Results

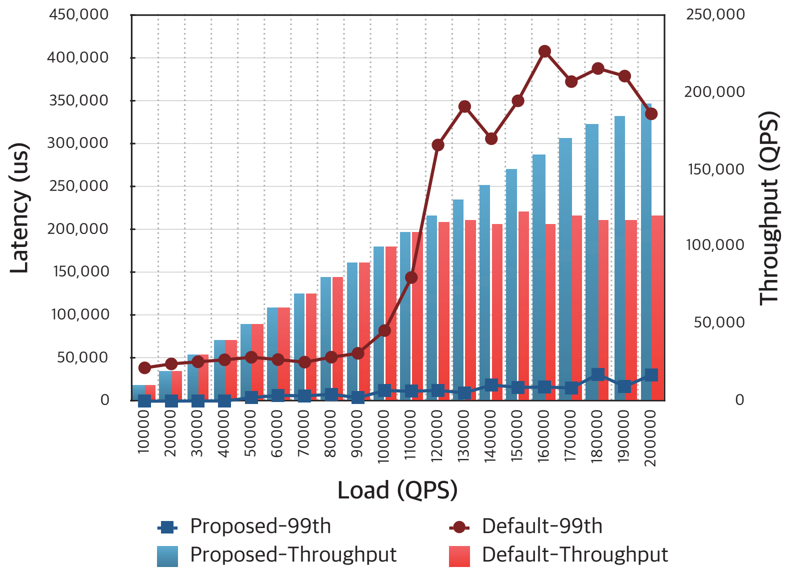
When creating a VM, its performance depends on the policy of scheduling it. The existing VM scheduling algorithms used in cloud platforms such as OpenStack and VMWare are based on the resource availability or resource entitlement. These approaches are resource efficient because it maximizes the cluster utilization, making the utilization of each host uniform. However, they are not the best in terms of the performance of VMs due to not considering workload patterns or characteristics. Also they decide which hosts to run the VMs on, but not which cores to run them on, and thus the contention for the shared resources, caused by colocation of workloads may make their performance degraded and unpredictable.

Local Placement

To show the effect of local placement of VMs in a host, we ran the thirteen VMs (one Memcached VM, six CPU intensive VMs, and six memory intensive VMs) in a host. Figure 5.5 shows the performance improvements for different placements of VMs on each testbed; the higher loads on the memcached VM is, the lower latency and higher throughput are led to by using the proposed approach.



(a) OpenStack



(b) VMWare vSphere

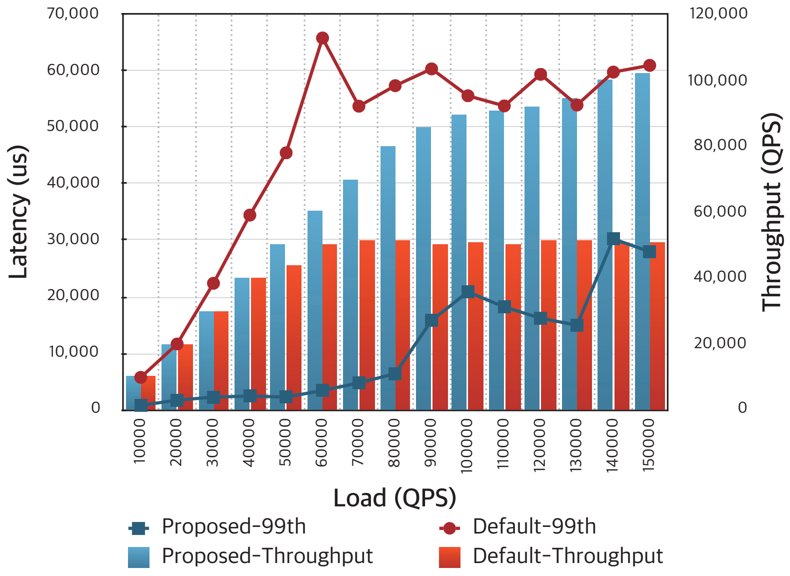
Figure 5.5: Comparison for Local Placement

The current existing approaches do not determine which workloads to schedule on specific cores, leading to high CPU and memory contention. As shown in Figure 5.5, our proposed approach improves the maximum throughput 1.4 times and reduces the latency by 58% when applied to OpenStack, and improves the maximum throughput 1.6 times and reduces the latency by 95% when applied to VMWare vSphere.

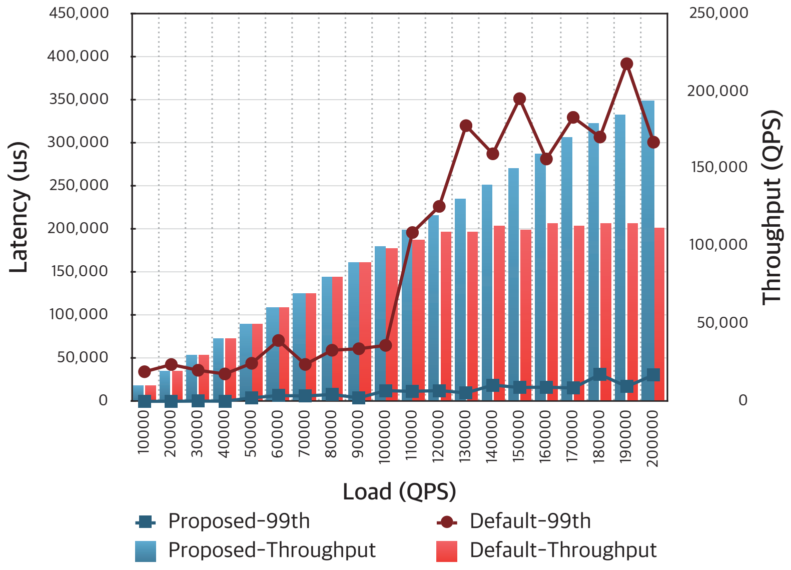
Global Placement

Through the global placement experiments, we examined how and where the memcached VM is scheduled with different scheduling algorithms, and then measured its performance. When the VM was provisioned, the schedulers in OpenStack and vSphere could not schedule the VM in the best place in terms of the memory intensity. We performed experiments for the VM provisioning three times in each case. In the case of OpenStack, although the available RAM size was equal among the hosts, the VM was always scheduled on host 1. The Filter scheduler selected the host based on the available RAM size, but since all hosts had the equal size of available RAM, the scheduler randomly selected a host among all hosts. Tables 5.2 and 5.3 show the resource usage in the OpenStack and VMWare testbeds, respectively.

Since the available RAM was equal, one host had the same available memory compared to the other hosts. In the OpenStack testbed, most memory intensive workloads were thus scheduled all across the hosts, and the memcached VM suffered from the higher contention of the memory. In the case of VMWare vSphere, the VM was scheduled on host 1 twice, while the VM was scheduled on host 3 once. Because the DRS algorithm is based on the host resource entitlement, which is the resource usage with respect to the resource capacity. Given the resource capacities, the capacity of memory was so larger than that of CPU that CPU was the most influential factor to the host load metric. As a result, it did not consider the memory intensity, and thus the VMs were scheduled regardless of the memory intensity, degrading the performance of the VMs.

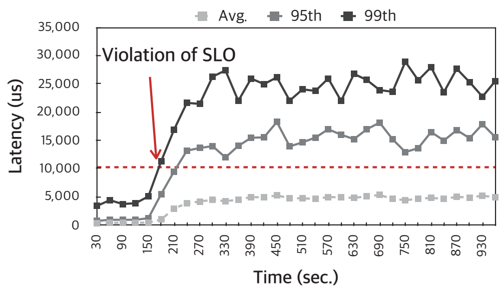


(a) OpenStack

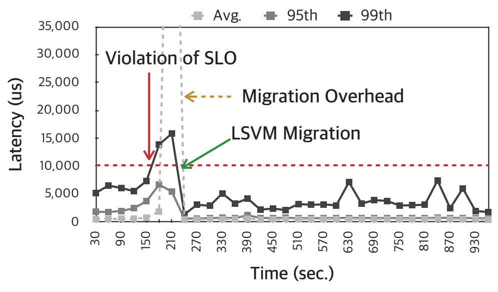


(b) VMWare vSphere

Figure 5.6: Comparison for Global Placement



(a) OpenStack Filter Scheduler (Default)

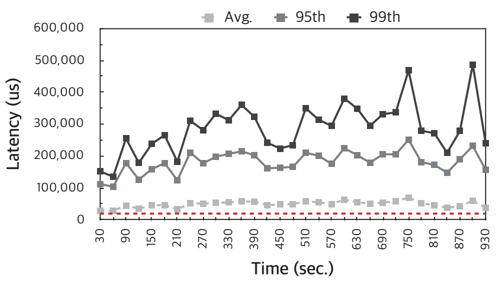


(b) Proposed Approach

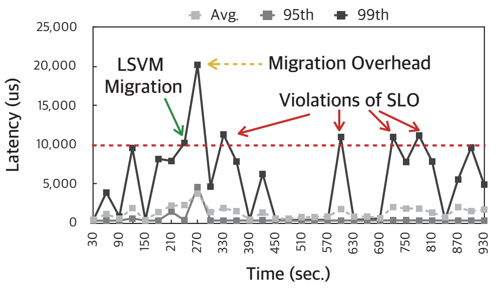
Figure 5.7: Dynamic scheduling (OpenStack). The existing OpenStack (a) could not handle the dynamic changes in memory intensity. However, our proposed approach (b) detected the violation of SLO and migrated the LSVM to the lowest memory intensive host other than the current one.

Table 5.2: Resource Usage in the OpenStack Testbed. Since the same number of VMs ran in each host, the memory of the same size was allocated to each of the VMs. However the memory intensity of each host was different, and consequently the Filter scheduler could not reflect the memory intensity.

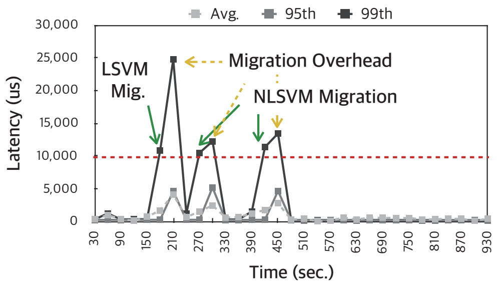
Host	H1	H2	H3	H4
Available RAM(MB)	208,708	208,708	208,708	208,708



(a) VMWare DRS (Default)



(b) LSVM Only Migration Approach



(c) Proposed Approach

Figure 5.8: Dynamic scheduling (VMWare vSphere). The existing DRS (a) dispatched the LSVM on the memory intensive host, which is inappropriate because of not considering the memory contention. The approach of migrating only the LSVM (b) led to the violations of LSVM's SLO for a longer period of time. However our approach of migrating Non-LSVMs as well (c) led to the violations for a shorter period of time.

Table 5.3: Resource Usage in the vSphere Testbed. The bold numbers indicate why the CPU resource was considered as the more important one, and thus DRS put an emphasis on the CPU resource rather than the memory resource or intensity.

Host	H1	H2	H3	H4
CPU Usage (GHz)	31.82	33.30	32.90	33.07
CPU Capacity (GHz)	38.38	38.38	38.38	38.38
CPU Usage/Cap.	**0.83**	**0.87**	**0.86**	**0.86**
Mem Usage (GB)	19.13	17.23	14.56	12.14
Mem Capacity (GB)	255.96	255.96	255.96	255.96
Mem Usage/Cap.	**0.08**	**0.07**	**0.06**	**0.05**

In contrast, when our proposed approach was applied to the OpenStack and VMWare testbeds, the scheduler always scheduled the memcached VM on host 4 that had the lowest memory intensity, and the colocated workloads on the cores, which minimized the contention for the shared resources. As shown in Figure 5.6, we could obtain the performance improvements in terms of both the throughput and tail latency. Although there was some degradation of performance in terms of latency in the high load compared with the default, it may be due to the trade-off between the throughput and latency because the throughput is twice as high.

Dynamic scheduling

Cloud datacenters are so dynamic that we should consider the case where the behavior of workload changes over time. We performed experiments in such dynamic scenarios in order to check how much performance could be improved with different dynamic scheduling algorithms. We used the initial placement of global placement experiments previously conducted. We conducted the experiment where CPU intensive workload changed to memory intensive one every 30 second after the memcached VM was placed initially. As shown in Figure 5.7, the existing default OpenStack was not able to schedule the VM automatically via live migration. That is, it could not deal with workload changes properly, leading to increases in the tail latency. At this moment, the memcached VM violated the SLO, but stayed in the memory intensive host leading to performance degradation.

However, when our proposed approach was applied, the scheduler predicted the violation of the SLO (10ms) and lively migrated the memcached VM to the lowest memory intensive host other than the current one. Although the tail latency could be increased temporally, because of the migration overhead, the SLO of memcached VM was met. Figure 5.8 shows the results of VMWare vSphere's dynamic scheduling in the three cases, which are DRS scheduling (default), LSVM migration approach, and all VMs (both LSVM and Non-LSVM) migration one (our proposed approach). In the case of scheduling with DRS, the tail latency of the memcached VM was exacerbated, but DRS did not migrate the VM. Since DRS scheduled VMs based on the resource entitlement, the load imbalance standard deviation (0.021) for all hosts did not exceed the migration threshold (0.141), and thus the migration process was not triggered. In the approach of LSVM only migration and core mapping with our local algorithm, the schedulers migrated the LSVM. The approach shows the stable tail-latency in the OpenStack testbed (Figure 5.7b). Unlike this case, sometimes the approach shows the unpredictable tail latency as in the vSphere (Figure 5.8b), leading to the violation of the SLO. As in this case, the approach of dynamically scheduling LSVM, performing the migration of LSVM might fail to meet its SLO. In this case, we should migrate the other VMs, which are Non-LSVMs, to the lowest memory intensive hosts other than their current ones. As shown in Figure 5.8c, even after the LSVM migration, if the SLO of LSVM was violated, then scheduler migrated the Non-LSVMs until the SLO was satisfied. Consequently, the use of our proposed approach met the SLO.

5.6 Applicability

It would potentially be much beneficial to apply our proposed approach in real-world applications. For example, the approach can be adopted to the public clouds such as Amazon EC2 and Google Compute Engine. They provide lots of memory intensive services such as in-memory databases. These services are memory intensive and latency sensitive as well, to which it could be beneficial to apply the approach. Other examples are the entertainment services such as gaming. The gaming service is a latency-critical one requiring low latency and lots of memory for storing temporal data. The approach could be applicable and effective due to its nature of dynamical change. In fact, it could be useful for any in-memory latency sensitive workloads.

5.7 Related Work

Our research is related to profiling resource contention. There have been some studies, and one of the approaches is measuring the workload sensitivity by giving pressure to a shared resource [12, 18, 70]. The benefit of this approach is to estimate the allowable limit of the contention for the shared resource, which affects the performance of workloads regardless of the server microarchitecture. However, the approach has a defect that taking an offline profiling should be preceded to estimate workload sensitivity. There is another approach that utilizes metrics to decide the state of shared resources. In this work, we take this approach and it is important how strongly the values of the metrics as the result of workload profiling are correlated to the characteristics. By suggesting a new performance metric that reflects the memory intensity, we can improve the weakness of using the L3 miss rate, which has been dominantly used to characterize the memory intensity of workloads, but has turned out to be only partially useful. There have been a number of attempts to detect the performance anomaly, using various performance metrics. Google uses the CPI (Cycle Per Instruction) information for classifying the abnormal behavior of workloads [7]. This approach finds the interfering workloads by using the correlation between the victim workload's CPI and interfering workloads' CPU utilization. This approach is similar to ours, in that it restricts the activity of interfering workloads, but they estimated the behavior of workloads by using the CPI, which can be influenced from various types of interfering sources, thus it has the limitation of identifying and recovering the contention for the shared resources. Monasca is one of the recent projects on the OpenStack [71]. In Monasca, they try to perform the analytics for detecting the performance anomaly by utilizing machine learning algorithms, which is currently under continuous development.

Our research is also related to scheduling of datacenter workloads. Recently, there have been some studies, considering a datacenter as a big logical computer and managing its resources. Mesos is one of the representative projects that views a datacenter cluster as a resource pool to maximize resource utilization by sharing clusters among distributed workloads [72]. However, Mesos targets the distributed workloads such as Spark [73], and our performance isolation techniques could be integrated into Mesos as well. And there have some projects on scheduling workloads at a datacenter, which considers its heterogeneity and resource contention, such as Paragon [74] and Quasar [11]. They use the history of profiled data and they also need some hosts for profiling, and thus their approaches require additional data or infrastructure for applying their resource management framework. However, our workload-aware resource management framework does not require any offline obtained data or any hosts for offline profiling. Some researchers have also developed resource management frameworks such as Kubernetes [75] for container and Mercury [76] for the resource management of big data workloads. Kubernetes has currently been developed actively by Google, and our workload-aware resource management techniques can be adopted for further improving performance. Mercury works on top of YARN [77], and uses the dynamic scheduling algorithms based on the policies, which are *guaranteed* and *queue-*

able ones. However, it focuses on efficient scheduling of many short-lived distributed jobs, and thus it is orthogonal to our workload-aware resource management framework. In this work, we present our mechanisms of local scheduling (host scheduling) and global scheduling (intra-cluster scheduling) in order to demonstrate the effectiveness of our workload-aware resource management framework. We plan to perform research on the cluster-level scheduling such as like inter-cluster scheduling or inter-zone scheduling. We also plan to perform research on the policy-based scheduling with more complicated workloads such as distributed analytic and multi-tier web platform ones. These are important research subjects regarding datacenters and the demand for research on these subject is increasing continuously.

Improving the QoS of latency-sensitive workloads is crucial to the users at a datacenter. To enhance the QoS of the workloads, there have been many attempts such as Bubble-flux [78], Heracles [25]. Similar to our research, these studies have been conducted to predict and/or monitor the latencies in order to meet the SLO. By enhancing our metric, we can make better and more effective algorithm for predicting tail-latencies. When improving the performance of latency-sensitive workloads, it is problematic to consolidate latency-sensitive workloads [23]. Recent studies have just focused on the colocation of latency-sensitive workloads represented as long-running production workloads and best-effort batch workloads such as distributed data intensive workloads. However, most workloads in public cloud datacenters might be latency sensitive ones, and the colocation of them could lead to a critical problem.

5.8 Conclusion

We have proposed a workload-aware resource management framework, which can lead to performance improvements in the latency and throughput of the target workloads. By using our proposed framework and scheduling algorithm, we could minimize the performance interferences with latency-sensitive workloads, known as representative datacenter ones. To mitigate the resource contention and understand the behavior of workloads, we have developed effective performance metrics to reflect memory intensity and the performance of latency-sensitive workloads. Base on these metrics, we have developed the workload-aware scheduling algorithms that minimize the performance interferences, thereby letting both host scheduling and inter-host scheduling minimize the shared resource contention cooperatively. To demonstrate the effectiveness of this framework, we have developed local (host) and global (inter-host) schedulers in our framework in the OpenStack and VMWare vSphere testbeds, which are most popular cloud platforms, and found the use of the framework can lead to significant performance improvements compared with the existing scheduling algorithms. We plan to extend our algorithm to execute on bigger and more complex systems such as NUMA ones and also with other diverse emerging workloads.

Chapter 6

Conclusion

Performance isolation is becoming more important than before. However, many operating systems and frameworks provide just only contention-unaware and fairness-centric resource allocation, which leads to significant performance degradation, yielding resource inefficiency and violation of service level objectives of workloads. We have found three challenges for effective performance isolation and addressed them by utilizing isolation techniques adaptively, hierarchically, and in hybrid manner.

In Chapter 3, we developed a hybrid isolation system that utilizes hardware and software isolation techniques in a hybrid manner by the characteristics of the workloads. We have explored the tradeoffs between hardware and software isolation techniques, and illustrated how these properties affect performance of consolidated workloads. We have proposed an algorithm for isolation to use isolation techniques mutually complementary through characteristics analysis of workloads and comparison of each isolation technique. Our experimental results show that our approach can improves tail-latency from $1.4\times - 76\times$ compared to the-state-of-art isolation framework while improving resource efficiency under the different levels of contentions. improves tail-latency from 1.4×−76× compared to the-state-of-art isolation framework while improving resource efficiency under the different levels of contentions.

In Chapter 4, we have illustrated the challenges of multitasking on edge devices, which is critical to the performance of tasks. To address these challenges, we present *EdgeIso*, an effective edge scheduler that mitigates resource contention on edges and isolates the performance of latency-critical tasks while running background tasks. It periodically profiles which resource contention is the dominant one for the tasks running on the edge, tracks the changes in the contentions efficiently, and performs isolations adaptively and dynamically by enforcing the appropriate isolation techniques incrementally. We have evaluated *EdgeIso* on an NVIDIA Jetson TX2 using several benchmarks of diverse SLOs and resource demands. The evaluation results show that *EdgeIso* can effectively mitigate resource contention, reduce the SLO violation ratio significantly for tasks of diverse SLOs compared with existing schemes under dynamic resource contention.

In Chapter 5, We have proposed a workload-aware resource management framework, which can lead to performance improvements in the latency and throughput of the target workloads. To mitigate the resource contention and understand the behavior of workloads, we have defined effective performance metrics to reflect memory intensity and the performance of latency-sensitive workloads. Base on these metrics, we have developed the workload-aware scheduling algorithms that minimize the performance interferences, thereby letting both host scheduling and inter-host scheduling minimize the shared resource contention cooperatively. To demonstrate the effectiveness of this framework, we have evaluated our algorithms on the OpenStack and VMWare vSphere testbeds. Evaluation results show significant performance improvements compared with the existing scheduling algorithms.

In the future work, we will investigate isolation techniques for the different micro-architectures such as AMD and ARM to generalize our ideas. we will also extend various edge server which have more resources than Jetson TX2, such as autonomous drones or cars. Finally, we will extend our hierarchical performance isolation scheme for workloads spanning multiple compute servers for large-scale workloads.

Bibliography

[1] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “Ssd: Single shot multibox detector,” in *European conference on computer vision (ECCV), 2016*.

[2] J. McCalpin, “Stream: Sustainable memory bandwidth in high performance computers,” *http://www. cs. virginia. edu/stream/*, 2006.

[3] C. Intel, “Improving real-time performance by utilizing cache allocation technology,” *Intel Corporation, April*, 2015.

[4] R. J. Wysocki, *CPU Performance Scaling*. Intel Corporation, 2017. `https://www.kernel.org/doc/html/v4.12/_sources/admin-guide/pm/cpufreq.rst.txt`.

[5] P. Turner, B. B. Rao, and N. Rao, “Cpu bandwidth control for cfs,” in *Linux Symposium*, vol. 10, pp. 245–254, Citeseer, 2010.

[6] S. Derr, *Control Group Cpusets*. BULL SA, 2004. `https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt`.

[7] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi 2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 379–391, 2013.

[8] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, pp. 55–64, 2013.

[9] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, *et al.*, "Perfiso: performance isolation for commercial latency-sensitive services," in *2018 USENIX Annual Technical Conference (ATC 2018)*, pp. 519–532, 2018.

[10] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 129–142, ACM, 2010.

[11] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[12] S. Kim, H. Eom, and H. Y. Yeom, "Virtual machine consolidation based on interference modeling," *the journal of Supercomputing*, vol. 66, no. 3, pp. 1489–1506, 2013.

[13] D. Seo, H. Eom, and H. Y. Yeom, "Mlb: A memory-aware load balancing method for mitigating memory contention," in *Conference on Timely Results in Operating Systems (TRIOS 2014)*, 2014.

[14] B. Teabe, A. Tchana, and D. Hagimont, “Application-specific quantum for multi-core platform scheduler,” in *Proceedings of the Eleventh European Conference on Computer Systems*, p. 3, 2016.

[15] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, “Rapl: memory power estimation and capping,” in *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pp. 189–194, ACM, 2010.

[16] 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 *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA 2015)*, pp. 271–282, 2015.

[17] H. Zhu and M. Erez, “Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 33–47, 2016.

[18] C. Delimitrou and C. Kozyrakis, “ibench: Quantifying interference for datacenter applications,” in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pp. 23–33, IEEE, 2013.

[19] “Nvidia jetson xavier,” 2019.

[20] “Intel vision accelerator design,” 2019.

[21] “Openstack.” https://www.openstack.org.

[22] A. Gulati, G. Shanmuganathan, A. M. Holler, and I. Ahmad, “Cloud scale resource management: Challenges and techniques.,” *HotCloud*, vol. 11, pp. 3:1–3:6, 2011.

[23] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*, p. 4, ACM, 2014.

[24] G. Linden, "Make data useful," 2006.

[25] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 450–462, ACM, 2015.

[26] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 107–120, 2019.

[27] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, 2008.

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC 2009)*, pp. 44–54, 2009.

[29] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[30] Y. Zhao, J. Rao, and Q. Yi, "Characterizing and optimizing the performance of multithreaded programs under interference," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pp. 287–297, 2016.

[31] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on numa systems: Asymmetry matters.," in *2015 USENIX Annual Technical Conference (ATC 2015)*, pp. 277–289, 2015.

[32] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC 2016)*, pp. 1–10, 2016.

[33] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, *et al.*, "Machine learning at facebook: Understanding inference at the edge,"

[34] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, *et al.*, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *ACM/IEEE SEC 2017*.

[35] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman, "Farmbeats: An iot platform for data-driven agriculture," in *USENIX NSDI, 2017*.

[36] S. Yang, "Iot stream processing and analytics in the fog," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 21–27, 2017.

[37] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan, "Bandwidth-efficient live video analytics for drones via edge computing," in *2018 IEEE/ACM Symposium on Edge Computing (SEC 2018)*.

[38] Y. Xiao and C. Zhu, "Vehicular fog computing: Vision and challenges," in *IEEE PerCom, 2017*.

[39] C. Zhu, J. Tao, G. Pastor, Y. Xiao, Y. Ji, Q. Zhou, Y. Li, and A. Ylä-Jääski, "Folo: Latency and quality optimized task allocation in vehicular fog computing," *IEEE Internet of Things Journal*, 2018.

[40] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[41] N. Talagala, S. Sundararaman, V. Sridhar, D. Arteaga, Q. Luo, S. Subramanian, S. Ghanta, L. Khermosh, and D. Roselli, "Eco: Harmonizing edge and cloud with ml/dl orchestration," in *USENIX HotEdge, 2018*.

[42] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *IEEE RTSS, 2017*.

[43] S. A. Noghabi, J. Kolb, P. Bodik, and E. Cuervo, "Steel: Simplified development and deployment of edge-cloud applications," in *USENIX HotCloud, 2018)*.

[44] S. Venugopal, M. Gazzetti, Y. Gkoufas, and K. Katrinis, "Shadow puppets: Cloud-level accurate ai inference at the speed and economy of edge," in *USENIX HotEdge, 2018*.

[45] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *ACM ASPLOS, 2018*.

[46] 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 *ACM SoCC 2014*.

[47] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *ACM EuroSys 2016*.

[48] Y. Wang, S. Liu, X. Wu, and W. Shi, "Cavbench: A benchmark suite for connected and autonomous vehicles," in *2018 IEEE/ACM Symposium on Edge Computing (SEC 2018)*, pp. 30–42, 2018.

[49] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino, "Scalability and performance evaluation of edge cloud systems for latency constrained applications," in *2018 IEEE/ACM Symposium on Edge Computing (SEC 2018)*, pp. 286–299, 2018.

[50] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *ACM SoCC, 2013*.

[51] S. Srikanthan, S. Dwarkadas, and K. Shen, "Coherence stalls or latency tolerance: Informed cpu scheduling for socket and core sharing.," in *2016 USENIX Annual Technical Conference (ATC 2016)*, pp. 323–336, 2016.

[52] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "dcat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–13, 2018.

[53] "Intel fog reference design," 2017.

[54] M. K. Kazuaki Ishizaki and G. Koblents, "Ibm spark gpu," 2017.

[55] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[56] "Memcached." https://memcached.org.

[57] "Redis." http://www.redis.io.

[58] "Vmware software-defined data center." http://www.vmware.com/files/pdf/techpaper/Technical-whitepaper-SDDC-Capabilities-IToutcomes.pdf.

[59] "Intel(r) 64 and ia-32 architectures software developer's manual,"

[60] "Spec 2006 benchmark." https://www.spec.org/cpu2006.

[61] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: bare-metal performance for i/o virtualization," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 411–422, 2012.

[62] J. Li, N. K. Sharma, D. R. 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*, pp. 1–14, 2014.

[63] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. Chiueh, "A comprehensive implementation and evaluation of direct interrupt delivery," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 1–15, 2015.

[64] L. Cheng and C.-L. Wang, "vbalance: using interrupt load balance to improve i/o performance for smp virtual machines," in *Proceedings of the Third ACM Symposium on Cloud Computing*, pp. 2:1–2:14, 2012.

[65] Y. Xu, M. Bailey, B. Noble, and F. Jahanian, “Small is better: Avoiding latency traps in virtualized data centers,” in *Proceedings of the 4th annual Symposium on Cloud Computing*, pp. 7:1–7:16, 2013.

[66] J. D. Little and S. C. Graves, “Little’s law,” in *Building intuition*, pp. 81–100, Springer, 2008.

[67] “Linux perf.” https://perf.wiki.kernel.org.

[68] “Mutilate.” https://github.com/leverich/mutilate.

[69] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, pp. 53–64, 2012.

[70] 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 44th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 248–259, 2011.

[71] “Monasca.” https://wiki.openstack.org/wiki/Monasca.

[72] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. 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*, pp. 295–308, 2011.

[73] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 15–28, USENIX Association, 2012.

[74] C. Delimitrou and C. Kozyrakis, “Paragon: Qos-aware scheduling for heterogeneous datacenters,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 77–88, 2013.

[75] “Kubernetes.” http://kubernetes.io.

[76] 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 *2015 USENIX Annual Technical Conference (ATC 2015)*, pp. 485–497, 2015.

[77] “Apache hadoop yarn.” http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[78] H. Yang, A. Breslow, J. Mars, and L. Tang, “Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 607–618, 2013.

요약

컴퓨터 하드웨어가 발전함에 따라 데이터센터부터 임베디드 디바이스까지 다양한 분야들에서 머신 내 프로세싱 코어의 수, 캐쉬, 메모리 등 한 머신 내의 컴퓨팅 자원들이 증가하고 있다. 점점 늘어나는 머신 내 컴퓨팅자원들을 효율적으로 쓰기 위해서는 복수의 워크로드들을 통합함으로써 병렬적으로 수행하고 시스템 자원들을 공유하는 것이 필요하다. 하지만, 자원 공유는 공유되는 자원들에 대한 심한 경합을 야기할 수 있고, 이로 인해 워크로드들의 성능이 심각하게 저하될 수 있다. 더욱이, 워크로드들은 지연시간이 중요한 것부터 최선의 수행만을 목표로 하는 것까지 다양한 서비스 수준 목표를 가진 워크로드들이 있을 수 있는데, 이는 워크로드들의 통합을 어렵게 만든다. 경합 문제를 해결하기 위해서 운영체제와 하드웨어 제조사들이 다양한 자원 격리 기법들을 제시하고 있지만 여러 격리 기법들을 성능-중심적인 관점이 아닌 공평한 자원 할당의 관점에서 사용해오고 있다. 게다가, 기존 기법들은 성능 격리를 제공하지 않거나 효율적이고 효과적으로 수행하지 못한다. 본 논문은 기존 운영체제 및 하드웨어의 자원 격리 기술을 활용하여 성능 격리를 적응적으로 수행하는 피드백 방식의 성능 격리 최적화 기법들을 제안한다. 이를 위해서는, 자원 경합들을 평가하는 효과적인 온라인 프로파일링이 필요하다. 또한, 워크로드의 프로파일들에 따라서 격리가 수행되어야 한다.

본 논문에서, 우리는 세 가지 시스템에 대해서 온라인 프로파일링의 도움을 받는 성능 격리 최적화기법들을 제시한다. 첫째, 일반적인 멀티코어 서버 환경에서 하드웨어 및 소프트웨어 격리기법들의 특성을 고려하는 성능 격리 최적화를 제시한다. 둘째, 엣지 디바이스 환경과 같이 동적으로 자원 경합이 크게 변하는 특성을 고려한 적응적 성능 격리 최적화를 제시한다. 마지막으로, 공평성중심으로 자원을 프로비저닝하는 클러스터 환경에서 지연시간이 중요한 가상머신들의 자원 경합을 완화하는 계층적인 자원 경합 인지 스케줄링 최적화를 제안한다. 본 논문에서 제

안한 온라인 프로파일링 기법들과 적응적 최적화기법들의 효과를 증명하기 위해 다양한 시스템 환경(멀티코어, 엣지 디바이스, 그리고 클러스터)에 구현하여 검증하였다. 실험 결과는 우리의 제안하는 적응적 성능 격리 최적화 접근방식이 통합된 워크로드들이 수행되는 동적인 환경에서 효과적으로 자원 경합을 줄여 기존 기법들 대비 워크로드들의 더 낮은 수행 시간 및 지연 시간을 보이는 동시에 높은 자원 효율성을 달성할 수 있음을 보였다.

주요어: 성능 격리, 자원 관리, 자원 경합, 적응형 시스템, 멀티코어 시스템, 엣지 컴퓨팅, 클러스터

학번: 2014-30324