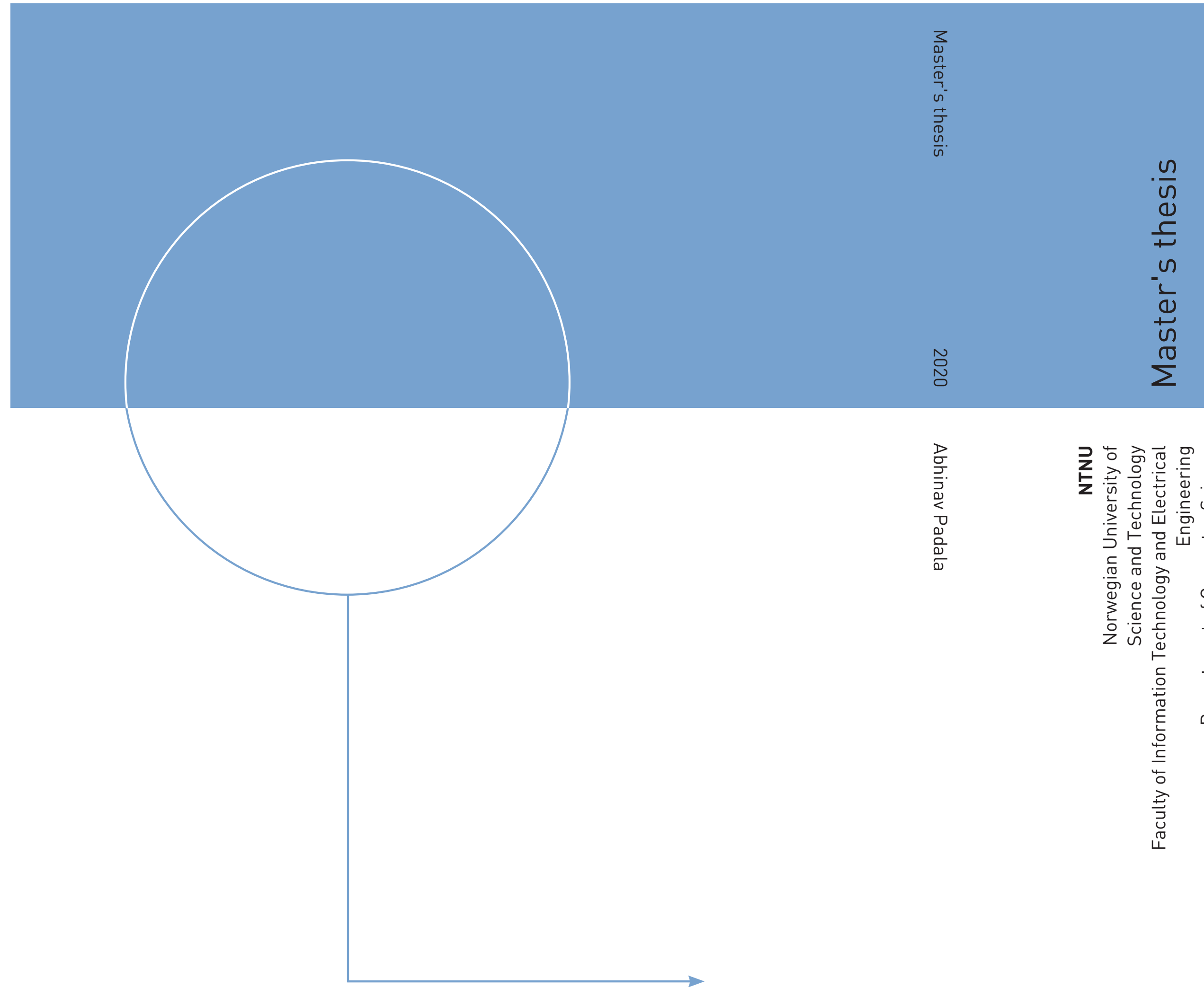

Abhinav Padala

Latency-aware Resource Management in Data Centres.

July 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

Latency-aware Resource Management in Data Centres.

Abhinav Padala

Information Systems
Submission date: July 2020
Supervisor: Rajiv Nishtala
Co-supervisor: Björn Gottschall

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Energy efficiency is a key issue in data centres. Data centres consume half of its maximum power even at low utilisation. In order to improve energy proportionality, machine utilisation is increased by co-locating best-effort (BE) workloads with latency-critical (LC) workloads. However, latency-critical workloads have strict quality-of-service (QoS) targets which must be met. When workloads are co-located, they share resources such as cores and last-level cache (LLC). A cluster manager is responsible for dynamically managing resources of the workloads in order to protect the performance of the LC workload while improving machine utilisation.

This thesis aims to study an existing cluster manager called Intel PRM. Intel PRM uses cycles per instruction (CPI) a throughput based metric, to make resource management decisions when workloads are co-located. We aim to optimise the existing cluster manager by modifying it to make decisions based on the application-level latency. This thesis only deals with CPU resource management. We succeed in improving the throughput of the best-effort workload from 4.6% to 54.0% while providing 100% QoS-guarantee.

Preface

I would like to thank my supervisors Rajiv Nishtala and Björn Gottschall for helping me throughout the project by providing valuable insights, and for introducing me to the world of resource efficient computing. I would also like to thank NTNU, Trondheim for providing the resources needed to conduct this study.

Assignment Text

QoS-aware cluster manager using machine learning.

A typical latency-critical service is based on client-server interaction, in which the client will send a certain request and the server side application will have to respond within a given time frame. This time frame is typically referred to as Quality of Service (QoS) target. To cope with user demand, such services are scaled across numerous servers. The question is how to maintain QoS, generally for any application that runs in a cluster environment, while aiming to minimise energy consumption or maximse throughput (by running batch jobs). For this, we need to do the following:

1. Run latency-critical services to multi-node scenario. Examples of latency-critical services given by programs in benchmark suite such as TailBench.
2. Experiment with and understand the existing cluster manager (Uber's Peleton).
3. Apply techniques using machine learning (such as Reinforcement learning) to optimise existing cluster scheduler (placement management strategies).

Research questions we would like to answer:

1. How workloads are currently scaled/distributed to multiple nodes to cope with user demand? i.e., exploring containers, Apache Mesos, etc.,
2. How do cluster schedulers work, and hypothetically, what is required to build one? (We will learn this from point (2)).
3. Optimise existing strategies with machine learning techniques .

Contents

Abstract . . . . **i**
Preface . . . . **iii**
Assignment Text . . . . **v**
Contents . . . . **vii**
Figures . . . . **ix**
Tables . . . . **xi**
Code Listings . . . . **xiii**
List of abbreviations . . . . **xv**
1 Introduction **1**
1.1 Research Objectives 2
1.2 Contributions 2
1.3 Outline 3
2 Background and Literature Review **5**
2.1 Background 5
2.1.1 Workloads in data centres 6
2.1.2 Tail latency 6
2.2 Linux scheduler 7
2.2.1 Scheduling policy of Linux 7
2.2.2 CFS Scheduler 8
2.3 Resource Manager 8
2.3.1 Latency vs CPI 9
2.4 Related work 10
3 Resource Managers Design **13**
3.1 Intel PRM Design 13
3.1.1 Core Isolation Mechanism 14
3.1.2 Design approach 15
3.2 Latency-aware PRM 17
3.3 Hybrid PRM 19
3.3.1 Tuning in Hybrid PRM 20
4 Implementation **21**
4.1 Intel PRM Implementation 21
4.1.1 Prerequisites 21
4.1.2 Implementation of Step One 21
4.1.3 Implementation of Step Two 24

4.2 Latency-aware PRM Implementation 25
4.3 Hybrid PRM Implementation 26
5 Evaluation 27
5.1 Experimentation Methodology 27
5.1.1 Hardware 27
5.1.2 Workloads 27
5.1.3 Determining Maximum throughput of Memcached 28
5.1.4 Workload co-location 29
5.1.5 Evaluation Metrics 30
5.2 Results 30
5.2.1 Intel PRM Results 31
5.2.2 Latency-aware PRM Results 31
5.2.3 Hybrid PRM Results 34
5.2.4 Tuned Hybrid PRM Results 34
5.3 Discussion 35
6 Conclusion 37
Bibliography 39

Figures

2.1 Impact of interference of a BE workload on the 95-th percentile tail latency of Memcached. . . . 7
2.2 Example of red-black tree [12]. . . . 9
2.3 Latency and CPI of a Memcached. . . . 10

3.1 High-level overview of Intel PRM 17
3.2 High-level overview of Hybrid PRM Implementation 20

5.1 Variation in 95-th percentile tail latency of Memcached with varying load. 29
5.2 Variation in 95-th percentile tail latency of Memcached with varying load-2. 30
5.3 Variation in 95-th percentile tail latency of Memcached when CPU resources are controlled by Intel PRM 31
5.5 Variation in 95-th percentile tail latency of Memcached when CPU resources are controlled by Latency-aware PRM - 2 32
5.4 Variation in 95-th percentile tail latency of Memcached when CPU resources are controlled by Latency-aware PRM 33
5.6 Variatation in 95-th percentile tail latency of Memcached when CPU resources are controlled by Hybrid PRM 34
5.7 Varitation in tail latency of Memcached when resources are controlled by Hybrid PRM tuned 35

Tables

5.1 Summary of QoS-guarantee of Memcached and Throughput of Stress-ng when scheduled by different PRM variants 30

Code Listings

3.1 Pesudo code for Heracles Top-level Controller 20
4.1 A sample workload configuration file 22
4.2 Python code to convert cpu usage of a container into a percentage value . 23
4.3 Algorithm for detecting if the BE workload is overutilising the cpu resources 25

List of abbreviations

TCO: Total Cost of Ownership
SLO: Service Level Objectives
LC: latency-critical
BE: best-effort
QoS: quality-of-service
RPS: Requests Per Second
PRM: Platform Resource Manager
IPC: Instructions Per Cycle
DVFS: Dynamic Voltage and Frequency Scaling

Chapter 1

Introduction

Data centres are places that provide software and hardware infrastructure that allow us to host large number of web applications [1]. In data centres, energy management is a key issue [2]. Barroso et al., suggest that servers in data centres are never idle nor are they utilised to the maximum extent. In most data centres, the server utilisation is between 10% and 50% of their maximum capacity. Servers consume about half of their full power even when the server utilisation is low. Low utilisation increases operational costs.
Many important web applications such as Web-search are latency-critical. They operate with strict quality-of-service (QoS) targets to improve the user experience. The load on these services is not always constant and sometimes it varies greatly. For example, in a 24 hour period, Google servers experience an average idleness of 30% [3]. During the times of low load, servers can be exploited by co-locating other non latency sensitive workloads or best-effort (BE) workloads to improve energy efficiency by increasing machine utilisation. However, workload management is a challenging task because of QoS policies of latency-critical (LC) workloads. Latency-critical workloads are expected to respond within a very short time frame, often within a fraction of second [1]. For latency-critical workloads, it is crucial that they meet the QoS tail latency [4]. Tail latency is the latency of slowest requests in the latency distribution. When other workloads are co-located with latency-critical workloads, there might be cases where the workloads may undergo a resource conflict. Resource interference negatively affects tail latency of latency-critical workload. Hence, co-location is a challenging task because while trying to improve machine utilisation, we must also consider QoS targets of latency-critical workloads. Workloads running on the same server share resources such as cores, last-level cache (LLC) etc. Cluster managers predict when a latency-critical workload is suffering from resource interference and suspend the co-located workloads to protect the LC workload performance. However, terminating the workloads completely for the sake of protecting LC workloads reduces the opportunity for improving machine utilisation.

1.1 Research Objectives

The research objectives are presented as a high-level goals and sub-goals are defined to present work to be done to achieve the objective.

1. Examine the behaviour of latency-critical workloads when co-located with batch workloads.
2. How do cluster schedulers work, and hypothetically, what is required to build one?
    - Experiment with existing cluster manager.
    - Explore the strategies used by the cluster manager to manage shared resources when workloads are co-located.
3. Optimise existing strategies of the cluster manager to improve machine utilisation.

1.2 Contributions

During this study, we explored two existing cluster managers: Uber Peloton and Intel PRM. Peloton was developed by Uber to manage diverse workloads in their organisation [5]. Initially, Peloton was chosen to conduct this research. During the exploration phase, it was found that Peloton is tightly coupled with many open-source projects such as Mesos, Zookeeper, Cassandra etc. Considering the time constraints of the Thesis, the idea to work with Peloton was dropped because it requires understanding of the tools that are coupled with it. Intel PRM is another Resource Manager that co-locates best-effort (BE) workloads with latency-critical workloads in an attempt to improve machine utilisation [6].[1] To the best of our knowledge, there is no academic publication that conducted a study on Intel PRM and this is the first known study that is conducted to optimise strategies of Intel PRM. Experiments are conducted by running a latency-critical workload and a best-effort workload on the same server. During the study, we found that Intel PRM uses cycles per instruction (CPI), a throughput based metric to manage workloads on the server. Even though it helps in co-locating best-effort workloads, we found that by making it latency-aware we can improve the machine utilisation greatly. This study deals with CPU resource management only. The major contributions of this work are:

1. Exploring Intel PRM to understand the strategies used by it to manage CPU resources when workloads are co-located.
2. We present Latency-aware PRM which uses strategies from Intel PRM but makes resource management decisions based on application-level latency of the latency-critical application. We show that it improves the throughput of the co-located workload from 4.6% to 56.2%. However, in some cases, QoS tail latency is not met.
3. We present Hybrid PRM, which uses strategies from Intel PRM and Heracles, a resource manager used to improve machine utilisation.

[1] cluster manager and resource manager are used interchangeably. They are used to manage resources in clusters.

4. Furthermore, we tune parameters of Hybrid PRM in order to improve machine utilisation. We show that, this method improves the throughput of co-located workload to 54% while providing 100% QoS-guarantee.

1.3 Outline

Chapter 2: The chapter discusses the background of the problem, types of workloads in the data centres, tail latency and presents other works that deal with resource management in data centres.
Chapter 3: The chapter presents the design of Intel PRM. It also discusses the design of the variants of the Intel PRM developed during the study in an attempt to improve machine utilisation.
Chapter 4: The chapter presents implementation details of the variants of Intel PRM that could not be covered in Chapter 3.
Chapter 5: The chapter discusses the evaluation methodology and presents results obtained when resources are controlled with different variants of the resource manager.
Chapter 6: The chapter concludes the research.

Chapter 2

Background and Literature Review

This chapter discusses the motivation behind the study and the related background . We briefly discuss the problem, complications in the problem and provide a high level overview of the solution. Later, we present the related work conducted by other researchers and organisations to address the problem.

2.1 Background

Data centres are places which host several servers that serve clients of applications [1, p. 2]. Data centres provide hardware and software infrastructure which allows businesses and organisations to deploy a large number of web applications. when it comes to operating and maintaining data centres, the total cost of ownership (TCO) includes infrastructure costs, electricity bill and other energy-dependent factors such as cost of energy which is required to cool hardware infrastructure. Barroso et al., [2] suggest that computer-energy consumption is a growing concern and shows that current trends indicate energy as a crucial factor in the total cost of ownership.
A system is said to be energy proportional if its energy consumption is directly proportional to the resource utilisation [7]. An ideal energy proportional system does not consume power at the idle stage and consumes power linearly with respect to utilisation of the system. For example, an ideal energy proportional system consumes 20 % of its maximum power when it is 20% utilised. Servers in the data centres are not energy proportional and they tend to consume excessive energy even when the utilisation is low. One way to improve return on investment is to maximise server utilisation. However, load on web applications running in data centres is not always constant. The load on most web applications differ greatly based on the number of end-users using it. A study conducted by Bilgir et al., suggests that in a 24 hour time period, load on Facebook data centres vary greatly and can go below 40% and above 90% of its maximum allowed capacity [8]. The average utilisation in most data centres is low, even at low utilisation servers still consume about 60% of their maximum energy [2, 8]. One straight-forward way to improve energy proportionality in data centres is to co-locate multiple workloads on servers and saturate resources [9]. However, co-locating workloads on the same server without taking proper measures has negative implications because of the nature of the

workloads.

2.1.1 Workloads in data centres

On a high level, there are two broad classes of workloads in data centres: online services and batch workloads [1, p. 24]. Online services such as web-search, social networking are latency-critical. These services are required to have low user-perceived latency to improve end-user experience. Along with user-facing services, back end services used by user-facing services such as Memcached is expected to operate with low latency. High throughput is also another key performance metric because popular internet services like these must serve thousands of requests per second [1, p. 25]. An example of a latency-critical service is Google search. Google search system performs the search and shows the results within a few milliseconds. It also interactively predicts and updates query results as user types in the search box [4]. As the size and complexity of the service and system that hosts the service increases, it becomes challenging to keep worst-case latency of the service short because worst-case latency gets amplified as system scales [4]. The worst-case latency can be referred to as Tail latency. Tail latency is the latency of the slowest requests in the tail of the latency distribution [1, p. 30]. On the other hand, batch workloads or best-effort (BE) workloads are non-interactive and non-latency sensitive workloads. Some examples are the workloads that perform data analytics, model training, backing up data and other maintenance tasks. Batch workloads have loose completion deadlines and are immune to variations in their completion time.

2.1.2 Tail latency

Generally, the communication between the machines in data centres takes place within a few milliseconds. In some cases, the latency values go beyond the average and lead to slower responses to requests. A 95th percentile tail latency of 10 ms means that 5 requests in every 100 requests are experiencing a delay of 10 ms. Popular web services handle hundreds of thousand requests a day and a delay in 5 % requests could create a bad experience for many end-users. This is a case when requests are served by a single server. In distributed systems, 1 client request may have to collect responses from tens to hundreds of servers. The severity of tail latency increases with scaling. Workloads that are latency-critical are commonly referred to as LC workloads. LC workloads have strict service level objectives (SLO) on tail latency. Various workloads in the data centres have different quality-of-service (QoS) demands. QoS of an LC workload can be understood as a relevant performance metric described in its service level requirements [10]. For instance, the QoS of Google's web search is measured using query latency and RPS (queries-per-second). Another example is Memcached which has QoS target that 95-th percentile tail latency must be below 10 ms. When running latency-critical workloads in data centres measures are taken to meet the QoS demands of the workload.
Conventional knowledge says that when LC workloads are co-located with other workloads they do not perform well. This is because co-located workloads can interfere with shared resources such as cores, last-level cache (LLC) that can lead to performance degradation of LC workloads which in turn leads to high variability in tail latency. To

avoid this, latency-critical workloads are run alone. This results in low machine utilisation when load is low and high energy consumption even at low machine utilisation. The variability in 95-th percentile tail latency of Memcached, when co-located with a batch workload is demonstrated using Figure 2.1. The figure demonstrates the effect of co-location on the LC workload when the workload is scheduled by Completely Fair Scheduler (CFS) of the Linux Operating System. More about conventional scheduling in Linux is discussed in section 2.2.

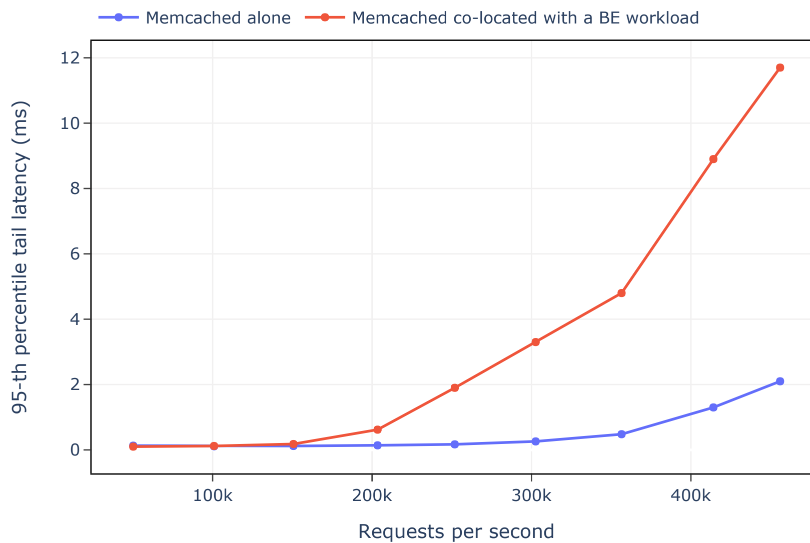

Figure 2.1: Impact of interference of a BE workload on the 95-th percentile tail latency of Memcached.

2.2 Linux scheduler

This section discusses the conventional scheduling policy of Linux Operating System to dynamically manage CPU resources allocated to the workloads running on the machine.

2.2.1 Scheduling policy of Linux

In a machine with a single core, Linux executes multiple processes by switching control of execution from one process to another process. An instance of a program executing on a machine is called a Process. Control switching happens within a very short time frame. The method that deals with when to switch control and which process to select to execute is called Scheduling. Usually, all the available resources of the machine are shared by the processes running on the machine. The sharing of resources to processes

is handled by Scheduler. Process starvation is a scenario where a process is in need of resources but is not given any resources because of resources being used by other processes. This arises because of resource conflicts amongst process. A scheduling algorithm is responsible for avoiding resource conflicts and process starvation. A Scheduling Policy is that which consists of rules that are used to determine which process to execute and when to switch control to another process [11]. In Linux, CPU time is divided into time slices. Each process is given one time slice worth of CPU time and when the time slice of that process expires, control switches to another process. At the time of conducting this study, Completely Fair Scheduler (CFS) is the process scheduler used by Linux.

2.2.2 CFS Scheduler

CFS is the process scheduler in Linux systems starting from version 2.6.23. CFS tries to maintain fairness when providing CPU time to processes running on the system. The amount of CPU time provided to a given process is called as virtual runtime [12]. CFS is responsible for keeping track of virtual time given to processes and any process which has been given an unfair amount of CPU time must be compensated by providing it the CPU time it needs. In other words, the smaller the virtual time of a process, the greater is its need of CPU time and vice versa. CFS achieves fairness by using a red-black tree. All the processes which need to be scheduled are stored in a red-black tree. Processes with low virtual time are stored on the left side of the tree and the process with higher virtual time are stored on the right side of the tree. CFS picks the processes from the left side, executes them for a certain period and adds the time it allocated to the process to the virtual time. If the execution of the process is complete, it removes the process from the tree or else the process is moved to the right side of the tree. By following this process, process with the need for the processor are given CPU time and are moved to the right side. Of the available process, the processes on the right with low virtual time are moved to the left to maintain fairness. An example of a red-black tree is represented in Figure 2.2. The idea of fairness does not work well in data centres as LC workloads have to be given higher priority over other workloads and at the same time machine utilisation must be improved. To address this researches are coming up with different techniques that can be used to develop resource managers that can help address the problem at hand.

2.3 Resource Manager

The problem we are trying to address can be formulated as improving machine utilisation in data centres by co-locating a BE workload with the LC workload while trying to meet QoS demands of the LC workloads. In this context, a resource manager is that which attempts to address the problem by allowing co-location of BE workload along with the host application (LC workload) in order to improve machine utilisation by saturating resources without compromising performance of the LC workload. Resource Managers dynamically allocate shared resources to workloads running on the machine and manage them in a way as to avoid performance degradation of the LC workload

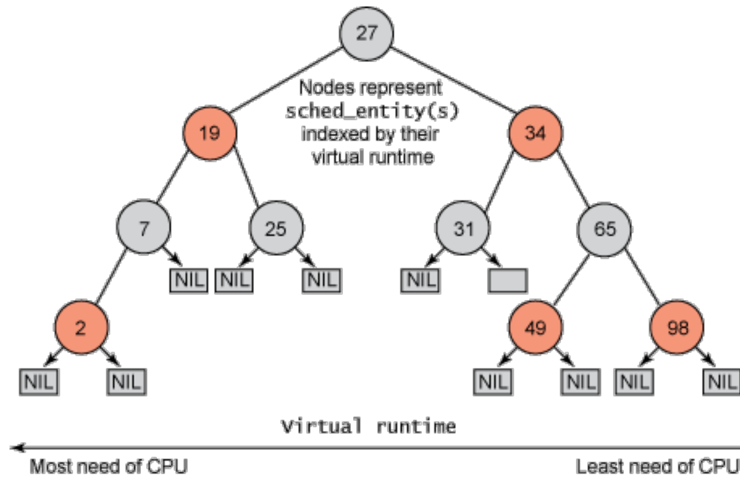

Figure 2.2: Example of red-black tree [12].

while attempting to improve energy proportionality. Researchers have suggested various techniques to build resource managers which are discussed in section 2.4. Resource managers like Intel PRM uses cycles per instruction (CPI) as the primary metric to make scheduling decisions when workloads are co-located on the server and some other like Heracles [3] use application-level latency as a guiding metric to make scheduling decisions.

2.3.1 Latency vs CPI

Some resource managers use throughput based metrics such as cycles per instruction (CPI) as a guiding metric to make scheduling decisions. CPI gives measurement of throughput and serves well when dealing with only batch workloads. However, when dealing with latency-critical workloads, CPI fails to capture how well the workload is running and how well it is meeting its QoS target. To demonstrate this, we conducted a simple experiment. We measured 95-th percentile latency and CPI of Memcached workload. While executing it, we simulated shared resource interference by applying pressure using a best-effort workload. The pressure on shared resources made the latency vary from the latency when the workload was running alone without any pressure on shared resources. We collected latency and CPI at various times and plotted the subplots to demonstrate how the metrics changed from one point to another. We observed that, given the CPI value it is difficult to determine how well a LC workload is meeting its QoS target. The QoS target of Memcached is 10 ms. In Figure 2.3, even with almost same CPI, 95-th percentile tail latency is below and above the QoS target which leads to ambiguity in determining the performance of the LC workload. We conclude that using CPI as a primary metric, it is not possible to determine the latency behaviour of the workload.

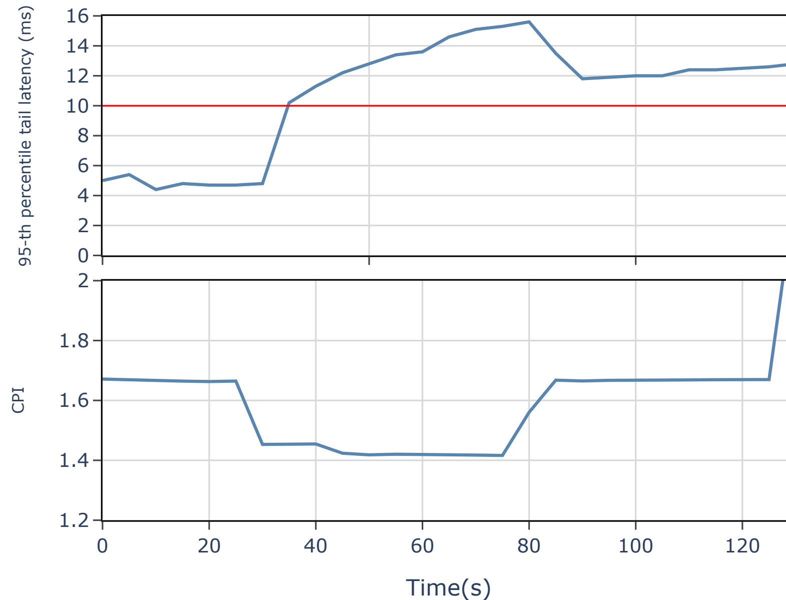

Figure 2.3: Latency and CPI of a Memcached.

2.4 Related work

Uber Peloton is a resource scheduler for managing workloads in a cluster environment [5]. It is built to improve machine utilisation in clusters when there are vast fluctuations in the load. It can manage resources of various kinds of workloads such as stateless, stateful and batch workloads. It co-locates workloads in a cluster and dynamically manages resources allocated to the workloads based on the requirements of the workload. Peloton organises workloads into different groups and allocates fixed resources to them. When the load on a specific group is low, resources of that group are lent to another group of workloads. Resources are returned back to the original group of workloads if there is an increase in load on the workloads of that group.

Bubble-Up [10] is a characterisation methodology that attempts to predict the performance degradation of workloads because of shared resource contention by co-located workloads. Prediction is carried out as a two step process. In Step 1, it measures to what degree the workload suffers when exposed to different levels of pressure on shared resources. This step tests how much pressure the LC workload can handle before violating QoS demands. In step 2, it measures the amount of pressure a BE workload puts on the shared resources. Using this measurements, Bubble-Up co-locates a best-effort workload with the latency-critical workload while respecting QoS policy of the LC workload.

Since it is already aware of the pressure the LC workload can take and pressure the BE workload generates, it handles resources in a way that pressure on the LC workload does not affect its QoS target. Bubble-Up method tries to improve machine utilisation while enforcing QoS policies by allowing co-location of workload with the LC workloads only when the predicted performance degradation of the LC workload is within specified thresholds. It is based on static profiling which means it requires prior knowledge of the workloads and data processing is conducted offline.

Bubble-flux [13] is based on run time approach and is a dynamic resource interference measurement methodology to maximise server utilisation. Bubble-flux probes server in real-time to measure any resource interference on the LC workloads and frequently predicts QoS of the LC workloads. The resource interference is measured by monitoring last-level cache occupancy and memory bandwidth. When it comes to controlling resources, another component frequently monitors the predicted QoS and adjusts resources allocated to BE workloads based on the condition whether the LC workload is meeting its QoS target or not. Bubble-flux does not require prior knowledge of the workloads that will be co-located and can adapt to the dynamic behaviour of workloads. It can be used to co-locate more than two workloads.

Leverich et al., [9] demonstrate how co-location can lead to QoS violations, using Memcached as a LC workload. In the study, they show the reasons such as queuing delay, load imbalances lead to QoS violations in Memcached when co-located with other workloads. They propose techniques like adjusting cpu shares and enforcing cpu bandwidth limits on co-located workloads as mitigation techniques to avoid QoS violation of an LC workloads. Setting cpu resource limits help to allocate fixed resources for the workloads. Ultimately, they demonstrate that it is possible to aggressively co-locate other workloads with a low latency sensitive workload like Memcached by respecting QoS policies. They use Memcached as a latency-critical workload and the throughput of SPEC CPU2006 benchmarks to determine effective machine utilisation when workloads are co-located.

Heracles [3] is a feedback-based controller that manages shared resources such as CPU, last-level cache (LLC) when BE workloads are co-located with a LC workload. Heracles controller frequently polls tail latency of the LC workload and calculates slack which is the difference between latency target we want to achieve and current latency value. Based on the slack value, Heracles allocates or takes away resources to and from the LC workload and BE workloads. Heracles also has another safeguard to protect LC workloads from resource interference. When the load on LC workload is greater than 85% of its maximum capacity, Heracles disables the execution of best-effort workload and allocates all the resources to the latency-critical workload.

Pythia [14] is another co-location manager that makes scheduling decisions based on instructions per cycle (IPC) metric. It considers IPC as a QoS metric of the latency-critical workload. It calculates the contention score of the best-effort workloads which we want to co-locate. Contention score of BE workload is associated with the LC workload and it is different for different LC workloads because some workloads are more sensitive to resource contentions while some are less sensitive. Initially, the contention score of each best-effort workload is calculated when co-located with the LC workload. After that, BE workloads whose contentiousness is detrimental to performance of the LC workload are

identified. When co-locating workloads, it selects best-effort workloads that does not negatively effect the latency-critical workload.

Hipster [7] is a solution that combines heuristic techniques and reinforcement learning to map latency-critical workloads and best-effort workloads on heterogeneous cores. It also attempts to select optimised Dynamic Voltage and Frequency Scaling (DVFS) settings to improve energy efficiency.

Cui et al., [15] propose a framework to detect sources that result in tail latency in an event-driven web services. They specifically conduct a study on asynchronous event-driven execution using Node.js framework. Node.js a JavaScript-based framework to develop event-driven web applications [16]. The authors suggest that, asynchronous event-driven services put forward a unique challenge in identifying sources of tail latency because of their event-driven nature. They identified that the major bottleneck of tail latency in Node.js is CPU processing because of the involvement of event queue and JavaScript garbage collector and propose a framework to reduce tail latency.

Paragon [17] is a QoS-aware scheduler that is also aware of interference on shared resources from co-located workloads. Paragon can schedule workloads that are unknown to it. When a new workload is introduced Paragon classifies the workload to the existing group of workloads and determines the server configuration on which the workload can perform better. While determining the server configuration, it also finds out the level of interference the workload can cause on other co-located workloads and also the level of interference it can tolerate. After scheduling, Paragon frequently monitors workload's performance metrics and alters scheduling decisions based on the workload behaviour.

Kubernetes [18] is an open-source container orchestration system which helps to deploy, manage containerised workloads. A containerised workload is also referred to as a container. A Linux container is that which consists of all the required dependencies that are needed to run a workload. Kubernetes helps to manage a cluster of containers and their hardware resources in order to maximise resource usage. It is mostly used in production servers and contains many other business-related advantages.

Chapter 3

Resource Managers Design

This chapter discusses the design of the variants of resource managers that are developed during the study. The variants are based on Intel Platform Resource Manager (PRM) design. Taking Intel PRM as a base, new variants have been developed by incrementally adding new methods in an attempt to maximise resource utilisation of data centres while respecting QoS policies of LC workloads. Each section describes one variant.

3.1 Intel PRM Design

Intel Platform Resource Manager (PRM) helps to co-locate best-effort workloads with latency-critical workloads in a cluster environment [6]. Intel PRM relies on Intel Resource Director Technology (Intel RDT) which has the capability to control shared resources such as Last Level Cache (LLC) and Memory Bandwidth which are used by the workloads. Intel RDT provides features for cache monitoring, cache allocation, memory monitoring and memory allocation. To manage CPU resources, Intel PRM uses tuning parameters provided by the Linux Kernel. This is discussed in detail in section Section 3.1.1. Intel PRM can control different shared resources that are available in a cluster. The focus of this study has been narrowed down to CPU resource management. The design addresses a scenario where there are only two workloads on the server, that is one latency-critical workload co-located with a best-effort workload. This is also referred to as pairwise-co-location.

Xu et al., has classified the techniques to schedule workloads in data centres into two categories: static scheduling and dynamic scheduling [14]. In the static scheduling, models which are used to make scheduling decisions are built offline and are used to predict the performance of the co-located applications [10]. In this approach, the system is aware of the workloads that are being co-located on the server. In the dynamic approach, performance metrics of the workloads are profiled online. Unlike static approach, dynamic approach is tolerant towards unforeseen situations such as situations where a new workload is deployed dynamically. Dynamic approach has capability to deal with unknown incoming workloads and unknown application behaviour.

Intel PRM uses a combination of static and dynamic approaches. Initially, it monitors and records the performance metrics of the latency-critical workload and a threshold

model is generated using the collected data. The threshold model is then used to detect any performance degradation caused on the LC workload by dynamically monitoring the current performance metrics of the workload and comparing them against the best case metrics that are determined when building the threshold model. This approach is referred to as dynamic monitoring and control approach [14]. In dynamic monitoring and control approach, when the system detects that the LC workload is affected by a BE workload, it schedules out BE workload for a certain specified period. However, Intel PRM has a different method to tackle such situations. Instead of scheduling out BE workloads, it sets the shared resources allocated to BE workloads to a bare minimum.
Intel PRM handles resource allocation dynamically. It uses different isolation techniques to isolate and handle different shared resources separately. The primary shared resource in the server are the cores. Section 3.1.1 discusses the techniques used by Intel PRM to manage cores when workloads are co-located. Furthermore, section 3.1.2 discuss the design of Intel PRM by categorising it into two steps.

3.1.1 Core Isolation Mechanism

Control groups

Control groups, also shortly referred as 'cgroups' allow us to group processes or tasks running in a system and allocate resources such as CPU time to that group of processes. Cgroups provide methods to group a set of tasks including their children processes into hierarchical groups. Using cgroups, the grouped processes can be associated to a set of parameters for subsystems [19]. A subsystem is a module and can be considered as a resource controller that can set resource restrictions on a set of processes. There are multiple subsystems and each subsystem represents a single resource such as CPU time or memory [20]. The subsystems relevant for this study are given below:

- **cpu:** This system uses scheduler to allocate CPU to cgroup processes.
- **cpuacct:** cpuacct generates reports for CPU resource usage by cgroup processes.

Intel PRM makes use of the concept of cgroups to categorise processes of a workload. It collects metrics related to cpu utilisation of the workload by reading metrics from cpuacct subsystem. It alters CPU resource allocated to the workload by tuning the parameters present in cpu subsystem. These are called tunable parameters.

CFS tunable parameters

In CFS, there are two ways using which CPU resources can be allocated to cgroups. One is relative tuning and the other is ceiling enforcement tuning [21]. In relative tuning, CPU resources allocated to cgroups is based on relative shares and in ceiling enforcement tuning, CPU resources are allocated by setting a hard limit which tells scheduler about how much resources a cgroup is allowed to use. This is briefly discussed in the subsections below:
Relative tuning: cpu share is a relative tunable parameter in CFS and it contains an integer value that describes the relative amount of CPU time given to the processes of

a cgroup. Each cgroup has its own cpu share parameter. For example, if cpu share of two cgroups is set to same value then it implies that processes of those two cgroups will receive equal amount of CPU time. Continuing the same example, if the cpu share of third cgroup is set to twice the value then it implies that the third cgroup will receive twice the CPU time than the first two cgroups. The minimum value that can be specified in cpu share is 2. Using relative tuning for resource management has implications. Relative tuning makes it hard to determine how much CPU time a cgroup is assigned. Since it is relative, the amount of CPU time assigned to a cgroup depends on the number of cgroups created in the system. For example, two cgroups with relative share of 1000 receive 50% of CPU time each. If the share of third cgroup is set as 1000 then the CPU time received by all the three cgroups is equal to 1/3 of total available CPU time. If we add another cgroup and set a relative share of 1000 then CPU time received by four cgroups becomes 25%.
Ceiling Enforcement tuning: Unlike relative tuning, this type of tuning helps in setting hard limits on the amount of CPU that a cgroup can use. To implement this type of tuning cfs period and cfs quota parameters are used. The quota that can be set to a cgroup depends on the integer value set to cfs period. The default value of cfs period in the machine that is used for this study is 100000 microseconds. By setting integer value x to cfs quota of a cgroup allows all the processes of that cgroup to run for x micro seconds during one period, which is 100000 microseconds. After processes run for the specified time, they are throttled and are only allowed to run during the next time period. Alternatively ceiling enforcement tuning can be understood intuitively using the following example: To make a cgroup use 100% of a single core set cfs quota value of a cgroup to 100000. In a multi-core machine setting the quota value to 200000 makes a cgroup use 2 cores. This behaviour has been observed during the study by running a workload, setting the parameters and by monitoring its CPU utilisation. The default value of cfs quota is -1 and it indicates that the cgroup is not under any CPU time restrictions. The lower limit of the cfs quota is 1000 microseconds [21].
Intel PRM uses both relative tuning and ceiling enforcement tuning to manage CPU resources allocated to BE workloads. This is discussed in the section 3.1.2.

3.1.2 Design approach

Intel Platform Resource Manager's design can be explained by classifying the methodology of it into two steps:

1. In Step 1, the best case performance metrics of the latency-critical workload is determined. This is done by collecting performance metrics of the latency-critical workload and by statistically analysing the collected metrics to build a best case threshold model.
2. In Step 2, the threshold model is used to detect resource contention by predicting the resource interference caused by best-effort workload. Then scheduling decisions are made in a way that the performance of the latency-critical workload is not affected by resource interference caused by best-effort workload.

Step One: Building threshold model

In Step one, Intel PRM determines the best case performance metrics of the latency-critical workload. It has two monitors that run with different time periods. For discussion purposes the monitors are hereby referred to as main-monitor and sub-monitor. When a latency-critical workload is running on the server, main-monitor polls for platform metrics such as cycles (C), instructions (I), cpu utilisation and other performance related metrics of the workload. Main-monitor collects metrics for every 18.5 seconds [1]. Along with main-monitor, sub-monitor polls for cpu utilisation of the workload for every 2 seconds. The usefulness of the sub-monitor is discussed in Step two.
Main-monitor uses Intel RDT to collect metrics. The primary metric of interest here is cycles per instruction (CPI) which is calculated for every 18.5 seconds. It denotes the number of cycles needed to execute an instruction of the latency-critical workload. Sub-monitor reads *cpuacct* subsystem to collect cpu utilisation of the latency-critical workload for every 2 seconds. *cpuacct* reports total CPU time consumed by the cgroup in nano-seconds. Intel PRM converts the CPU usage of the workload into a percentage value. For example, a cpu utilisation of 100% means that the workload is utilising one core to full extent. More details about converting CPU time into cpu utilisation percentage value is discussed in Chapter 4.
After collecting the performance and cpu usage metrics, a threshold model is built by analysing the collected data. The analysis is done offline. Intel PRM, by default, uses Gaussian Mixture Model to find the best case CPI. It considers maximum recorded cpu utilisation of the latency-critical workload as the total available cpu utilisation of the machine. Consider the following example: let us say a machine has 10 cores which gives us the possibility to use 1000 % of CPU resources. Let us say that the maximum recorded cpu utilisation of the LC workload is 800%. Even though LC workload uses 200% less cpu, Intel PRM considers the 800 % as the maximum available cpu resource on the machine and ignores the remaining resources.

Step Two: Contention detection and resource controlling

The second step includes detecting shared resource contention and controlling the shared resources of the best-effort workload when it is co-located with the latency-critical workload. Initially, when a LC workload is co-located with a BE workload, Intel PRM sets shared resources allocated to the BE workload to minimum value possible. In case of CPU resources, it sets cpu shares and cpu quota of the BE workload to minimum integer values. Main-monitor polls for cycles, instructions of the latency-critical workload and calculates CPI periodically. CPI is lesser the better metric. If the current CPI of the LC workload is lower than CPI value in threshold model then it means that the LC workload is getting the required CPU resources and there is no interference on shared resource that leads to performance degradation of the LC workload. The comparison is done by a contention detector. Along with the main-monitor, sub-monitor runs for every 2 seconds to check if best-effort effort workload is consuming more cpu resources that may negat-

[1] From the code, we observed that by default main-monitor polls for every 20 * 1000 - 1500 milliseconds

ively impact the latency-critical workload. The idea is that the sum of the cpu utilisation of the LC and the BE workload must not be greater than maximum recorded cpu utilisation determined in Step One. When these safeguards are not active, Intel PRM uses the opportunity to increase CPU resources of the BE workload by a constant step. When two monitors find that there is no interference from the best-effort workload, then the resources allocated to the BE workload are increased incrementally. The CPU time available on the machine is divided into 20 levels and one level is increased whenever corresponding condition is met. Increasing of resources to the best-effort workload happens periodically until one of the monitor finds that the current configuration is detrimental to the performance of the latency-critical workload. When the resource level of the BE workload reaches level 20, Intel PRM handles control over to the CFS scheduler.
If the current CPI is higher than that of CPI determined during the threshold model generation, then it means that the best-effort workload is consuming more shared resources that can lead to performance degradation of the latency-critical workload. If there is contention, it signals a resource controller that a contention has been detected and resource allocation to the best-effort workload must be altered to keep the performance of the latency-critical workload healthy. Resource controller controls the corresponding resource [2]. A corresponding resource, in this case is cpu quota of best-effort workload. Intel PRM mitigates contention by setting the CPU resources allocated to best-effort workload to minimum value by tuning relative and ceiling enforcement parameters as discussed in 3.1.1. Along with the main-monitor, the sub-monitor runs for every 2 seconds to check if BE workload cpu consumption is affecting the LC workload utilisation. If a violation of this rule is detected, then contention detector sends a signal to the resource controller that resources of best-effort workload must be set to minimum.

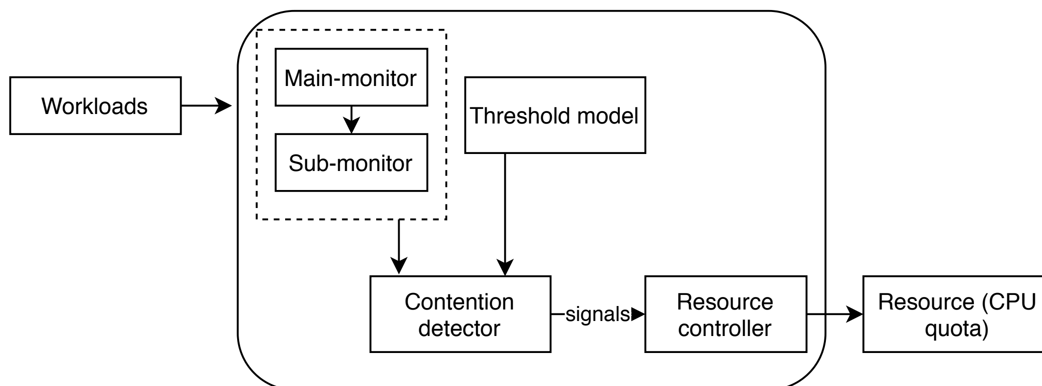

Figure 3.1: High-level overview of Intel PRM

3.2 Latency-aware PRM

As discussed in the Chapter 2, CPI is not a reliable metric when co-locating workloads with a latency-critical workload. To overcome this limitation, we attempt to make

[2]In general, resource can be cpu quota, LLC.

scheduling decisions based on the application-level latency of the LC workload. LC workloads have QoS targets. For example, a latency-critical workload has QoS target that the 95-th percentile latency of the workload cannot be more than 10 ms. Latency-aware PRM is a variant of the Intel PRM which makes scheduling decisions based on QoS target of the LC workload rather than depending on CPI as originally done by Intel PRM. This variant implements an iso-latency method, which means that it can improve machine utilisation as long as latency-critical workload is meeting its QoS target. Iso-latency management means adjusting resources in a way such that the latency-critical workload barely meets its QoS target [22].

This variant consists of only one monitor called as QoS-monitor which periodically polls for tail latency of the LC workload. The period with which it polls for tail latency is equal to the period of Main-monitor which is (20*1000 - 1500) ms. We consider that there is a shared resource contention when latency of the LC workload is beyond the QoS target. This variant adopts the idea of original Intel PRM where a contention is detected only when current CPI is greater that determined threshold CPI. Similar to that idea, the detector of this variant considers that there is an interference when the current tail latency is greater than the QoS target of the LC workload.

The available CPU resources are divided into 50 levels. The number of levels has been increased from 20 to 50 because some applications like Memcached are sensitive to resource interference. A small resource interference on workloads like Memcached can cause significant QoS violations. To avoid this and to carefully allocate resources to BE workload the number of levels has been increased. By increasing levels, we are trying to lessen the quantity of resource associated with each level. Intel PRM treats the maximum recorded CPU utilisation value of LC workload as the total CPU resource value available on the machine. This wastes the CPU resources which have not been used. This variant uses all the available CPU resources on the machine. For example, in a machine with 20 cores, by increasing each level we increase CPU resources equal to (20/50) cores. A limit is set on maximum number of levels that can be assigned to a BE workload. The limit is 48 and we determined it empirically while conducting experiments. The reason behind setting the limit is because we observed that when resource level of the BE workload is maximum, Intel PRM handles control over to CFS scheduler. To avoid this and to have complete control over the way resources are allocated, we set a limit. By setting a hard limit on the resource we also make sure that there is also some CPU resources left for the latency-critical workload to use.

Latency-aware PRM contains a new component called Fetcher which collects tail latency from the latency-critical workload. Fetcher also stores the most recent latency data. QoS-monitor polls for tail latency periodically from Fetcher and the detector decides if the the resources of the BE workload must be increased or decreased depending on the current latency value and specified best case latency value. When detector detects that the tail latency of the workload is below the threshold it signals resource controller to increase the level of CPU resources allocated to BE workload by 1 and keeps repeating this process. In case, if the tail latency is above or equal to threshold, detector signals resource controller to set the CPU resources of best-effort workload to minimum. Similar to Intel PRM, this variant uses CFS tunable parameters and sets cpu shares and cpu quota

to minimum.

3.3 Hybrid PRM

Hybrid PRM is another variant based on the previous variants. It adopts methods from Intel PRM, it is latency-aware and also adopts methods from another resource manager called Heracles. Heracles is dynamic resource manager which makes scheduling decisions based on application-level latency. It manages shared resources such as cores, last-level-cache, memory bandwidth etc when workloads are co-located with a latency-critical workload. Hybrid PRM uses the core management algorithm from Heracles resource controller. Heracles manages resources of both LC and BE workloads. The current variant extracts parts of Heracles which deal with BE resource management only and leaves LC resource management to Completely Fair Scheduler scheduler.
The idea behind Hybrid PRM is that, resources must be saturated to improve the machine utilisation. The resources can be saturated as much as possible, by letting the co-located BE workload use them, as long as performance of the LC workload is not affected negatively. In this case, latency of the workload is used as a metric to determine performance. Hybrid PRM continuously monitors tail latency and latency slack to make scheduling decisions. Latency Slack is the difference between the target latency we want to achieve and current latency the workload is experiencing. It also considers the load on the LC workload as input to decide if a workload must be co-located or not. Because as discussed previously, load on many web applications varies greatly. Hybrid PRM disables co location when load on the latency-critical workload is above certain threshold. Unlike Intel PRM, this variant does not require any threshold model building but it needs value of latency target we want to achieve.
This variant consists of two controllers: Top-level controller and Sub-controller. The pseudo code for the Top-level controller is given in code listing 3.1. The high-level overview of Hybrid PRM is presented using Figure 3.2. Similar to Latency-aware PRM, Fetcher is responsible for collecting metrics. Here, it collects two metrics: latency value and load. Top-level controller polls for tail latency and load of the LC workload from Fetcher for every 15 seconds. If the load on the LC workload is higher than 85% of its maximum capacity, Top-level controller suspends the execution of the BE workload. A co-located best-effort workload is allowed to run only when the load on LC workload is below 80% of its maximum capacity. This is done to guard the latency-critical workload from resource interference caused by BE workload and prevent the workload from violating QoS policies because of pressure from the BE workload. When the slack value is negative, Top-level controller suspends co-location by suspending the BE workload. The slack value becomes negative when there is a spike in load on the workload. When this happens, the BE workload is suspended for 5 minutes to let the LC workload use all the resources to recover the latency to its normal state. When the previously mentioned conditions are not active, then Hybrid PRM signs the sub-controller to increase CPU resource of BE workload. Sub-controller loop runs for every 2 seconds and increases one core every time. If the previously mentioned conditions are active then sub-controller is disabled and will not run until it receives a signal from Top-level controller.

Code listing 3.1: Pesudo code for Heracles Top-level Controller

```
1  while True:
2      slack = ( Qos\_target - latency ) / QoS\_target
3      if slack < 0:
4          SuspendBE()
5          CoolDown()
6      elif load > 0.85:
7          SuspendBE()
8      elif slack < 0.80:
9         EnableBE()
10     elif slack < 0.10:
11         HoldBEResources()
12     elif slack > 0.10:
13         SignalSubController() -> IncreaseBECore()
14     sleep(15)
```

3.3.1 Tuning in Hybrid PRM

Three modifications has been made to Hybrid PRM to improve machine utilisation:

- Hybrid PRM increases resources equivalent to one core every time it detects an opportunity. In this variant, the total available CPU resources have been divided into 50 levels. This is similar to the method followed by Latency-aware PRM.
- When the load on the LC workload is above 95% of its maximum capacity, the BE workload is suspended and when the load is below 90%, resource allocation to the BE workload is started.
- When there is a latency spike, slack becomes negative. In such case, cool down time has been changed to 3 minutes from 5 minutes.

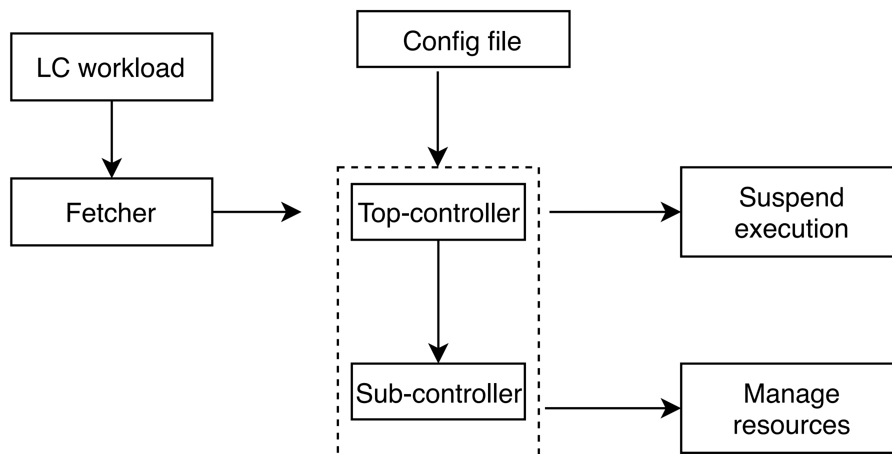

Figure 3.2: High-level overview of Hybrid PRM Implementation

Chapter 4

Implementation

This chapter describes the implementation details of the variants of PRM discussed in Chapter 3.

4.1 Intel PRM Implementation

Most of the Intel PRM is developed with Python programming language and some parts of it use Go and C programming languages. Intel PRM uses popular python libraries such as numpy [23], pandas [24], scipy [25] and scikit-learn [26] for data pre processing and statistical analysis. Intel PRM uses the concept of docker containers to monitor and control shared resources of workloads running in docker containers. In the docker environment, applications are packaged into a standard form along with all the required dependencies. The packaged formed is called a container [27].

4.1.1 Prerequisites

As discussed previously, Intel PRM relies mainly on Intel RDT technology to collect platform metrics. At the time of conducting this study, Linux is the only supported operating system that can run Intel RDT software. To run Intel PRM, a copy of Intel RDT must be installed on the machine. Intel PRM requires that copies of Python 3.6.x interpreter, a Go language compiler and a GCC compiler be installed on the machine. We must also install previously mentioned Python libraries and Docker on the machine in order to make Intel PRM work.

4.1.2 Implementation of Step One

Initially, the latency-critical workload must be monitored and platform metrics of the workload must be recorded in order to build a best case threshold model for the workload. Intel PRM uses a workload configuration file provided by the user to identify the workloads. The configuration file is a Json file and it must include details such as name of the workload, requested cpu count for the workload and the type of the workload

which indicates if the workload is latency-critical or best-effort. The name of the workload must be the name of the docker container in which the workload is running. A sample workload configuration file is given in Code listing 4.1.

Code listing 4.1: A sample workload configuration file

```
{
    "workload1": {
        "cpus":8,
        "type": "latency_critical"
    },
    "workload2": {
        "cpus":2,
        "type":"best_efforts"
    }
}
```

Intel PRM monitors and records the platform metrics of all the workloads given in the workload configuration file. However, we observed that the threshold model is built using the metrics of the latency-critical workload. To make Intel PRM monitor the workloads, workloads must be run as docker containers. For each workload, one docker container is created and for the container, a cgroup is created which groups all processes, including child processes of the workload. Intel PRM uses the created cgroup to track the group of processes that spawned from the workloads and collects platform metrics for every 18.5 seconds using the main-monitor. Sub-monitor reads cpu utilisation of the workloads from cpuacct subsystem. cpu utilisation metric exposed by cpuacct subsystem is in nanoseconds.
Intel PRM converts the cpu utilisation of the workloads into a percentage value using Code listing 4.2. It reads the total cpu usage of the system using /proc/stat file system. It reads the cpu utilisation of cgroup using cpuacct.usage file. It calculates the usage every 2 seconds by taking a difference of two consecutive readings. Difference is calculated for total usage of the system and usage of the cgroup. Conversion of time metric into a percentage value is given in **line 24** of 4.2. This process is carried out for each workload and corresponding cpu utilisation is recorded as a percentage value. Main-monitor collects performance metrics such as cycles, instructions using a utility called 'pgos'. The utility gives command line access to Intel RDT. Using the utility as an intermediary, Intel PRM collects metrics using Intel RDT. Pgos collects metrics for each cgroup which contains all the processes belonging to the workload that we want to monitor. The recordings from monitors are stored independently in two csv files. The variables of Code listing 4.2 is given below:

Code listing 4.2: Python code to convert cpu usage of a container into a percentage value

```
def update_cpu_usage(self):
    """ calculate cpu usage of container """
    try:
        total_usage = 0
        system_usage = 0
        cpu_util = 0.0
        cur = time.time() * 1e9

        with open("/proc/stat") as f:
            stats = [int(e) for e in f.readline().split()[1:]]
            system_usage = sum(stats) * 1e9 / 100

        cgroup_stat = path_join('/sys/fs/cgroup/cpu', self.parent_path,
                                self.con_path, 'cpuacct.usage')

        with open(cgroup_stat, 'r') as fi:
            total_usage = int(fi.read().strip())

        cpu_delta = total_usage - self.cpu_usage
        system_delta = system_usage - self.system_usage
        cpu_no = multiprocessing.cpu_count()

        if cpu_delta > 0 and system_delta > 0:
            cpu_util = (float(cpu_delta) / system_delta) * cpu_no * 100

        self.timestamp = cur
        self.utils = cpu_util
        self.cpu_usage = total_usage
        self.system_usage = system_usage
        except (ValueError, IOError):
            pass
```

- system_usage: system_usage is the CPU time spent on performing different kinds of work in the system.
- self.parent_path: For each cgroup one docker container is created and /sys/fs/cgroup/cpu/docker contains list of all the containers running in the system.
- self.con_path: Indicates docker container name for which we want to monitor CPU usage
- cgroup_stat: reads CPU time used by the workload since it started.
- cpu_no: Indicates total availble cores on the machine.
- cpu_util: Indicates the cpu utilisation of the workload as a percentage value.

After gathering the required data, data must be analysed offline to build a best case threshold model that can help us detect contention on shared resources. scikit-learn is a machine learning library which consists of tools required for performing statistical analysis. Intel PRM uses Gaussian Mixture model provided by scikit-learn to analyse the collected data. It initially pre-processes collected data using pandas and numpy, which are data processing libraries, and then uses the processed data to build a Gaussian Mixture Model. The outcome of the analysis is a Json file which contains best case performance metrics of the LC workload. The two metrics that are relevant for the study are: a CPI threshold of LC workload and maximum cpu utilisation of LC workload. Using

these metrics, Intel PRM decides how to orchestrate CPU resources when workloads are co-located.

4.1.3 Implementation of Step Two

This section discusses the implementation details of contention detection and resource controlling. After the threshold model is generated, the model serves as a basis for contention detector to detect resource contention from the BE workload. When the LC workload and the BE workload are running together in a server, Intel PRM monitors the performance metrics and cpu utilisation of the workloads. If current CPI of the LC workload is greater that the best case CPI, it implies that the LC workload is being affected because of some resource interference from the BE workload. This is also referred as a resource contention. Another step taken by Intel PRM to avoid performance degradation of LC workload is by checking its cpu utilisation using the sub-monitor. The sum of cpu utilisation of LC workload and BE workload must not be greater than cpu utilisation threshold determined during threshold model building. If the aforementioned condition is violated, Intel PRM considers that there is a resource interference from the BE workload.
Intel PRM initially sets cpu shares of the LC workload to 200000 and of the BE workload to 2 using relative tuning. By setting such values, it is restricting the relative share of the BE workload to the lowest value and thus giving high share of CPU time to processes of the LC workload. Along with relative tuning, Eris also uses ceiling enforcement tuning by setting cpu quota of the BE workload to integer value of 1000 micro seconds using cpu.csf_quota_us parameter. This is the lowest limit of ceiling enforcement parameter. By default, Intel PRM divides the CPU resources into 20 levels and increases CPU resource of the BE workload by 1 level when corresponding condition is met. Increasing each level increases cpu quota of BE workload by a constant step which is determined by dividing maximum cpu utilisation of LC workloads by total number of levels which is 20. If there is a resource contention, then the CPU time of the BE workload is set to minimum again by setting ceiling enforcement parameter to 1000. In case where resource level of BE workload reaches 20, then Intel PRM sets the ceiling parameter of BE workload to -1, which is the default value. It implies that Intel PRM has handed resource allocation of the BE workload to CFS scheduler and CFS tries to maintain fairness in allocating CPU resources to both the workloads, that is LC and BE workload.
To increase resources Intel PRM uses another safeguard. When contention detector does not detect any contention from sub-monitor, it increments a cycle count value by 1 which is just a variable. From the source code, we identified that cycle threshold is hard-coded as 7. One resource level is increased whenever cycle count is equal to cycle threshold. Cycle count is reset to zero when contention detector detects contention. It implies that, one resource level is increased for every 14 seconds if Intel PRM detects no contention [1]. Algorithm for detecting if the BE workload is over utilising the cpu resources is given in Code listing 4.3. The algorithm also contains condition for increases CPU resource of

[1]It is to be noted that Intel PRM has mechanisms similar to this to increment other resource levels such as LLC. It contains LLC count and different LLC threshold values

the BE workload.

Code listing 4.3: Algorithm for detecting if the BE workload is overutilising the cpu resources

```
while True:
    margin = 100000 * 0.5
    cyc_count = 0
    cyc_threshold = 7
    exceed = ( LC_utilisation + BE_utilisation ) * 1000 + margin > LC_max_utilisation
    if exceed:
        BE_cpu_quota = cpu_quota_min
    else:
        cyc_count = cyc_count + 1
        if cyc_count >= cyc_threshold:
            cyc_count = 0
            BE.increase_resource_level()
```

- **margin**: margin is the value equal to the cpu quota of half a logical processor
- **cyc_count**: cycle count used to check if a resource level must be increased or not.
- **cyc_threshold**: cycle threshold.
- **LC_utilisation**: CPU utilisation of LC workload running on the machine.
- **BE_utilisation**: CPU utilisation of BE workload running on the machine.
- **LC_max_utilisation**: The maximum recorded cpu utilisation of the LC workload.
- **BE_cpu_quota**: cpu quota given to a BE workload.
- **cpu_quota_min**: indicates the lowest possible cpu quota that could be set which is 1000 microseconds.
- **BE_increase_resource_level()**: Represents a function that increments cpu quota level of BE workload by 1 level.

4.2 Latency-aware PRM Implementation

Implementation of Latency-aware PRM can be categorised into two steps. In step one, the QoS target which is a latency target must be determined. It is usually determined by the developers of the workload. After obtaining the target, it must be specified in a configuration file which is a JSON file. In step two, latency-critical workload and best-effort workload are co-located on the same server. Fetcher is a component which collects latency value from the LC workload and stores it. QoS-monitor polls for latency from Fetcher. Detector periodically reads current latency from QoS-monitor and latency target value from configuration file and decides if there is a performance degradation in latency-critical workload. Similar to Intel PRM, when started this variant sets the cpu shares of LC workload to 200000 and BE workload to 2 using relative tuning and sets cpu quota of BE workload to lowest value which is 1000. For every 18.5 seconds, depending on if the latency is in desired range or not, detector signals resource manager to adjust resources. If the latency is above the target, the CPU resources are again set to minimum by setting cpu.shares and cpu.cfs_quota to minimum. In case the latency value is below the target then detector signals resource manager to increase CPU resource level of BE workload by 1. In this variant, each level is equal to the total available CPU resources in percentage divided by 50.

4.3 Hybrid PRM Implementation

Similar to previous variants, this variant when started sets the CPU resources of the BE workload to a bare minimum by tuning relative and ceiling enforcement parameters. It sets the cpu.shares parameter of the LC workload to 200000 and of the BE workload to 2. Additionally, it also sets cpu quota of the BE workload to 1000 by tuning cpu.cfs_quota_us parameter. When the safeguards of this variant are not active then it increases CPU resources of the BE workload by increasing cpu quota of the workload. For example, setting cpu.cfs_quota_us parameters of the cgroup to which the workload belongs to the value equal to cfs period of the machine makes the workload use CPU resources equal to one core. For every two seconds, Sub-controller increases the resources of the BE workload by 1 core. Sub-controller stops only when it receives a signal from Top-level controller that BE workload is affecting the latency of the LC workload.

Chapter 5

Evaluation

This chapter presents the experimentation methodology that is followed to evaluate the different variants of the platform resource manager. The methodology contains details about the machine and the workloads used for this study. Additionally, it discusses the process followed to determine the maximum throughput of the latency-critical workload chosen for this study and presents the evaluation metrics used to evaluate each PRM. Later, it presents the results obtained when the CPU resources are controlled by different variants.

5.1 Experimentation Methodology

5.1.1 Hardware

We perform the evaluation of different variants of PRM on the NTNU EPIC compute cluster. Each node runs Linux kernel 3.10 and contains two Intel Xeon E5-2695v4 sockets that together comprise 36 CPUs. Each core is capable of frequency scaling from 1.20 GHz to 2.00 GHz with steps of 0.1 GHz. Each socket has 1.2 MB, 4.5 MB, and 45 MB of L1, L2 and shared last level cache, respectively. The server contains 128 GB of DDR4-2400 GHz RAM. Hyperthreading was disabled as in most production servers.

5.1.2 Workloads

This section discusses the workloads used for the study. Memcached is chosen as a LC workload and stress-ng is chosen as a BE workload to stress CPU resources.

Memcached

Cloudsuite is an open-source benchmark suite for cloud services [28]. Data caching is one of the applications provided by Cloudsuite benchmark. Data caching is a Memcached data caching server which uses twitter data set and simulates Twitter data caching server behaviour. Memcached is an open-source distributed caching system [29]. It acts as a store for small chunks of data which is extracted from the results of database calls or

API calls. Memcached is not a user-facing service, however, it is used by many important web services and has strict QoS targets. Motivations behind choosing Memcached as a LC workload for the study are listed below:

- Memcached is very sensitive to resource interference from co-located workloads [9]. This makes the job of simulating resource interference easy when co-located with a BE workload.
- Memcached has strict quality of service targets. Data caching workload from Cloudsuite benchmark assumes that 95% of the requests are responded or serviced within 10ms.
- Memcached is not directly a user-facing-service. However, it is an important service used in the back-ends of many large applications and deployed on thousands of servers [9]. We believe that knowledge gained from this study can be used in real world scenarios.

Stress-ng

Stress-ng is a tool used to stress test a computer in various ways. It can be used to stress different resources of a computer such as CPU, cache etc. For this study, stress-ng is used as a BE workload. We use stress-ng to put stress on CPU resources of the machine.
To make Intel PRM monitor and control the resources used by the workloads, the workloads must be formatted into docker images before they are run on the machine. Clousuite benchmark provides docker images for Memcached client and Memcached server. Memcached client sends requests to the Memcached server to access the server's data.

5.1.3 Determining Maximum throughput of Memcached

Maximum throughput is the maximum number of requests that a Memcached server can serve without violating QoS policies. The target QoS is that 95-th percentile tail latency must be under 10ms. To identify maximum throughput that can be achieved using our hardware we ran Memcached server by varying load, that is, the number of requests sent through Memcached client. We recorded 95-th percentile tail latency under different load.
The experimental set up to determine the maximum throughput of Memcached server consists of one instance of Memcached server and one instance of Memcached client. Of the available 36 cores on our hardware, we used 17 cores of socket 0 to run Memcached server with 17 threads, one core for each thread. We used 18 cores of socket 1 to run Memcached client with 18 threads. We left 1 core of socket 0 idle to run Intel PRM on it. This was done to not let any overhead from Intel PRM interfere with the Memcached server.
Memcached client provided by Cloudsuite allows us to enter the number of requests that must be sent by client. To determine the maximum throughput, we simulated a time varying load. The difference between every two consecutive loads is constant. The load is increased by constant load factor for every 200 seconds. Starting with the minimum load of 50k requests per second, load is increased by change factor until the QoS requirements

of the Memcached are violated. In our case, QoS requirements are violated when the 95-th percentile latency is above 10 milliseconds.
When performing experiment we observed a latency spike when the rps is in between 450k and 500k. This is represented using Figure Figure 5.1. To determine the maximum throughput accurately we conducted another experiment by varying load by constant factor. This is represented using Figure 5.2. From the experiment, we observed that, for our hardware set up Memcached can serve 450k requests per second while respecting QoS target. Increase in load after 450k rps would result in QoS violation. Even though we varied load by constant factor, we can see that load curve is not uniform in Figure 5.2. This because of the queuing of requests. For example, when we configure Memcached client to send 430k requests per second, Memcached server does not serve exactly 430k requests. We observed there if a variation in number of requests configured to be sent and number of requests processed.

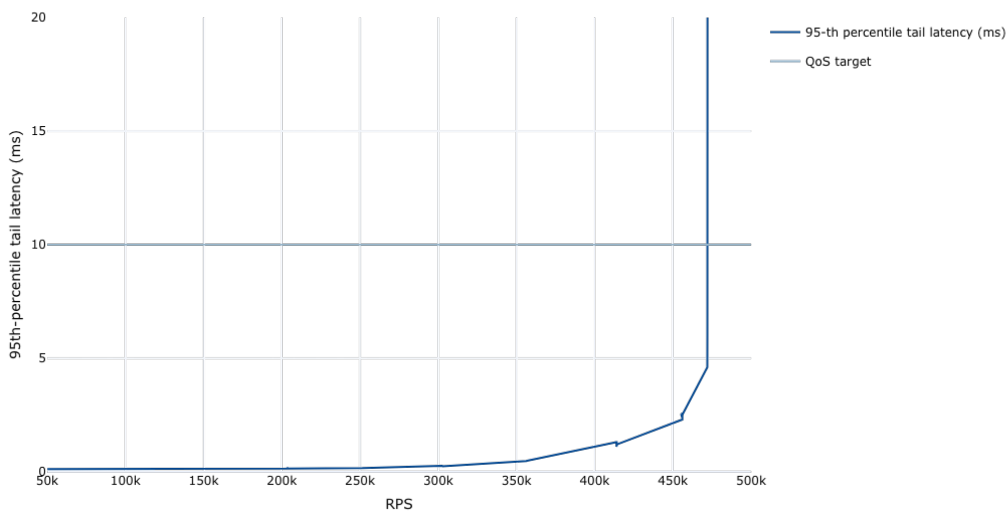

Figure 5.1: Variation in 95-th percentile tail latency of Memcached with varying load.

5.1.4 Workload co-location

During the whole study the LC workload is made to run as mentioned in Section 5.1.3. To evaluate different versions of PRMs, we ran stress-ng as a BE workload and stressed 8 cores of the machine completely. Stress-ng was co-located on the same socket as Memcached server. To pin Memcached server and stress-ng to socket 0 a Linux tool called taskset is used [30]. Taskset helps in running the workloads on the selected set of cores. The idea behind this set up is to simulate CPU resource interference by BE workload by letting the workloads run on the same socket. As a consequence, the interference cause by BE workload affects the performance of the LC workloads which in turn leads to QoS violation.

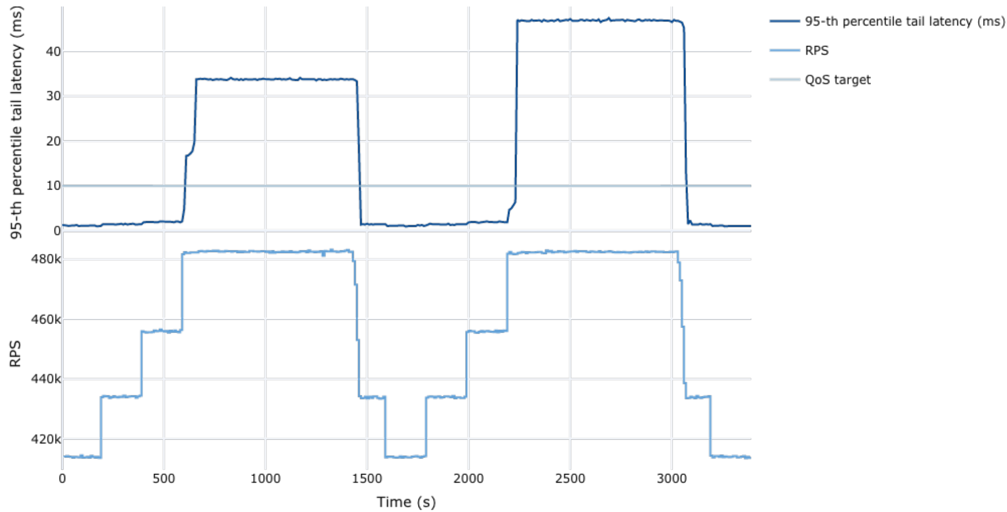

Figure 5.2: Variation in 95-th percentile tail latency of Memcached with varying load-2.

5.1.5 Evaluation Metrics

We evaluate different variants of PRM using two metrics: QoS-guarantee and BE throughput. QoS-guarantee indicates the percentage of samples that met QoS target. BE throughput indicates the number of instructions that are executed when the resources are controlled by a PRM. We use perf tool [31] to determine the throughput of Stress-ng. We compute the throughput of the BE workload when resources are controlled by different variants of PRM and normalise it to the throughput of the workload when it is running alone. We also report the percentage of CPU resources used by the workloads to explain the results. Additionally, we report CPI of Memcached to highlight the cases where CPI fails to indicate the latency behaviour of Memcached.

5.2 Results

In this section we present the results obtained by evaluating various variants of PRM: Intel PRM, Latency-aware PRM, Hybrid PRM, Tuned Hybrid PRM. The summary of the results are given in Table 5.1. In the following sections, each subsection explain results of each variant.

Table 5.1: Summary of QoS-guarantee of Memcached and Throughput of Stress-ng when scheduled by different PRM variants

PRM variant	QoS guarantee	Throughput
Intel PRM	100%	4.6%
latency-aware PRM	89.2 - 97%	56.2%
Hybrid PRM	100%	14.7%
Hybrid PRM tuned	100%	54.0%

5.2.1 Intel PRM Results

Figure 5.3 presents the impact of co-location on the tail latency of the Memcached when CPU resource are controlled by Intel PRM. Intel PRM provides 100% QoS-guarantee which means all the samples are under QoS-target of 10 ms. Intel PRM makes scheduling decisions based on CPI metric. As a result, it sets CPU resources of the BE workload to minimum whenever current CPI of Memcached is greater than determined threshold CPI. Intel PRM helps in obtaining the throughput of only 4.6% compared to the throughput when Stress-ng is run alone. At all loads there are no violations in meeting QoS targets. Even at lowest load in the graph, Memcached still uses 65% of CPU resources.

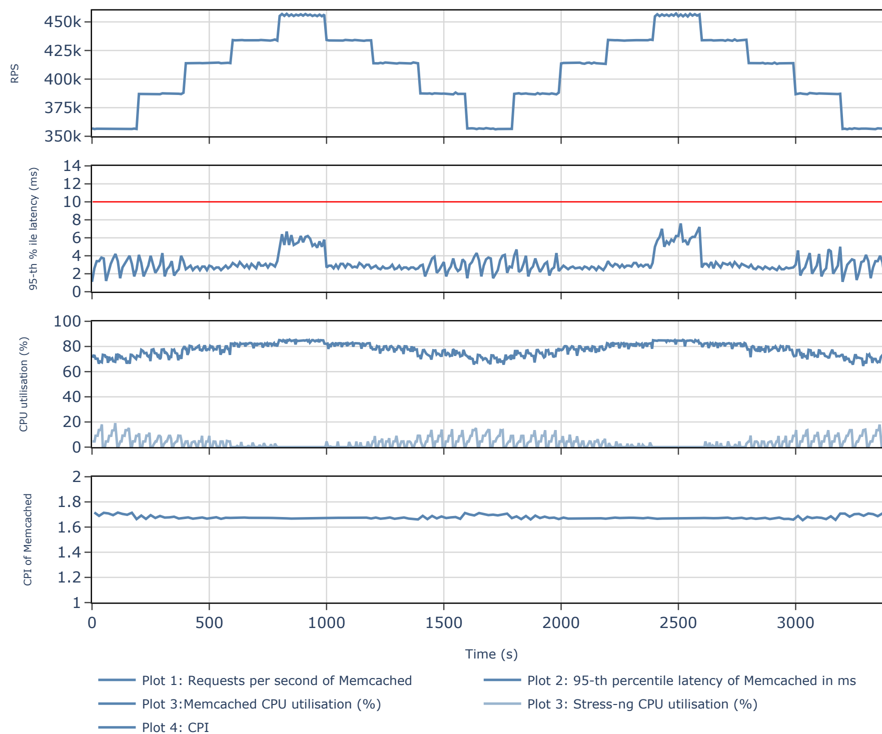

Figure 5.3: Variation in 95-th percentile tail latency of Memcached when CPU resources are controlled by Intel PRM

5.2.2 Latency-aware PRM Results

This variant makes scheduling decisions based on application level latency of the LC workload. Figure 5.4 presents the impact of co-location on tail latency of Memcached when CPU resources are controlled by Latency-aware PRM. Initially, CPU resources of the BE workload are set to minimum. From the figure, it can be seen that as QoS target

is met, the resource manager starts increasing CPU resources allocated to Stress-ng. As a result, CPU utilisation of Stress-ng increases. When at low load, Memcached needs only half of the total available CPU resources. This variant, uses this opportunity to improve the throughput of Stress-ng. When the PRM detects that the tail latency is above the target, which is 10 ms, it sets the CPU resources of Stress-ng to minimum. But this happens only after the latency has crossed the target. This is a serious drawback of this variant. It does not have any safeguard to prevent this. However, it improves machine utilisation by running Stress-ng when the latency is low. From the experimentation, we observed that the throughput of Stress-ng reaches upto 56.2% when CPU resources are handled by this variant. Another major observation from the figure is that, when the load is low, Memcached only needs half of the CPU resources to meet QoS-target. Comparing this to the results of Intel PRM, Intel PRM was allowing Memcached to use more resources that it needed. From this observation we can conclude that when co-locating workloads application level latency is more reliable metric to improve machine utilisation than throughput based metric such as CPI. However, we observed that during the times of high load this variant fails to provide 100 % QoS-guarantee. During the times of high load there is a possibility of experiencing latency spikes which may be create bad experience to end-users. Figure 5.5 shows that when the load is at maximum capacity there is a possibility of experiencing huge latency spikes.

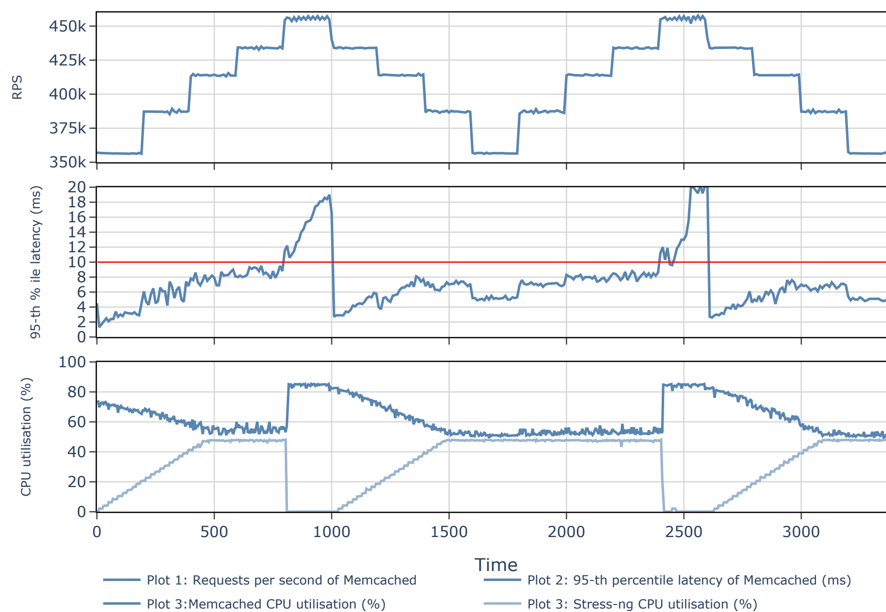

Figure 5.5: Variation in 95-th percentile tail latency of Memcached when CPU resources are controlled by Latency-aware PRM - 2

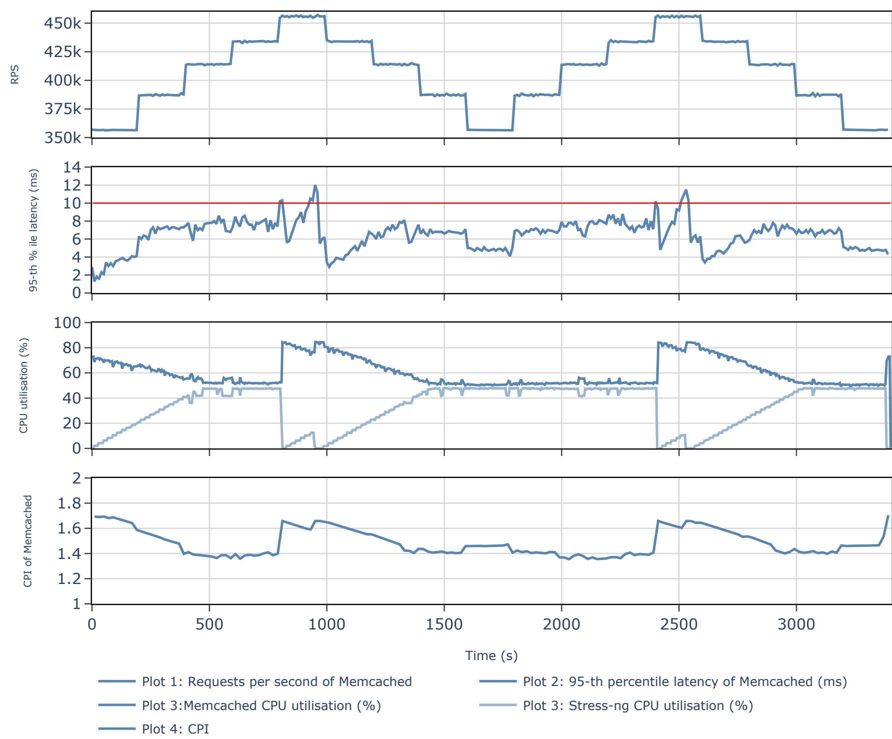

Figure 5.4: Variation in 95-th percentile tail latency of Memcached when CPU resources are controlled by Latency-aware PRM

5.2.3 Hybrid PRM Results

This variant fixes the drawback of the previous variant. When the load is high previous variant fails to handle the tail latency. Hybrid PRM suspends the Stress-ng workload when the load on Memcached is above 85% of its maximum capacity and enables it only when the load on the Memcached is below 80% of its maximum capacity. Figure 5.6 shows the impact of co-location on tail latency of Memcached when resources are controlled by Hybrid PRM. This variant achieves 100% QoS-guarantee but when resources are controlled by this variant the throughput of Stress-ng is only 14.7%. This is because the experimentation was done on small scale. This variant enables execution of Stress-ng when the load on the Memcached reaches 360k rps.

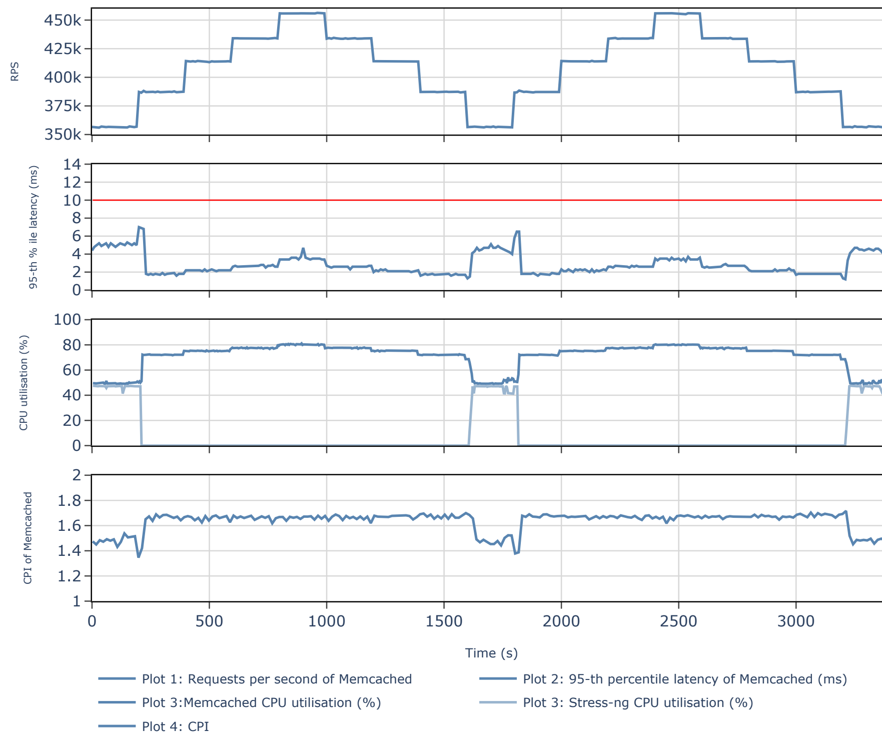

Figure 5.6: Variatation in 95-th percentile tail latency of Memcached when CPU resources are controlled by Hybrid PRM

5.2.4 Tuned Hybrid PRM Results

By tuning parameters of Hybrid PRM we achieve the QoS-guarantee of 100% and the throughput of 54.0%. Figure 5.7 shows the results obtained when resources are con-

trolled by this variant. This variant suspends the execution of the BE workload when the load on the Memcached is above 95% of its maximum capacity and enables it only when load is below 90%. When the load is below 90%, it polls for latency slack value and uses the opportunity to execute Stress-ng. Subplot 3 of the figure shows that this variant attempts to utilise the machine to the maximum extent possible. A major observation of the graph is the utilisation of latency slack to guide scheduling decisions. This variant checks that latency-critical barely meets the QoS target and exploits other resources to run best-effort workload.

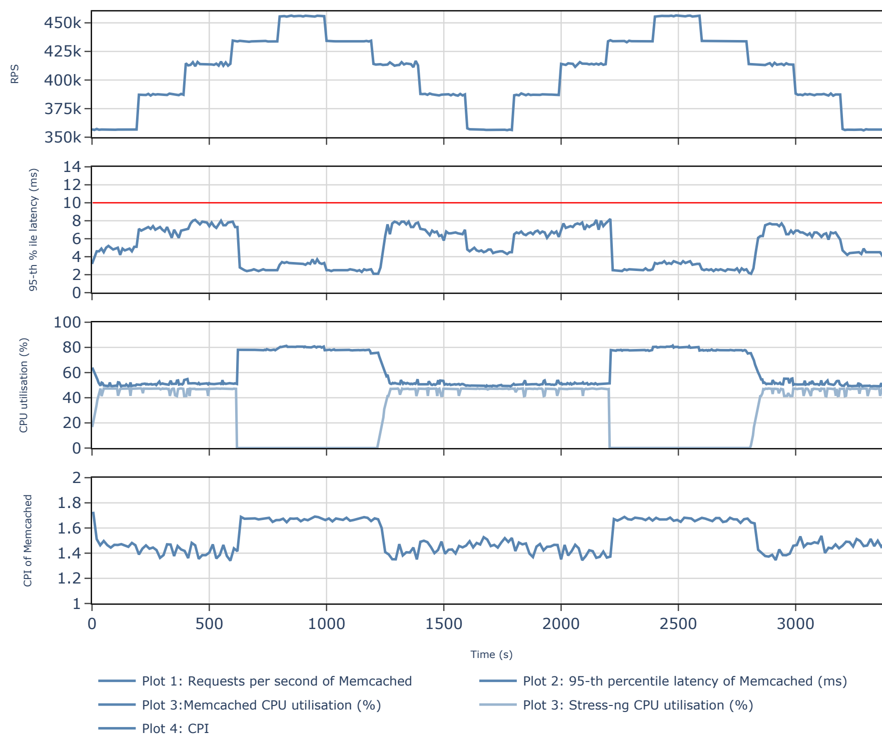

Figure 5.7: Varitation in tail latency of Memcached when resources are controlled by Hybrid PRM tuned

5.3 Discussion

Four different variants have been evaluated during this study. Of all the variants, Intel PRM over provisions resources to the latency-critical workload. Applications like Memcached can run with moderate resources and still meet QoS target. Intel PRM fails to detect this and does not use this opportunity to co-locate best-effort workloads. How-

ever, it always makes sure that performance of the latency-critical workload is healthy. On the other hand, variants that made scheduling decisions based on application level latency were able to co-locate best-effort workload when the latency-critical workload did not need resources. Latency-aware PRM is a naive approach that did not have any safeguards to protect the LC workload from violating service level objectives (SLO). Hybrid PRM's strategy of suspending the best-effort workload based on the current load and current latency of the Memcached improves machine utilisation greatly while respecting QoS policies of the Memcached.

Chapter 6

Conclusion

In this work we explored the strategies used by Intel PRM to make scheduling decisions when a best-effort workload is co-located with a latency-critical workload. We show that cycles per instruction (CPI) is not a reliable metric to determine latency behaviour of the latency-critical workload and we show that using CPI as a QoS metric does not yield great results in improving machine utilisation. We introduce different variants of PRM which are based on the methods of Intel PRM but which make scheduling decisions based on application level latency of the latency-critical workload. We implement core management algorithm from Heracles into Intel PRM and with simple tuning of parameters we show that the throughput of co-located workload can be improved significantly. We conduct experiments and show that Hybrid PRM, which uses methods from Intel PRM and Heracles, achieves 100% QoS-guarantee and achieves a best-effort workload throughput of 54.0%.

Bibliography

[1] L. A. Barroso, J. CLidaras and U. Holzle, 'The datacenter as a computer an introduction to the design of warehouse-scale machines second edition', 2013.

[2] L. Barroso and U. Holzle, 'The case for energy-proportional computing', *Computer*, vol. 40, pp. 33–37, Jan. 2008. DOI: 10.1109/MC.2007.443.

[3] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan and C. Kozyrakis, 'Heracles: Improving resource efficiency at scale', *SIGARCH Comput. Archit. News*, 2015.

[4] J. Dean and L. A. Barroso, 'The tail at scale', *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408794. [Online]. Available: https://doi.org/10.1145/2408776.2408794.

[5] Uber, *Peloton*, Available at:https://github.com/uber/peloton, (Accessed 4-June-2020).

[6] Intel, *Intel platform resource manager*, Available at:https://github.com/intel/platform-resource-manager, (Accessed 13-May-2020).

[7] R. Nishtala, P. Carpenter, V. Petrucci and X. Martorell, 'Hipster: Hybrid task manager for latency-critical cloud workloads', pp. 409–420, 2017.

[8] O. Bilgir, M. Martonosi and Q. Wu, 'Exploring the potential of cmp core count management on data center energy savings', May 2012.

[9] J. Leverich and C. Kozyrakis, 'Reconciling high server utilization and sub-millisecond quality-of-service', 2014.

[10] 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', pp. 248–259, 2011. [Online]. Available: https://doi.org/10.1145/2155620.2155650.

[11] C. Marco and B. Daniel P., *Understanding the linux kernel*, 2005.

[12] J. M, *Inside the linux 2.6 completely fair scheduler*, Available at:https://developer.ibm.com/tutorials/l-completely-fair-scheduler/, 2018, (Accessed 28-Mar-2020).

[13] H. Yang, A. Breslow, J. Mars and L. Tang, 'Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers', ISCA '13, pp. 607–618, 2013. DOI: 10.1145/2485922.2485974. [Online]. Available: https://doi.org/10.1145/2485922.2485974.

[14] R. Xu, S. Mitra, J. Rahman, P. Bai, B. Zhou, G. Bronevetsky and S. Bagchi, 'Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads', pp. 146–160, 2018.

[15] W. Cui, D. Richins, Y. Zhu and V. J. Reddi, 'Tail latency in node.js: Energy efficient turbo boosting for long latency requests in event-driven web services', VEE 2019, pp. 152–164, 2019. DOI: 10.1145/3313808.3313823. [Online]. Available: https://doi.org/10.1145/3313808.3313823.

[16] O. Foundation, *About node.js*, https://nodejs.org/en/about/, (Accessed 26-06-2020).

[17] C. Delimitrou and C. Kozyrakis, 'Qos-aware scheduling in heterogeneous datacenters with paragon', *ACM Trans. Comput. Syst.*, vol. 31, no. 4, Dec. 2013, ISSN: 0734-2071. DOI: 10.1145/2556583. [Online]. Available: https://doi.org/10.1145/2556583.

[18] Kubernetes, *Kubernetes documentation*, Available at:https://kubernetes.io/docs/home, (Accessed 7-June-2020).

[19] P. Menage, *Cgroups*, Available at:https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, (Accessed 4-Apr-2020).

[20] R. hat, *Chapter 1. introduction to control groups (cgroups)*, Available at:https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01, (Accessed 4-Apr-2020).

[21] R. hat, *Cpu*, Available at:https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpu, (Accessed 4-Apr-2020).

[22] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso and C. Kozyrakis, 'Towards energy proportionality for large-scale latency-critical workloads', pp. 301–312, 2014.

[23] NumPy, *Numpy*, Available at:https://numpy.org/, (Accessed 13-May-2020).

[24] pandas, *Pandas*, Available at:https://pandas.pydata.org/, (Accessed 13-May-2020).

[25] Scipy, *Scipy*, Available at:https://www.scipy.org/, (Accessed 13-May-2020).

[26] scikit-learn, *Scikit-learn machine learning in python*, Available at:https://scikit-learn.org/stable/, (Accessed 13-May-2020).

[27] Docker, *What is a container? a standardized unit of software*, Available at:https://www.docker.com/resources/what-container, (Accessed 13-May-2020).

[28] Cloudsuite, *A benchmark suite for cloud services*, Available at:https://www.cloudsuite.ch/, (Accessed 26-06-2020).

[29] Memcached, *What is memcached?*, Available at:http://memcached.org/, (Accessed 26-06-2020).

[30] taskset, *Taskset(1) - linux man page*, Available at:https://linux.die.net/man/1/taskset, (Accessed 26-06-2020).

[31] *Linux kernel profiling with perf*, Available at:https://perf.wiki.kernel.org/index.php/Tutorial#Counting_with_perf_stat, (Accessed 20-June-2020).