Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Jonas Sandberg, Michael Moen Allport

Q-PRM

A QoS Aware Resource Manager for Colocated Services

Master's thesis in Computer Science
Supervisor: Björn Gottschall
Co-supervisor: Rajiv Nishtala
July 2021

NTNU
Norwegian University of
Science and Technology

Jonas Sandberg, Michael Moen Allport

Q-PRM

A QoS Aware Resource Manager for Colocated Services

Master's thesis in Computer Science
Supervisor: Björn Gottschall
Co-supervisor: Rajiv Nishtala
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Problem Statement

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 the Quality of Service (QoS) target. Normally, it is necessary to sacrifice server efficiency in order to meet such QoS targets, e.g. by reserving an entire server for a latency-critical service even when this means under-utilizing the server. Recent research has explored the idea of colocating such services with additional workloads to increase server utilization, which translates to higher energy and cost efficiency. The question is how to maximize the throughput of additional work, while at the same time ensuring that the QoS target is met as a main priority.

The class of software that has been developed to solve this problem at the single-server level is known as resource managers. Most resource managers depend on some collection of system performance metrics in order to function, but more recently some (e.g. Octopus-Man, PARTIES) have been developed which instead directly use the measured QoS, known as QoS-aware resource managers. Our goal is to evaluate the effectiveness of the QoS-aware approach compared to the traditional metrics-based approach. In order to do this, we should do the following:

1. Choose at least one benchmark from the CloudSuite benchmark suite as our latency-critical service.
    a. Set up the benchmark on a multi-node server. The client and server parts of the benchmark should be isolated from one another by running on separate CPU nodes.
    b. Run the benchmark, see how the QoS varies with the RPS/load, and determine the QoS target of the service.
    c. To improve our understanding of its behavior, collect additional performance statistics, such as CPU, memory and power usage.
2. Choose a best-effort task. Colocate it with the latency-critical task, and observe the impact on the QoS.
3. Set up the Intel Platform Resource Manager.
    a. See whether the resource manager is able to improve QoS/throughput.
    b. Figure out how the resource manager works.

4. Implement a QoS-aware resource manager.
    a. Modify the CloudSuite benchmark so that the client transmits the measured QoS to the resource manager.
    b. Modify the Intel Platform Resource Manager to make it QoS-aware.
    c. Implement one or more QoS-based controller algorithms.
5. Evaluate the QoS-aware PRM against the standard PRM.

Abstract

The growth of cloud-computing has lead to a large number of latency-sensitive workloads being hosted in the cloud. These workloads are often user-facing, which makes conventional power-saving techniques hard to fully utilize without adverse impact on the user's experience. Resource managers attempt to alleviate this problem by using real-time metrics in order to control workload resources. By balancing resource requirements of different workloads, we might be able to leverage those measurements to meet latency requirements. In this thesis we improve upon a conventional resource manager, the Intel Platform Resource Manager, by making it aware of workload latency. Our approach yielded a 92.4% QoS guarantee, compared to 37.8% for the unmodified Platform Resource Manager.

Sammendrag

Veksten innenfor bruk av skytjenester har ført til at tjenester som er følsomme for forsinkelser i større grad bor i skyen. Disse tjenestene er ofte rettet mot brukere, et faktum som gjør konsvensjonelle energibesparingsmetoder krevende å utnytte uten å forverre sluttbrukerens opplevelse. Ressurskontrollere prøver å lette på dette problemet ved å ta sanntidsmålinger av tjenestenes ressursbruk. Disse målingene kan brukes til å balansere ressursbruk mellom tjenester for å møte minimumskravene for forsinkelse. I denne oppgaven forbedrer vi en eksisterenede, konvensjonell ressurskontroller, Intel Platform Resource Manager, ved å gjøre den bevisst på tjenestens forsinkelse. Våre endringer førte til en 92.4% QoS garanti, sammenlignet med 37.8% for den umodifiserte ressurskontrolleren.

Preface

This thesis has been written during Spring 2021 in order to fulfill the requirements for our Master of Technology degree at NTNU Gløshaugen. The field was completely unknown to us before starting this thesis, and was at times both overwhelming and challenging. It is for this reason we wish to especially thank our supervisors, Björn Gottschall and Rajiv Nishtala, who helped steer us in the right direction and give sound advice. We are very grateful for their patience and time during the entire process.

Michael Moen Allport & Jonas Sandberg — July 2021

Contents

Figures

Tables

Chapter 1

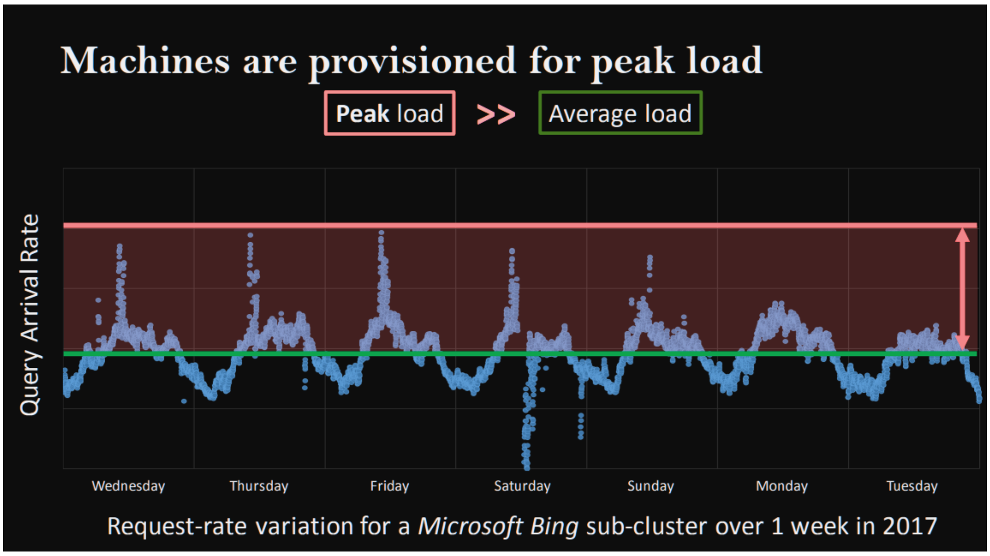
Introduction

Cloud computing has in recent years seen mass-adoption by much of the IT-industry, and has fundamentally altered the way companies interact with digital technologies. In the past, accessing digital services required a big initial investment in server and network infrastructure, as well as long-term costs like service technicians and maintenance staff [1]. Furthermore, scaling server infrastructure require more trained personell, and *time* [2]. Cloud providers often offer "pay-on-demand" models that allow businesses to keep their costs proportional to their web traffic or storage used. Cloud providers themselves benefit from economies of scale, leveraging the fact that warehouse-scale-computers (WSCs) can efficiently service multiple customers and workloads simultaneously. Furthermore, access to a greater number of servers also yield more parallellization options - a root server might delegate smaller subtasks to thousands of servers concurrently [3]. This flexibility and ease of scale have resulted in a 62% annual increase in cloud traffic from 2010 to 2016 [4].

However, server utilization remains low even in large-scale operations - servers mostly operate between 10-50% CPU utilization [5]. This is far from ideal, because servers are not *energy proportional*, i.e. consumption of energy is not proportional to the actual work being done [6]. In fact, an idling server uses 50% of it's peak power usage. There are two reasons why WSCs fall victim of low utilization: One is by design - that in case another WSC experiences an unplanned outage, it's imperative to have enough capacity to handle the incoming load. Another reason is to have enough capacity to serve the natural fluctuation in usage, usually corresponding to diurnal cycles.

A seemingly reasonable way to solve this problem might be to use power-saving techniques when the server load is low. Mobile devices have been applying these kinds of techniques successfully for about a decade [5]. The load pattern of mobile devices typically results in long idling periods, followed by smaller periods of high load when the device is being used, which lends itself perfectly for entering low-power states when idle.

There are a few reasons why it might be hard to enter low-power states on select servers during low-load periods. A common practice for large services is to distribute the load evenly across several hundred servers. Not only is computation distributed, but also data. Having several databases on different servers removes a potential bottleneck in having a large, central database. Furthermore, distributing the data increases the pace of recovery in case of an outage. The most efficient low-power states entail turning off parts of the memory, rendering the techniques mentioned above less effective. Transitioning between low-power and high-power states also incur latency penalties, impacting service response time. Thus, there are significant advantages to having all servers available, even at low loads.



Machines are provisioned for peak load

Peak load >> Average load

Request-rate variation for a *Microsoft Bing* sub-cluster over 1 week in 2017

Figure 1.1: Real-world load patterns measured over a period of a week, from Microsoft Bing [7]

One possible way to solve the efficiency problems outlined earlier is *colocating services*. By colocating different services on the same physical node and sharing resources, we might be able to utilize idle time.

As we will expand upon later, user-facing and latency-critical services are common workloads in datacenters. Colocating these types of services is especially challenging, because of *intererence* — performance degradation due to contention of shared resources [8]. Interference is often unpredictable and the exact mechanisms are not trivial to measure.

Software that attempts to manage colocated services are called resource managers, and is what we will showcase in this thesis. By using real-time latency measurements from services, QoS-aware resource managers distribute resources to the different services. Due to abrupt unplanned outages or diurnal patterns, the load can rapidly change, and resource managers should ideally be able to quickly respond to changes in latency.

For this thesis, we wanted to investigate the following research questions:

1. Is the QoS-aware approach more effective than the conventional approach to resource management?

2. What is the cost associated with implementing a QoS-aware resource manager?

In order to do this, we modified the Intel Platform Resource Manager to make it QoS-aware. Hence we present Q-PRM, which builds upon the control mechanisms of the Intel PRM, but uses the QoS (the measured real-time tail latency of the service) instead of platform metrics. We modified an existing load generator for the latency-critical service Memcached in order to create a system capable of monitoring latency in real-time, and fed it to our resource manager. The latency is passed to a *controller* algorithm, which makes latency-guided decisions on where to allocate resources. We implement and compare four such controllers. By modifying the resource manager, we achieved up to 92.4% QoS guarantee, compared to 37.8% for the unmodified Platform Resource Manager. At the same time, we were able to utilize idle CPU time for additional work, such as molecular dynamics simulations.

Initially, we planned to set up a second latency-critical service in addition to Memcached. We wanted to demonstrate that our approach is able to generalize to other services. Because of the challenge of finding a load generator-service combination that would work with our system, we did not have the time in the end.[1] As a compromise, we prioritized setting up a second, more realistic best-effort workload (see Section 5.4.2).

The thesis is structured accordingly: Chapter 2 establishes key concepts within the field, as well as explaining the different control mechanisms used. Chapter 3 gives us an overview over the related work, and Chapter 4 showcases the design of the original resource manager, as well as our modifications and controllers. Chapter 5 outlines our experimental setup, and details our data collection the metrics we used. Chapter 6 presents our results, and Chapter 7 discusses and evaluates those results as well as their implications. Finally, Chapter 8 concludes our thesis.

[1] Our setup required a load generator that could use a dynamic load curve, as well as continuously report the real-time tail latency. Although there are a few general-purpose load generators that are capable of this (such as *wrk2*), additional tailoring to our system was necessary.

Chapter 2

Background

2.1 Fundamental Concepts

2.1.1 Quality of Service

In computer networking and telephony, *quality of service* (QoS) is a term used to describe the overall performance of a service. Depending on the particular context, this could refer to specific measurements such as the latency, throughput or availability of a service. **In this thesis, QoS specifically refers to the 99th percentile tail latency of a service.**

2.1.2 Tail Latency

Tail latency is the small pecentage of requests that take the longest to respond relative to the bulk of requests — that is, the tail end of the response time distribution. Typically, one would look at the 95th or 99th percentile of requests. In order to calculate the 99th percentile latency, one would record the time it takes to answer to a request, sort the requests by latency, discard the worst 1% of response times, and then look at the largest remaining value. In other words, this is the worst response time of the 99% fastest requests. From our own runs we measured that the requests in the 99th percentile were approximately 4 times slower than the average response time, as shown in Figure 2.1.

2.1.3 Latency-Critical Workloads

Latency-critical (LC) workloads (or services) are typically interactive, user-facing services. Their defining feature is that they are *latency-sensitive*, meaning that they are subject to strict QoS requirements, possibly on the micro- or millisecond level. Internet applications such as email, web search or social media are common examples. Many such services make use of other, lower-level services like Memcached [9], creating chains of LC workloads that all need to be served in time. These workloads are also quite bursty in nature, due to the unpredictability of requests arriving [7, 10].

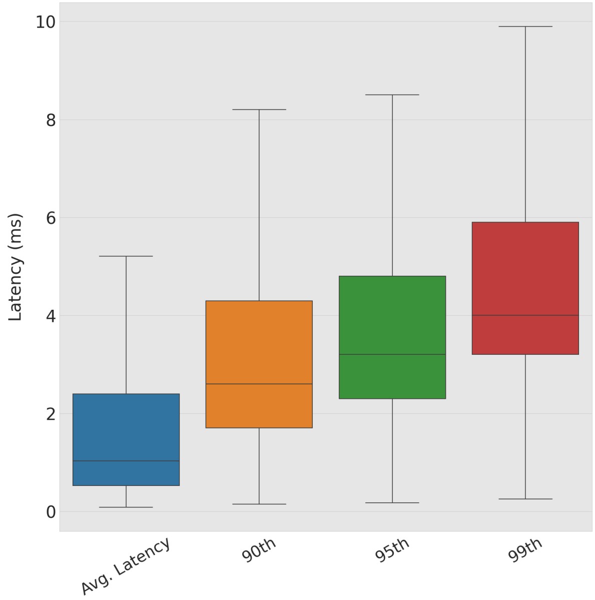


Figure 2.1: The average, 90th, 95th and 99th percentile latencies measured for Memcached

2.1.4 Best-Effort Workloads

Best-effort (BE) workloads (also known as batch workloads) are workloads which, in contrast to latency-critical workloads, are not latency-sensitive. Typical examples could be large-scale number crunching, like generating pay-rolls and reports, maintenance jobs like cleaning up databases and creating backups, or running scientific simulations. In principle, any non-interactive task, such as an operating system background process, could be considered to be best-effort, but the term usually refers to compute-intensive, disk-bound workloads. For the experiments in this work, we use `NAMD`, which simulates molecular dynamics, and `fotonik3d`, which calculates transmission coefficients of photons.

2.1.5 Load Testing

Load The *load* on a latency-critical service is the rate of incoming requests that it receives, measured in *requests per second* (RPS). It is a measure of the amount of work the service needs to do per unit time. Most services are able to process requests with some degree of parallelism, but this depends on the nature of the service. This could be multi-node parallelism, in the case of distributed services, or multi threaded parallelism for single-node services. We only consider single-node services in this thesis.

The higher the load, the more requests the service needs to process in a period of time. This means that the service requires more computing resoures, such as CPU time and memory bandwidth. Also, because it takes more time to process each request on average, the latency will be higher. If the resources available to a service are insufficient to handle the load, the service may start dropping requests. At some point, the load is too high for the service to handle even when all of the system resources are at its disposal. We refer to this as the *critical point* of the service.

Load Testing *Load testing* is the process of putting a service under a load and measuring the latency of its responses. *A load generator* is used to simulate an artificial load, by generating requests and sending them to the service. The requests are generated from a probability distribution so that a desired RPS is achieved, which could be either constant or dynamic (varying over time).

The purpose of load testing is to determine the QoS that a service is able to provide under a given load. Normally, this is done when there is no interference from other tasks on the server, in order to determine the *baseline* performance of the service itself. By gradually increasing the load and measuring the QoS, one can graph the QoS against the load.

According to Delimitrou et al. in their work on PARTIES [11], the QoS of a latency-critical service will start rapidly increasing at some point when the load is too high. This point is called the *max load*, and is found by load testing the service without interference and finding the knee of the QoS-load curve. To be clear, the max load refers to the highest load that the service can handle before the QoS starts deteriorating rapidly, *specifically in the absence of interference*.

Although load testing is normally performed without interference, in this thesis we also benchmark the service while it is under interference from best-effort jobs. In this case, the service has an *effective* max load that is lower than the baseline max load.

2.1.6 QoS Target

The QoS target is the level of QoS that a latency-critical service should be able to provide. Since the QoS is the tail latency of the service, the target is an upper bound on the tail latency. If the tail latency exceeds the target at some point in time, we say that there is a QoS violation. On the other hand, as long as the tail latency is lower than the target, we say that the QoS target is met.

In production systems, such as when deploying a latency-critical service to a cloud hosting provider, the QoS target could be specified by the customer or an administrator. In this case the QoS target is determined by external requirements, such as the expectations of its users. On the other hand, the QoS target can also be defined in terms of the max load of the service on a given platform. This is the convention in the existing academic work on resource managers, such as PARTIES. Hence, we define the QoS target by the average QoS provided by the service at its max load under no interference.

The QoS target is an essential parameter to the Q-PRM, our QoS-aware resource manager. It is the only piece of information that it needs to know that is specific to each latency-critical task. Also, our evaluation metrics are defined in terms of the QoS target. Section 6.1 describes how we find the QoS target experimentally.

2.2 Resource Managers

A resource manager (RM) is a program that runs on a server and decides how to divide some set of hardware resources between a set of tasks. In this work, we refer to the set of tasks as the *workload configuration*.[1] The workload configuration designates each task as either a latency-critical (LC) or best-effort (BE) task. There may be other processes running on the operating system, but these are not managed. A task typically refers to a container instance, and therefore includes all of the processes within that container.

The particular set of managed resources varies, depending on the resource manager design and the capabilites of the server. These could be, for example:

- CPU cores
- CPU frequency
- Cache ways
- Memory bandwidth
- Disk bandwidth
- Network bandwidth

Usually, the CPU resource is managed either as the number of cores, or the fraction of CPU time allocated to tasks.

The resource manager has to find a partitioning of the resources that satisfies the QoS target of each latency-critical task, and maximizes the total throughput of the best-effort tasks. This decision must be based on some collection of metrics that are measured continuously and characterize the current behavior of the system. For example, performance statistics such as the CPU usage or cache miss ratio could be used. The problem is to determine whether the latency-critical tasks have the resources they need to meet their QoS targets. This is complicated by the fact that different LC tasks require different amounts and types of resources. Additionally, the resource requirements depend on the intensity of the load experienced by the LC task.

2.2.1 QoS-Awareness

While conventional resource managers use performance metrics in order to gauge resource contention between services, this is a proxy for what we are actually interested in — the QoS of the latency-critical service. Coupled with the growth of cloud services, there has been an increasing amount of research done on *QoS-aware Resource Managers*, the specifics of which we will outline in Chapter 3. QoS-aware resource managers, generally, are aware of the QoS Target of the latency-

[1] This is the terminology used by the Intel Platform Resource Manager.

critical applications, as well as their real-time latency. Some resource managers use other metrics in addition to the latency in order to allocate resources.

2.3 Memcached

For our experiments, we use *Memcached* [12] to function as a latency-critical service. Memcached is a widely used, open-source memory object caching system. It is used by large web platforms such as Facebook [9] and Twitter [13] to reduce key-value access times. By caching database objects in memory as a hash table, the average time to access them is reduced. Not only does one avoid slower disk accesses, but also the overhead of database processing. Memcached is not an especially compute-intensive application . To service a request, it has to do a hash table lookup and either store or retrieve an object from memory. Hence, most of its execution time is spent stalling for memory stores or loads. However, Memcached is very sensitive to memory bandwidth interference in spite of this, because it has to retrieve objects and respond to requests very quickly [11]. In other words, while Memcached may not use the total memory bandwidth available, it is still very sensitive to memory access delays, which could be caused by interference from other running tasks.

Chapter 3

Related Work

There has been substantial work on developing QoS-aware resource managers in recent years. Much of the older literature focuses on colocating a single LC workload with multiple BE workloads, while newer methods also open up the possibility for colocating several LC workloads. We will describe some of the approaches and their respective papers next.

3.1 Heracles

Heracles [14] is a "dynamic controller" that combines a single latency-critical workload with several best-effort workloads, giving and taking resources from BE jobs by managing four hardware and software mechanisms. The authors pose that interference only occurs when a resource is saturated. Thus, Heracles attempts to decompose the resource optimization problem into smaller subproblems.

When the measured QoS is sufficiently under the QoS target, Heracles uses gradient descent to find the minimum configuration of LLC occupancy and CPU cores that satisfies the QoS target. Furthermore, it also incorporates CPU frequency and network controllers.

3.2 Octopus-Man

Octopus-Man [15] is a QoS-aware resource manager for heterogenous multicore server systems. It runs on a single server and monitors both hardware and OS performance metrics, as well as the the QoS of its services. Based on this profile, it then decides on a mapping of latency-critical and best-effort tasks to CPU cores (so-called brawny and wimpy cores). When the load on LC services is low, the QoS target can be met by using the more energy efficient wimpy cores for the latency-critical services, and the more powerful brawny cores for the best-effort jobs, which results in better energy efficiency. When the load is low, the wimpy cores are sufficient to meet the QoS target. When the load is high, the Octopus-Man instead uses brawny cores to meet the LC QoS target, while wimpy cores are used for best-effort jobs.

Octopus-Man has two different algorithms for its mapper. The PID version samples QoS at a fixed interval and stores it in a buffer, then determines a core allocation by computing a function of the current, net total and derivative error (difference between the latency and the QoS target). The output of the function is linearly scaled and mapped to the set of core configurations. The function has 3 parameters that are tuned per application using the root locus method. Also, they measure average latency vs. RPS per application, and use linear regression to predict QoS as a function of a core allocation. The PID mapper is not service-agnostic, since it needs to profile each service to tune the PID controller.

The second algorithm utilizes a state machine in order to model Octupus-Man's task mapper. Each state represents a configuration of brawny and wimpy cores. The transition to another state is predicated on QoS measurements triggering certain conditions. If the QoS measurements are below a certain threshold, which is derived from the QoS target, the state machine either increases the amount of wimpy cores or migrates the tasks to brawny cores. The inverse operation takes place if the QoS measurements are above, or is likely to violate QoS targets in the immediate future.

3.3 PARTIES

PARTIES [11] is a QoS-aware resource manager capable of managing multiple colocated latency-critical services. PARTIES introduces the concept of *fungibility*, i.e. that resources can be traded between services. By trading resources, as well as adjusting resources that affect the whole system, the PARTIES-approach attempts to find the minimal resource usage that satifies all QoS targets. Those resources can then be reclaimed for best-effort jobs. PARTIES finds the optimal configuration *experimentally*, by measuring the QoS before and after any resource changes.

Like the Q-PRM, it has to be provided with the QoS target of each latency-critical service. Additionally, it needs to know the network bandwidth requirements of each service. Otherwise, it is service-agnostic and does not use any empirically tuned parameters.

Chapter 4

Design and Implementation

The Intel Platform Resource Manager (PRM) is an open source, non-QoS-aware resource manager. It uses hardware performance counters and OS statistics to collect information about the resource usage of the latency-critical workload, monitoring CPU utilization, memory bandwidth, and a large number of cache metrics. The Intel PRM is capable of managing a colocation consisting of multiple LC and BE workloads. However, it only regulates the BE workloads directly. These are regulated together as a group, so that each individual workload receives an equal share of the total best-effort resource allocation.

4.1 Intel Platform Resource Manager

4.1.1 Overview

The Intel PRM consists of two executable parts. In addition to the main runtime, which is responsible for resource monitoring and regulation, there is the offline *Analyzer*, which is used to build a set of resource *thresholds* from previously collected metrics.

The PRM is deployed in a three-step process. The first step is a profiling run, during which regulation is disabled and the PRM collects a set of metrics as the latency-critical workload runs without best-effort workloads. In the second step, the Analyzer generates the thresholds based on the recorded *max usage* of every resource. During normal operation, the best-effort workloads are colocated along with the latency-critical workload, regulation is enabled, and the PRM uses the threshold profile to detect resource contention.

The PRM uses the following main concepts:

- Each workload runs in a separate Docker container.
- Only the BE workloads are regulated. The LC workloads are unrestricted.
- There are two managed resources: the CPU quota, and the last-level cache (LLC) occupancy. Each of them has a global *resource level*, which is shared between the BE workloads.

- The *Monitor* continuously collects the global cache metrics, as well as each workload container's CPU utilization. It is responsible for detecting resource contention based on the generated thresholds.
- The *Controller* implements the logic that regulates the resources. There is only one implementation, the `NaiveController`:
  - If CPU cycle contention is detected, the CPU quota level is set to zero. If contention has not been detected for 14 seconds, the level is incremented by one.
  - If LLC contention is detected, the LLC level is set to zero. If contention has not been detected for 2 minutes, the level is incremented by one.

4.1.2 Control Mechanisms

The Intel PRM controls two resources — the CPU quota and the last-level cache (LLC) occupancy. Every job, whether LC or BE, will be isolated into Docker containers, which allow these jobs to be resource throttled and monitored. We will go into the details of the control mechanisms next.

Docker

Docker is a type of virtualization software that wraps an application in a container — where one isolates an application's dependencies, storage, memory, I/O and much more. Docker containers are used throughout the PRM in order to isolate the indiviudal latency-critical and best-effort tasks. By interfacing with Linux *cgroups*, containers can be continuously monitored for performance metrics.

CPU Quota

Using Linux *cgroups*, the Intel PRM controls the CPU quota, which is an upper bound on how much CPU time a given process gets. Since the PRM only regulates best-effort jobs, the CPU quota is only ever set for these jobs.

Intel CAT

On the LLC-side, the Platform Resource Manager uses Intel Cache Monitoring Technology (CMT) to monitor cache metrics, and Intel Cache Allocation Technology (CAT) to control cache occupancy [16]. In PARTIES, the authors state that LLC-occupancy indirectly controls the memory bandwidth, because cache hit rate is highly correlated with memory bandwidth [11].

Resource Levels

The magnitude of the CPU quota and LLC adjustments are fixed, meaning that every adjustment increase and decrease the respective resources by the same value. The Intel PRM also holds internal states of the current resource level for both the CPU quota and the cache. These resource levels are represented as integers from 0 to 20, and the PRM then maps that level to a concrete CPU quota value and LLC cache ways.

The CPU resource level determines the discrete CPU quota value by using Linux' *cgroups*. The maximum allowable discrete CPU quota value is either discovered during the preliminary profiling or recorded during run-time of the PRM. If no contention is discovered at this threshold, the best-effort job is allowed as much CPU time as it needs.

The LLC resource level determines the number of cache ways that are allocated to the best-effort task, while the latency-critical tasks are always allowed to use the entire cache. Level 0 corresponds to 2 cache ways, while the max level corresponds to the entire cache and depends on the associativity of the server's last-level cache. Because our server is equipped with a 20-way L3 cache (see Section 5.1), the max level corresponds to 18 cache ways on our system.[1]

4.1.3 Runtime Explanation

In its initial state, the Intel PRM strips best-effort jobs of all resources, letting the latency-critical job work unrestricted. The LLC- and CPU-regulation run on different threads. The CPU quota regulation is very simple - it checks if the current CPU utilization, latency-critical and best-effort, is higher than the recorded CPU utilization for the LC job. On the LLC-side, the controlling mechanisms are more complicated. During run-time, the PRM builds up a history of cache-related metrics, recording the metrics of the LC-application as a function of CPU utilization. The goal is to detect potential *resource contention* of the LLC, recording cache metrics at different CPU utilization levels.

If the utilization threshold is not breached for 3 consecutive measurements, the CPU quota is increased. Consequently, if no LLC-contention is detected for 3 consecutive measurements, the BE-job gets an additional cache way. If the PRM detects that an increase in the resource level will cause potential contention, it holds the current resource level. If the utilization threshold is breached, or LLC-contention is uncovered, the PRM strips the BE-job of access to that resource. It does this by setting the CPU quota to a very small number, or setting the number of available cache ways to 0. All best-effort jobs are treated as one resource entity - meaning that if there are several best-effort jobs registered by the PRM, they are all subject to the same resource level value, and get an equal share of allocated resources.

4.2 QoS-Aware Platform Resource Manager

One of our main tasks for this project was to make the Intel Platform Resource Manager QoS-aware. Although we made some rather fundamental changes to the

[1] If the `--exclusive-cat` option is enabled, each cache way is exclusively allocated to either the LC or BE tasks.

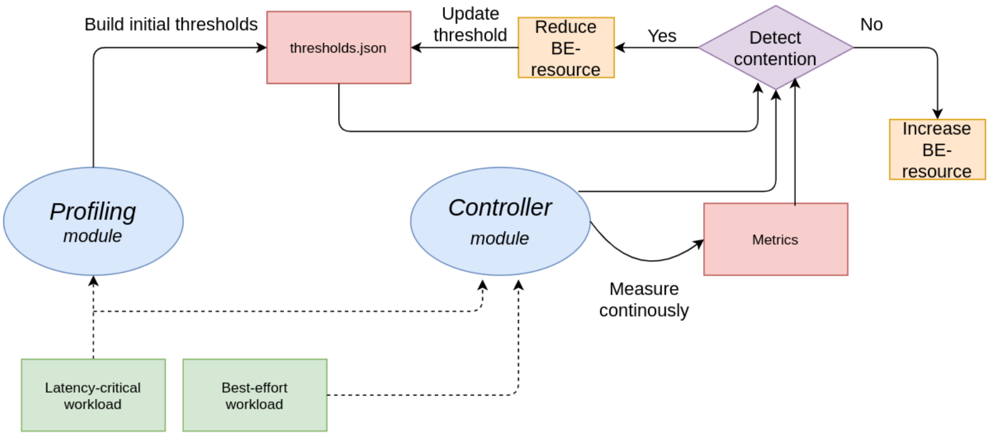


Figure 4.1: Overview of the unmodified Intel PRM
The "Profiling" module is ran first, generating the initial thresholds. The "Controller" module is running continously during the lifetime of the workloads, detecting contention, controlling resources and updating the thresholds.

source code of the PRM, most of the original structure is preserved. We refer to our modified version of the PRM simply as the Q-PRM (QoS-aware Platform Resource Manager).

The most important change introduced in the Q-PRM is that the QoS alone is used to make allocation decisions, rather than the CPU utilization and other metrics that are considered by the PRM. Consequently, there is no preliminary profiling and no concept of thresholds in the Q-PRM. The profiling component of the PRM is not used. Instead, the QoS target serves as a threshold for the QoS, and this must be provided to the Q-PRM as a parameter.

In the original PRM, the controller logic was distributed between the main threads, the CPU quota and LLC resource classes, and the NaiveController controller. These components were connected by the concept of *contention*. We decided to simplify this structure by gathering the functionality in the Controller class, partly because our needs were simpler, and partly because the original structure was needlessly complicated. For the Q-PRM, a controller algorithm (corresponding to a Controller subclass) is specified from the command line. The controller owns both of the CPU quota and LLC resources, and is provided with the QoS measurement on every update. The QoS is read once per second from a file which is also updated every second by the client of the latency-critical service (the load generator, see Section 5.3). Each controller implementation then has to decide how to set the best-effort resource levels, based on the QoS. Because we wanted to keep the optimization problem simple, the controllers we implemented for this thesis only regulate the CPU quota.

4.2.1 The Optimization Problem

A simplified model of the optimization problem that we need to solve is that, at a given point in time, for a given colocation of one latency-critical and one latency-critical workload, the QoS is determined by the intensity of the load on the LC service (the RPS), and the amount of resources that is available on the system for the LC service to handle that load. Assuming there are no background tasks running on the system other than the BE job, the amount of resources available is determined by the interference from the BE job, which is a function of the CPU quota it is allocated by the RM. Hence the QoS is a function of the current LC load, which the RM has no control over, and the BE CPU quota. Our task is to find the optimal level Q of CPU quota, meaning the highest level that we can give the BE job without violating the QoS target, based on the measured QoS. Q is then only a function of the current load on the LC service.

4.2.2 QoS-Aware Controllers

The following controllers base their regulation decisions on the QoS *slack*. We define the slack as the relative error of the currently observed QoS with respect to the QoS target:

$$\text{Slack} = \frac{\text{Target} - \text{QoS}}{\text{Target}}$$

From this definition, it follows that when the slack is nonnegative, the QoS target is met, meaning that the controller should consider allocating more CPU time to the best-effort workload, depending on the amount of slack. Conversely, if the slack is negative there is currently a QoS violation, and the controller should attempt to recover by throttling the best-effort workload.

Step

The first controller we implemented was the Step controller. It was intended to be similar to the PRM's NaiveController, except that it uses the QoS and QoS target instead of the CPU utilization and the CPU threshold. The controller only increases or decreases the CPU quota by a single resource level at a time. Contrary to the NaiveController, it does not throttle the quota to level 0 whenever CPU contention is detected, but merely reduces the level by one step. For every 3 seconds that no contention has been detected, the level is increased by one.

Proportional

This algorithm uses a linear function to determine the CPU quota given to best-effort jobs. The change in CPU quota is proportional to the slack. In other words, when the latency is either much higher or much lower than the threshold, this controller increases and decreases the CPU quota more aggressively. The intuition behind this is rooted in the fact that we want to quickly recover from sudden load increases or decreases.

Proportional v2

This controller is mostly identical to the one above, except for one key difference: exponential mapping from resource level to CPU quota when the resource level is low. Monitoring how the proportional controller reacts to changing load, we noticed a behavior where at *high loads* we did not have enough resolution in our CPU quota increase/decreases. Every discrete increase and decrease in resource level is mapped to a constant change in the CPU quota, and this is not ideal at high loads. The behavior we observed was highly oscillatory. Our solution was to use an exponential mapping from resource level to CPU quota when the resource level is low, in order to increase the resolution when the load is high. As we will see in the results, this approach yielded us good results.

Basic

The Basic controller uses a ring buffer to store the 10 most recent QoS slack measurements. From these measurements one of two running averages is computed, using either the entire buffer or just the 3 most recent values. As long as no QoS violation is detected, the controller waits 10 seconds before computing the average slack for this period. This average is then used to compute a loss function, which is simply the absolute difference between the average measured slack and 0.2. The reasoning for this is that the controller should attempt to maintain a slack of approximately 20% (a tardiness of 0.8), as a compromise to ensure that there is some margin to prevent a sudden QoS violation. At each decision interval of 10 seconds, the controller checks if the loss has improved or worsened since the last time it was computed. If the loss has increased (the slack has moved away from 20% in either direction), the *direction* of CPU quota regulation is reversed. If not, the controller continues to increase or decrease the CPU quota resource level in the same direction as before. In other words, whether the Basic controller continues to scale up or down the resource level of the best-effort workload depends on whether or not there has been an improvement to the slack in the last 10 seconds. On the other hand, the magnitude of the changes made to the quota level is determined by the magnitude of the slack, similar to the Proportional controllers. If there is a large amount of slack, greater steps are taken; if there is less than 25% slack, only single steps are used, etc.

A violation is considered to occur whenever the 3-second average slack is less than 5%. This average is computed once per second, as opposed to the 10-second average. When a violation is detected, the quota is immediately reduced to level 0 and the controller goes into recovery mode. In recovery mode, the controller waits until the 10-second average slack is at least 30% before making any changes to the best-effort quota level. When the slack has recovered, the quota level is restored to half of what is was when the violation happened. This mechanism was an attempt to recover more quickly from a QoS violation, to perform more best-effort work, compared to the NaiveController controller of the PRM, which slowly recovers from quota level 0 even when there may be sufficient slack to be less cautious. As we will see in Chapter 7, this strategy works well when the violation is mainly caused by best-effort interference. On the other hand, when the violation is mainly caused by a high load on the latency-critical service, it often leads to repeated QoS violations instead.

Chapter 5

Experimental Setup and Methodology

5.1 Server Platform

We performed the evaluation of our resource manager on the NTNU IDUN computing cluster [17], on an Intel Xeon E5-2695 server running CentOS 8. The server has two CPU sockets, which correspond to two NUMA (non-uniform memory access) nodes. In a NUMA system, the memory is shared between the nodes such that is appears as a larger shared memory to the operating system. Each node has its own memory controller and memory, but can acess the other node's memory through an interconnect incurring higher latencies. For our purposes, the advantage of this architecture is that we can achieve some degree of isolation between the processes running on each node by constraining them to run on one node, and to only use the local memory of that node. This allows us to simulate a client-server setup where each node acts as a host machine with no need for two physical hosts, which simplifies setting up and performing experiments. Table 5.1 lists the specs of our server platform.

5.2 Overview

The aim of our experiments is to compare different resource management schemes for a colocation consisting of a single latency-critical task and a single best-effort task. That is, we run the Intel PRM resource manager along with one LC task and one BE task, and evaluate its performance. Because the Intel PRM is designed to manage its tasks as Docker containers, the LC and BE workloads each run in their own containers. This also makes it easier to clean up after a benchmark, by destroying the containers and then recreating them for the next run.

To characterize the baseline performance of the workload colocation, and to evaluate the perfomance of our resource manager, we perform load testing of the latency-critical service. Essentially, a load generator is ran on one of the CPU

Table 5.1: Platform specification

Model	Intel Xeon E5-2695 v4
OS	CentOS 8.2.2004
Kernel	Linux 4.18
Virtualization technology	Docker Engine 20.10.2
Sockets	2
Cores/socket	18
Threads/core	1
Base/max turbo frequency	2.10 GHz / 3.30 GHz
Frequency driver	ACPI (performance governor)
L1 inst/data cache	576 KiB / 576 KiB
L2 cache	4.5 MiB
L3 cache	45 MiB
Memory	128 GiB

nodes, which generates requests for the service running on the other node. Since the load generator and service act as the client and the server, we refer to the nodes on which they run as the client and server nodes respectively. We refer to the lifetime of a single instance of the LC application and load generator as a benchmarking run. When a run is complete, we stop all the components of the system so that the next benchmark starts with a clean slate. The reasoning behind this is that every new benchmark (most importantly the Memcached instances) should be independent from the previous ones. In practice it is not possible to make these experiments completely deterministic, because for every run there are side-effects on the state of the operating system which are outside of our control. Still, we make an effort to reduce the variability of our results.

A high-level overview of our setup is shown in Figure **??**. The server node consists of the LC task, the BE task, and the Intel Platform Resource Manager. The client node consists of the load generator and data collection scripts. The server node represents the system under test. The reason we run our data collection scripts and the load generator on the client node is so that it does not affect this system.

One deviation from the above description is that we limit the server node to 9 CPU cores. Initially, we used all of the 18 cores on the node, as we do with the client node. However, we discovered that one of the two main components of Intel PRM requires a minimum level of CPU utilization from the LC application to function at all, and on our system Memcached did not reach this level even at the max load.[1] As a result, we decided to downsize the server node, so that the relative CPU utilization for Memcached would be higher. This was done by allocating only 9 CPU cores to the LC, BE and RM programs running on the server node, as well as by configuring the maximum CPU count to 9 in the PRM.

[1] Specifically the metrics subsystem of the PRM. We explain how this works in Chapter 4.

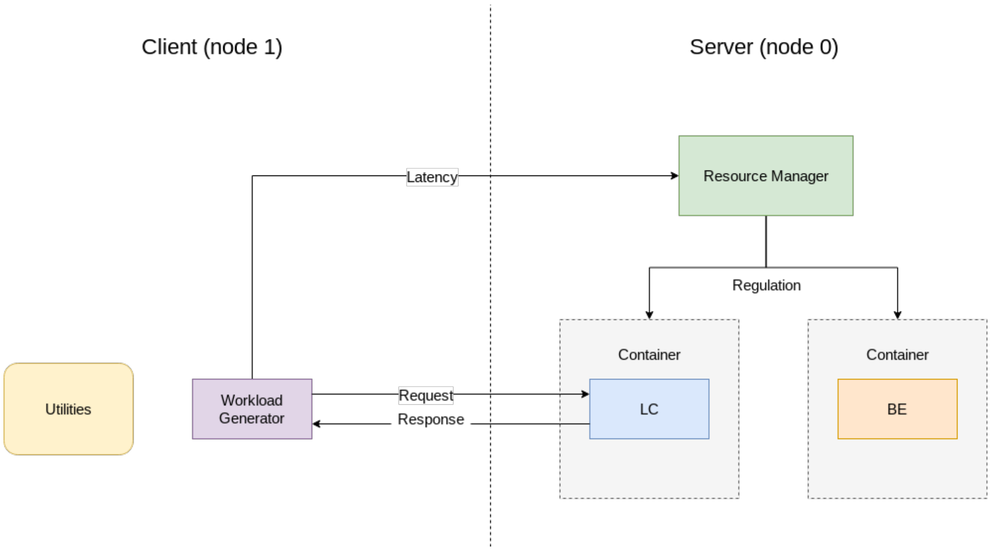


Figure 5.1: Overview of the Experimental Setup

5.3 Data Collection

In addition to the tail latency, our benchmarking script collects the CPU and memory usage of the LC and BE tasks, the number of instructions retired by the BE task, and the logging output from Intel PRM. This data is timestamped, which allows us to analyze the behavior of the resource manager and its workload as it evolves over time.

Tail Latency The load generator records the latency of each request and reports the tail latency at a specified interval. For constant throughput runs, the reporting interval is equal to the duration of the run. For dynamic throughput runs, the interval is one second, and the tail latency is continuously written to a file. The QoS-aware version of Intel PRM reads this file to get the tail latency.

CPU and Memory Stats We collect the CPU utilization and memory usage of the LC and BE tasks by querying the Docker Engine API (v1.26). The API responds once per second with the total resource usage of each container. Alternatively, the same statistics could be collected by reading the sysfs files at the cgroup level of each container.

Instructions Retired We use `perf`, a Linux performance analysis tool, to record the number of instructions retired by the processes in the best-effort task's cgroup. `perf` uses the hardware performance counters of the system to count certain events, in this case the instructions retired.

Intel PRM Actions While running, the Intel PRM logs every resource allocation action it takes. Specifically, it logs the CPU quota and LLC resource level changes for every BE task along with a timestamp. We record the standard output of the PRM to a file, which is then parsed for the relevant output into a CSV file.

5.4 Workloads

5.4.1 Memcached

We use Memcached as our latency-critical workload. Our setup for Memcached is based on the CloudSuite benchmark suite [18], specifically the *Data-Caching* benchmark, which consists of a server Docker image containing Memcached itself, and a client image containing CloudSuite's custom Memcached load generator. The load generator image comes with a 300 MB Twitter dataset that needs to be scaled up prior to running the benchmark. To make it possible to use a new container for every benchmarking run, we pre-scaled the dataset by a factor of 30 and included it in the image. We also modified the load generator itself in order to obtain precise timestamped output, and so that it would continuously write the current 99$^{\text{th}}$ percentile latency to a file once per second.

Table 5.2 lists the configuration we used for Memcached and the load generator. These are mostly the default settings recommended by CloudSuite. The client and the server are configured to use as many threads as there are CPU cores. The server has a memory limit equal to the size of the local memory on the server node.

Table 5.2: Memcached benchmark configuration

(a) Memcached

Version	1.4.24
Worker threads	9
Memory limit	64 GiB
Minimum object size	550 MiB

(b) Load generator

Worker threads	18
Connections	450
Arrival distribution	Exponential
Dataset scaling factor	30

In our benchmarking setup, we distinguish between constant and dynamic throughput runs. For our profiling runs, the load generator generates requests at a constant RPS for duration of the run, and the latency distribution is only reported as a total statistic at the end of the run. For the dynamic RPS runs, the load generator uses a predefined sinusoidal load curve, and we collect the tail latency once per second. We use the data collected from these runs to analyze the behavior of the system over time.

A single run consists of a warmup stage followed by a main stage. For the warmup stage, a Memcached container is run along with a load generator container. The load generator then pre-loads the Memcached server with the dataset. When this process is complete, the load generator is killed and the script immediately proceeds to the main benchmarking stage. At this point the Intel PRM, BE workload container and data collection scripts are started. Then the main load generator is run in a new container, which performs the actual load testing benchmark using the specified load curve. By dividing the run into two stages, the idea is that we only start collecting data after the warmup has completed.

5.4.2 SPEC CPU®2017

We initially used the stress testing tool *stress-ng* to simulate a best-effort workload, but eventually decided to use the SPEC CPU®2017 benchmark suite instead. Specifically, we use the `549.fotonik3d_r` and `508.namd_r` benchmarks from the SPECrate®2017 Floating Point suite. We refer to these simply as `fotonik3d` and `NAMD` from here on. These two in particular were picked because we wanted to compare two significantly different best-effort workloads. We expected `fotonik3d` to be more memory intensive, and `NAMD` to be more compute intensive. Considering that Intel PRM regulates CPU and cache usage separately, we wanted to see if, for example, it would be less effective at regulating `fotonik3d` when cache allocation was disabled.

The SPECrate®2017 Floating Point suite is normally used to score the performance of a computer in terms of throughput. Typically one would run one or more copies of the same benchmark in parallel and compare the time used on the system under test to the time used on a reference computer. The way we use the suite here is non-standard, as our intention is not to benchmark the server itself, but rather to simulate a somewhat realistic best-effort workload. Using SPEC®CPU 2017 is preferable to load testers like `stress-ng` because the workload is more realistic. Furthermore, since these benchmarks are deterministic in the sense that the same instructions are executed each time they are run, there should be less variation between repeated identical runs.

Our setup involves running one of the above benchmarks in a container. We only run one instance of one of these benchmarks at a time. We use a very simple wrapper script as the entrypoint to the container. This script is used to restart the given SPEC®CPU benchmark whenever it has finished executing, so that there is always some best-effort work available for the duration of a benchmarking run.

5.5 Intel PRM Profiling

We run the Intel PRM with a set of *thresholds* that are generated following the profiling procedure described in the README. Memcached is put under the same dynamic load that is used in the main experiment (see Section 6.2.1), but extended to 30 minutes. The PRM runs in the background with the `--collect-metrics` and `--record` flags, collecting the platform metrics and CPU utilization statistics. The thresholds are then generated from the collected data by running the *Analyzer*.

For the actual benchmarks, we run the PRM with the `--collect-metrics`,

`--detect`, `--control` and `--enable-hold` flags. We use the default sampling/decision intervals for CPU utilization (2 seconds) and cache metrics (20 seconds).

It should be noted that the README is ambiguous about how the PRM should actually be profiled and run, as well as about the meaning of the different command line options. Based on our understanding of the code, the preliminary profiling run should only include the latency-critical workload, and no best-effort workload. The threshold profile can then be used to detect resource contention, presumably because the thresholds will be exceeded when the best-effort workload is added.

5.6 Evaluation Metrics

To evaluate the performance of each resource manager implementation, we consider the QoS of the latency-critical task and the throughput of the best-effort task. Because there is a tradeoff between the QoS and throughput, both aspects must be taken into account in order to evaluate the resource manager.

Quality of Service We follow a convention established by the work on resource managers such as Octopus-Man [15] and Twig [19], where the QoS is reported using two distinct metrics. The *QoS guarantee* is the fraction of samples for which the measured QoS is below the QoS target, or in other words the fraction of time for which the QoS target was met. This is reported as a total statistic for each benchmarking run. The *QoS tardiness* is the ratio of the measured QoS to the QoS target, for each sample. A sample with a QoS tardiness above 1 corresponds to a QoS violation. Whereas the guarantee only reflects whether or not there is a violaton, the tardiness reflects the magnitude of the violation — a higher tardiness means a more serious violation. We report the average QoS tardiness for a benchmarking run.

Instructions Retired For the best-effort workload, we use the number of instructions retired as a measure of the amount of work done during the course of a run. We normalize this to the instructions retired by the best-effort workload when it runs alongside the latency-critical task, with no limit on its resource usage imposed by the resource manager.

Chapter 6

Evaluation

6.1 QoS Target and Max Load

In order to determine the QoS target for Memcached, we have to determine its max load under our setup. To find the max load, we follow the load testing procedure explained in Section 2.1.5. We run a series of benchmarks across a range of RPS. Based on the definition of QoS target, the idea is to measure the baseline performance of Memcached in the absence of resource contention. In other words, when it has all of the resources of the server node at its disposal. Hence, we use no best-effort task, and no resource manager, for these runs.

A single run is a 5 minute benchmark at a constant RPS. The RPS ranges from from 450,000 to 530,000, incremented in steps of 2,500. The range of interest was determined by trial and error in earlier, less extensive experiments. Because the sampled QoS has a high amount of dispersion, it was sampled 25 times for each RPS value. The resulting QoS vs. RPS graph is presented in Figure 6.1.

The mean QoS jumps from 4.76 ms at 500,000 RPS, to 20.4 ms at 502,500 RPS. We find that the max load is 500,000 RPS, which corresponds to a mean QoS of 4.76 ms. Hence we set the QoS target to 4.76 ms.

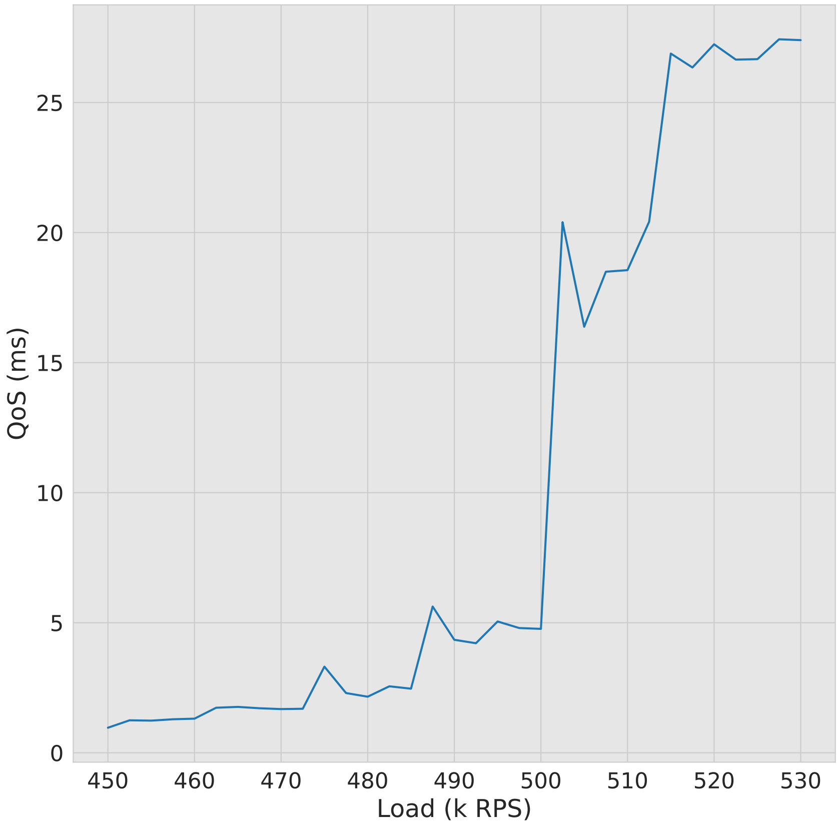


Figure 6.1: Measured QoS vs. constant throughput load (k RPS) for Memcached

6.2 Resource Manager Evaluation

6.2.1 Benchmark Configurations

This section describes the main part of our experiment, in which we compare different implementations of the Intel PRM. The benchmark configurations we use are explained in Table 6.1. Each benchmark configuration is repeated for 20 runs, from which we compute the average metrics. For every benchmark except `No RM`, there are 20 runs for each of the best-effort tasks (`fotonik3d` and `NAMD`). Every run uses the same 10 minute dynamic load curve, which is shown in in Figure 6.2. The load is sinusoidal and ranges from 380,000 to 500,000 RPS.

Table 6.1: Benchmark configurations

Resource Manager	Benchmark	Controller	BE task	Resources
–	No RM	–	–	–
	No RM w/ BE	–	NAMD/fotonik3d	–
PRM	PRM	Naive	NAMD/fotonik3d	CPU
	PRM w/ CAT	Naive	NAMD/fotonik3d	CPU, LLC
Q-PRM	Step	Step	NAMD/fotonik3d	CPU
	Proportional	Proportional	NAMD/fotonik3d	CPU
	Proportional v2	Proportional v2	NAMD/fotonik3d	CPU
	Basic	Basic	NAMD/fotonik3d	CPU

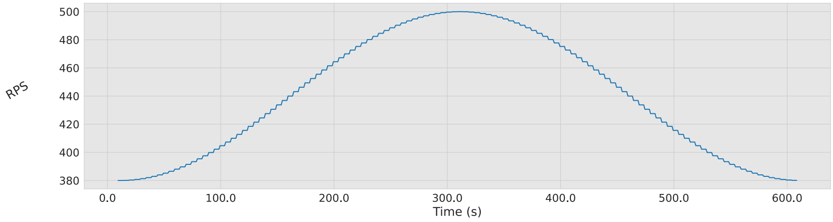


Figure 6.2: Dynamic load curve (RPS in thousands)

Establishing a Baseline

As a starting point, we benchmark Memcached using no resource manager. This establishes a baseline for what we can expect in terms of QoS and best-effort throughput.

No RM This benchmark simply involves running and load testing Memcached by itself, using no best-effort task. This results in a mean QoS guarantee of 99.2±1%. Seeing as the following benchmarks introduce resource contention in the form of a best-effort task, this is the highest QoS guarantee that could be achieved in the following experiment by an ideal resource manager. The fact that the guarantee is not 100% is explained by the fact that the load curve reaches the max load when it peaks at 500,000 RPS. Although a 100 % guarantee is achieved for some runs, there is a degree of variability between them.

No RM w/ BE We introduce a best-effort task to infere with Memcached. Because there is no resource manager, no limit is imposed on the resource usage of the best-effort task. In other words, the LC and BE tasks contend for resources as equals. Seeing as Memcached is under the same high load as before, the result is unsurprising — the QoS guarantee is very low (0.08% for NAMD and 0.03% for fotonik3d), and the number of instructions retired is high. The amount of instructions retired that we measure under these conditions represents the highest possible best-effort throughput in this experiment. This is because when a resource manager is introduced, it can only restrict the resource usage of the best-effort task compared to this baseline. Hence, for the following benchmarks, we normalize the instructions retired as fraction of these results, for NAMD and fotonik3d respectively.

Comparing Resource Managers

This is the main part of our experiment, in which we compare the different controllers of the Intel PRM and Q-PRM. We compare the unmodified PRM, which uses the NaiveController controller, to the Step, Proportional, Proportional v2 and Basic controllers of the Q-PRM. For the PRM, we use two different configurations: the PRM benchmark where the PRM is run with the `-disable-cat` option, and the PRM w/ CAT benchmark, which uses last-level cache allocation (CAT). This comparison is interesting because our Q-PRM does not use cache allocation. We wanted to see how important this mechanism is for the effectiveness of the PRM. It also allows us to compare the PRM and Q-PRM in the case where they both only regulate the CPU quota.

6.2.2 Results

Our results for each of the benchmark configurations are presented below. The QoS guarantee and tardiness metrics are presented as box plots in Figure 6.3 and Figure 6.4 respectively. Here, the median value is indicated by the center line of the box, the box itself indicates the interquartile range, and the whiskers show the minimum and maximum values. The metrics for NAMD and fotonik3d are shown on the left, and their average is shown on the right. Figure 6.5 presents the normalized number of instructions retired.

Overall, the Q-PRM performs much better in terms of QoS, while the PRM achieves a higher best-effort throughput. There are only small differences between the individual Q-PRM controllers, and between the PRM and PRM w/ CAT benchmarks. On average, the Q-PRM controllers achieve a 90.0% guarantee, compared a 37.3% guarantee for the PRM. On the other hand, the PRM manages to retire 4.37 times as many instructions as the Q-PRM.

The average tardiness is 0.84 for the Q-PRM and 1.29 for the PRM. In other words, the average tail latency sample for the Q-PRM has a 16% slack with respect to the QoS target, while the average sample for the PRM violates the QoS target by 29 %. Figure 6.4 illustrates that, for the PRM, the tardiness is much more dispersed around the median, compared to the Q-PRM.

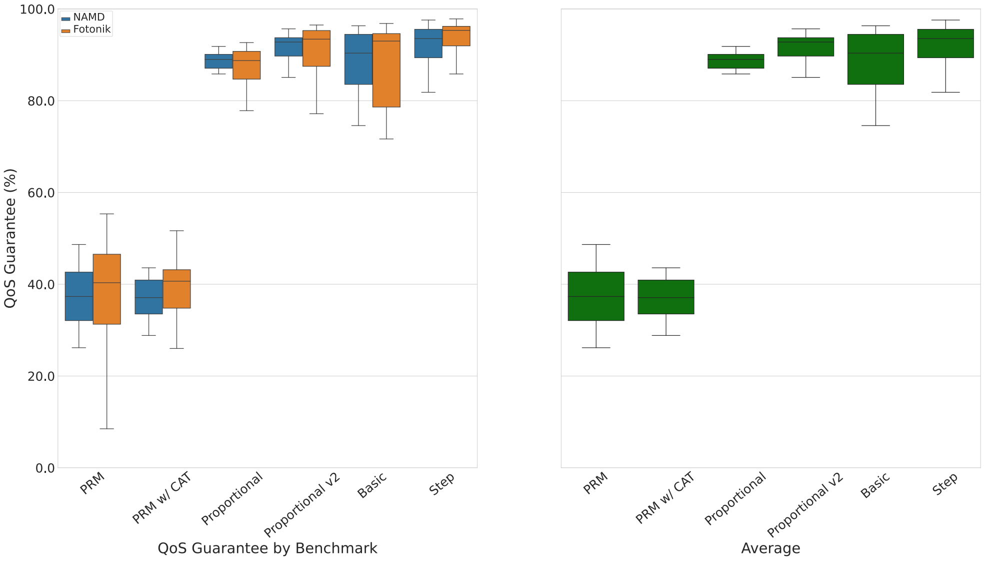


Figure 6.3: QoS guarantee for each benchmark. Values for NAMD and fotonik3d respectively are shown on the left, while their average is shown on the right. The boxes indicate the interquartile range, the whiskers indicate min and max values, and the center line indicates the median.

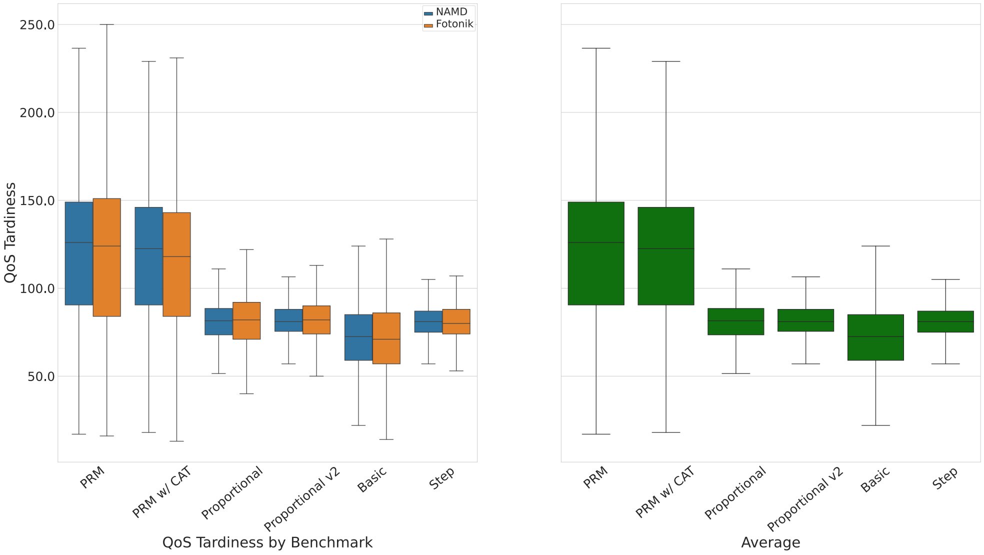


Figure 6.4: QoS tardiness for each benchmark. Values for NAMD and fotonik3d respectively are shown on the left, while their average is shown on the right. The boxes indicate the interquartile range, the whiskers indicate min and max values and the center line indicates the median.

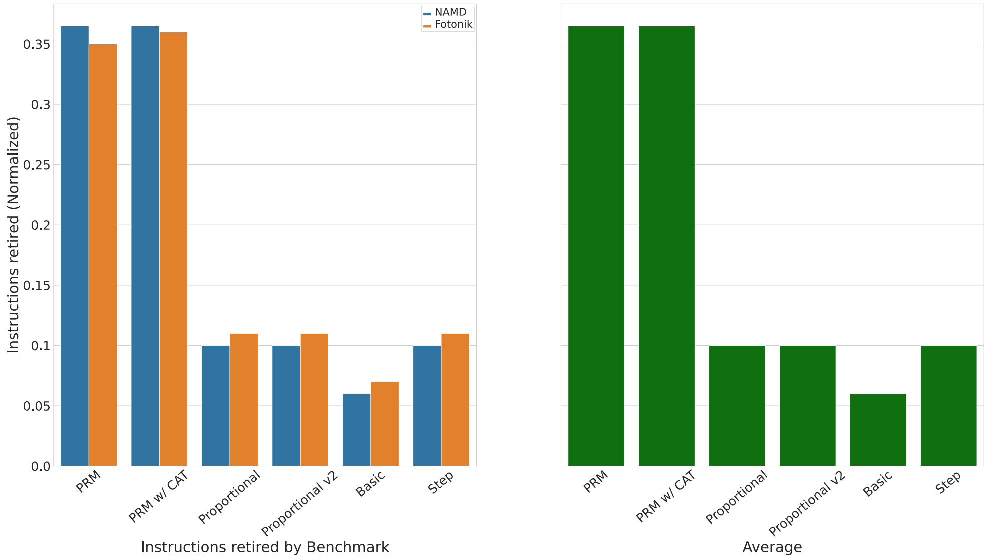
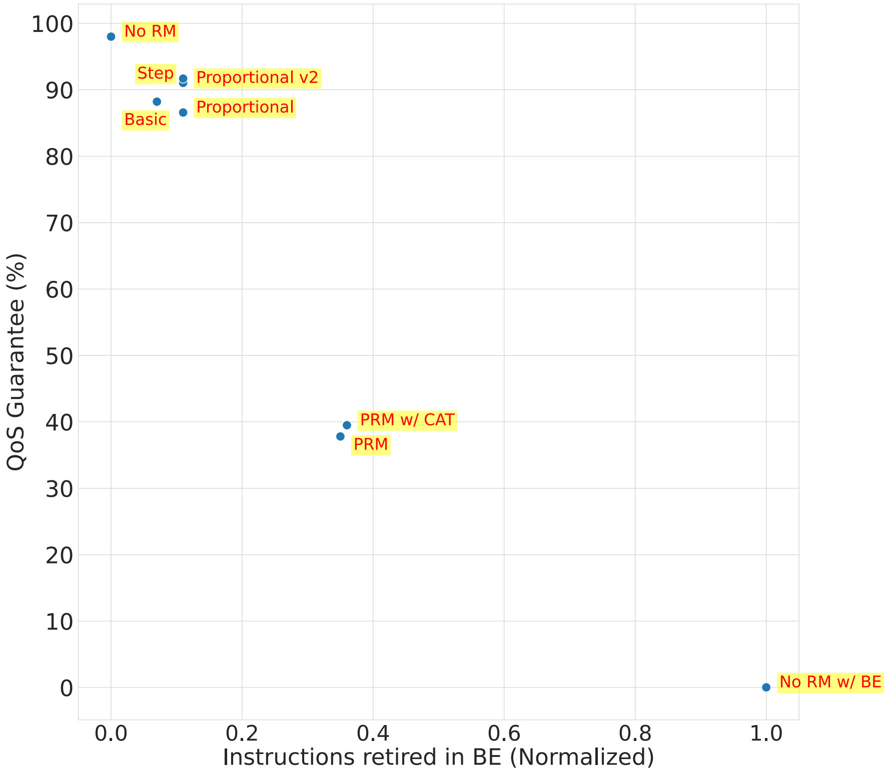


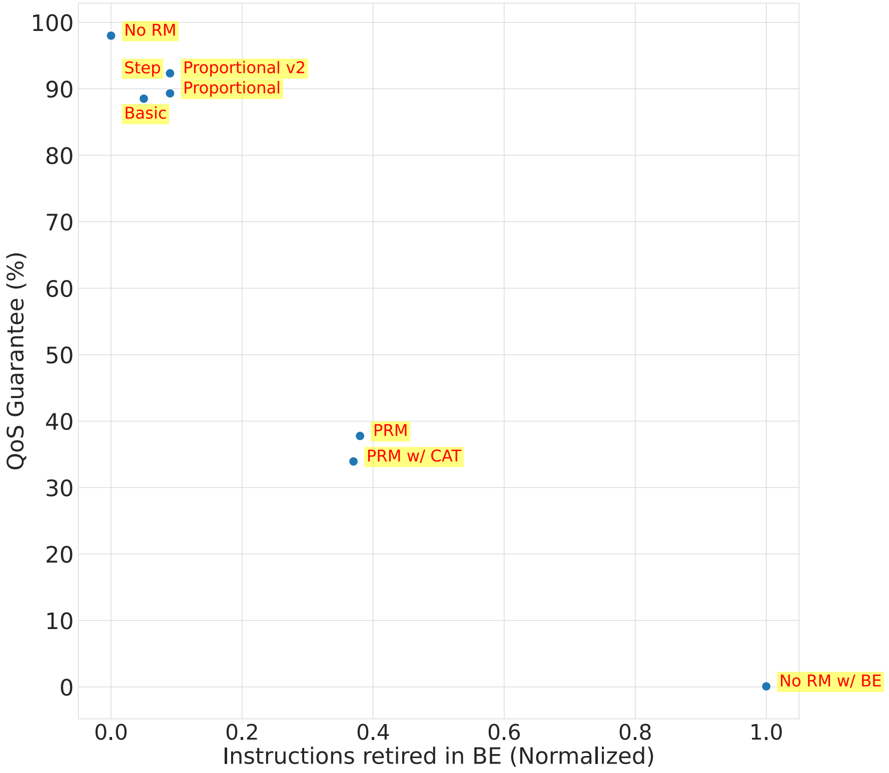
Figure 6.5: Instructions retired for each benchmark, normalized to the No RM w/ BE benchmark. Values for NAMD and fotonik3d respectively are shown on the left, while their average is shown on the right.

Figure 6.6 illustrates the tradeoff between the QoS and throughput. Here, the QoS guarantee and instructions retired are shown as a scatter plot, with each benchmark indicated by a labeled dot. An imaginary, ideal resource manager would lie in the upper right corner of the graph. However, this level of performance is most likely impossible to reach in practice. The upper bound on the performance would have to be determined experimentally. The benchmarks cluster into two distinct groups corresponding to the PRM and Q-PRM benchmarks. Still, there are some notable deviations from the general trend, within each group. For example, PRM w/ CAT outperforms PRM in terms of both QoS and throughput when fotonik3d is used, while the opposite is true when NAMD is used. Furthermore, Proportional v2 manages to achieve a higher guarantee than Proportional without doing any less best-effort work.

Overall, the PRM performs better when LLC regulation (CAT) is disabled. For the Q-PRM, the best controllers overall are Step and Proportional v2, which manage to achieve an average QoS guarantee of 92.0% and 91.7% respectively.



(a) fotonik3d



(b) NAMD

Figure 6.6: QoS Guarantee over Instructions retired

Chapter 7

Discussion

7.1 Initial Observations

Much of our initial time was spent experimenting with different benchmark configurations and graphing the results over time. We wanted to understand how Memcached behaves under a dynamic load, and how it reacts to resource contention. Furthermore, we wanted to observe how the PRM responds to a changing load on the system, and the effects of its actions on the QoS. In order to understand this behavior, it was important to plot the collected data over time.

Figure 7.1, from a single run of the `No RM` benchmark, illustrates how Memcached responds to a dynamic load when there is no best-effort interference. The QoS is roughly proportional to the RPS as long as the load stays within the range that Memcached can handle.[1] We can also see that the QoS is very unstable in a short time frame, but that it really oscillates around a more stable average. The QoS target is indicated by a red line in the following graphs.

Figure 7.2 is drawn from a single run of the `PRM` benchmark, using `NAMD` as the best-effort workload. The figure shows the QoS of Memcached along with the CPU quota of `NAMD`. The load generator also reports the number of *unfulfilled requests* every second. These are the requests that have been sent by the client, but have not yet been responded to. This statistic is not the same as the number of *dropped requests*, which are the requests that are never responded to. Unsurprisingly, we observe that the QoS is loosely correlated with the number of unfulfilled requests. We expect to see this because, when the QoS is high relative to the RPS, the rate of incoming responses will be lower than the rate of outgoing requests (the load), and the queue of unfulfilled requests will grow. We see that when the QoS drops suddenly, the queue is emptied very quickly.

[1] This relationship actually becomes nonlinear as the load approaches the max load. The QoS grows exponentially above the max load.

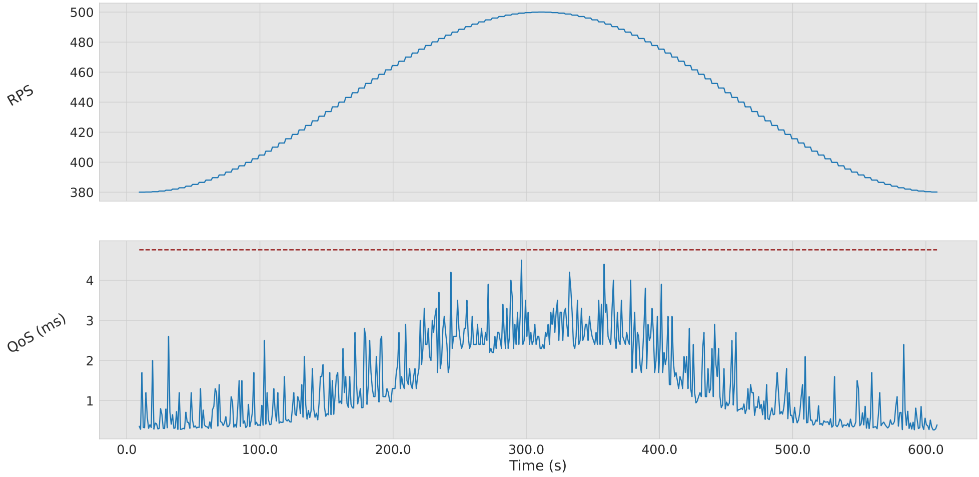


Figure 7.1: No RM: Load (RPS) and QoS over a 10 minute bechmark (QoS target indicated by red line). This shows how Memcached responds to a dynamic load in the absence of resource contention.

7.1.1 CPU Quota vs. Cache Allocation

Figure 7.2 also tells us that the CPU quota of the best-effort task is actually effective as a control mechanism, in the sense that it directly impacts Memcached's quality of service. We see that whenever the best-effort quota is throttled, the QoS improves instantly. When it is increased again toward the end of the run, the QoS target is immediately violated. The QoS is sensitive to even small adjustments to the quota, e.g. when going from resource level 0 to level 1. On the other hand, we don't see the same pattern with LLC allocation. There is no obvious relationship between the number of cache ways allocated to the best-effort task and the QoS of the latency-critical service.

Another observation is that even when the PRM throttles and holds the quota at level 0 (which corresponds to a CPU time of 1%) for an extended period, it is not able to meet the QoS target. Partly, this is because the load is close to the max load (even the No RM benchmark violates the target for some runs), but this does not explain the entire difference. In general, we find that there is a significant amount of contention between the latency-critical and best-effort workloads, even when the best-effort task is limited to a minimum amount of execution time. This is problematic because it means that the PRM is not able to scale the best-effort workload over its full range (from no execution to unlimited execution). Since we use such a high load for these experiments, the resource manager is not able to uphold the QoS guarantee.

Our initial hypothesis to explain this behavior was that as long as the best-effort task is allowed any amount of execution time, and its LLC occupancy is unrestricted, it will contend for memory bandwidth with the latency-critical service. Hence, we expected that the PRM should be able to manage the QoS at least

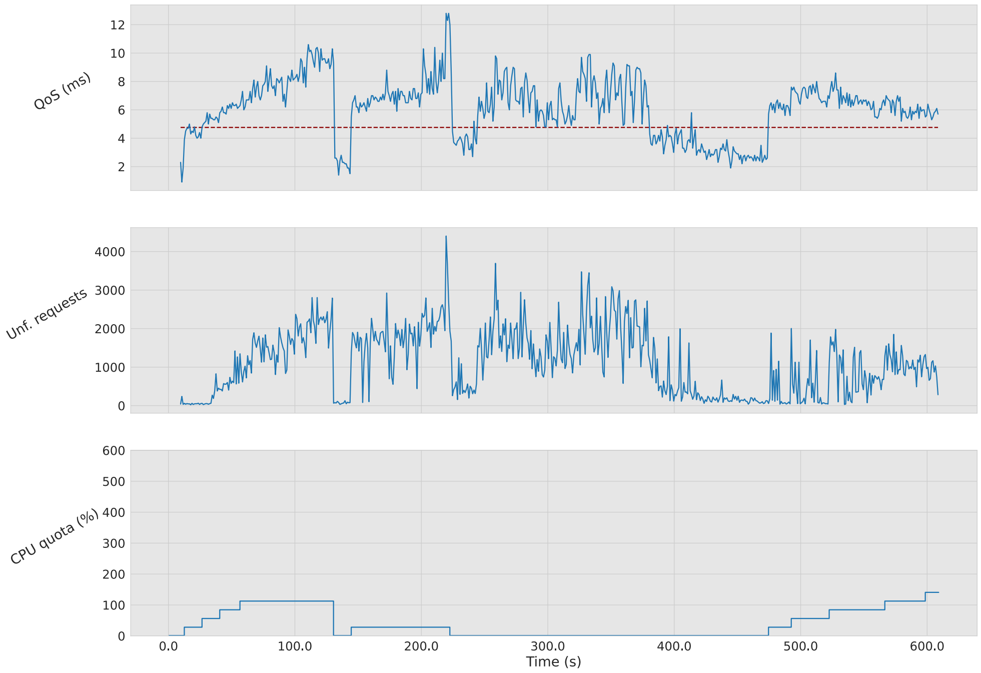


Figure 7.2: From the PRM benchmark using NAMD: the QoS and unfulfilled requests of Memcached, and the CPU quota of NAMD, over time

by employing the LLC mechanism (as with the PRM w/ CAT benchmark). Surprisingly, we found that even when both resources are minimized, the interference is not eliminated. Presumably, the only way to completely eliminate the interference would be to afford the best-effort task no execution time at all. However, the Intel Platform Resource Manager uses a minimum CPU quota corresponding to 1% CPU time, and we did not find the time to modify this for the Q-PRM (and repeat our benchmarks).

To investigate this we performed an experiment where we compared the effectiveness of minimizing the CPU quota alone, versus minimizing both the CPU quota and LLC allocation. Figure 7.3 shows a comparison of the QoS for these configurations. The blue line represents the baseline QoS when Memcached runs by itself, while for the orange and green lines Memcached is colocated with fotonik3d. The orange line shows the QoS when the CPU quota is fixed at the minimum level for the duration of the run; the green line shows the QoS when both the CPU quota and LLC levels are fixed to the minimum. We see that when the cache access of fotonik3d is restricted to the minimum level (2 cache ways out of 20), the QoS actually degrades. Sampling the QoS guarantee across 10 repeated runs for each of these configurations, we get an average guarantee of 96.7% when only the CPU quota is minimized, versus 69.5% when the LLC is minimized as well.

A possible explanation is that the best-effort task may not be able to fit its working set in the last-level cache when it is limited to only 2 cache ways. This

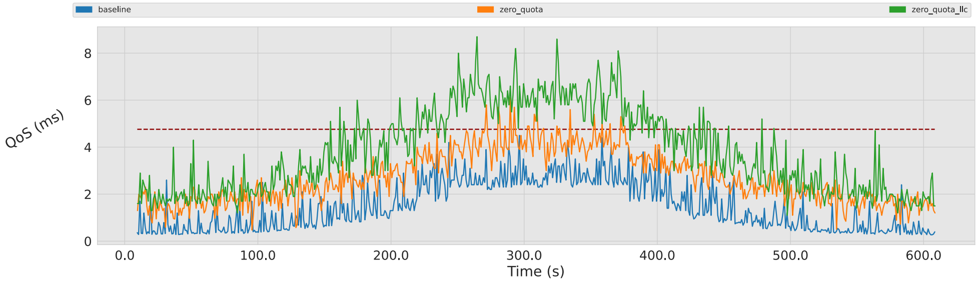


Figure 7.3: Memcached's QoS over time for three different benchmarks. The blue line shows the baseline, where there is no best-effort workload to interfere with Memcached. The orange line shows the impact of fotonik3d with 1% CPU quota and unrestricted cache access. The green line shows the impact of fotonik3d with 1% CPU quota and 2 cache ways.

could lead to a higher cache miss ratio and memory access rate for the best-effort task, which would mean a higher level of memory bandwidth contention. If this is the case, it shows that LLC allocation is less predictable than CPU quota as a control mechanism, and may even work against its purpose. Compared to the CPU quota, it may be more dependent upon the particular characteristics and behavior of the best-effort workload.

7.2 Evaluation

In this section we evaluate the resource managers/controllers based our main experimental results. We focus on what should be their two primary goals: when the load is high, they should prevent a QoS violation by down-regulating the best-effort workload; when the load is low, they should exploit the available headroom by up-regulating the best-effort workload.

Overall, our results are similar for each of the best-effort workloads. The average guarantees for NAMD and fotonik3d are 63.2% and 64.0% respectively. We had hoped that there would be more of a difference between their interference patterns, but apparently there is not, at least for Memcached. fotonik3d does however seem slightly easier to manage in terms of interference.

7.2.1 Intel PRM

In Section 6.2.2 we found that the PRM achieves a very low QoS guarantee. There are three main points that we want to make about its behavior:

- The PRM does not accurately detect CPU contention, and therefore responds too late to QoS violations.
- LLC contention is never detected. However, this does not have a large impact on the QoS.

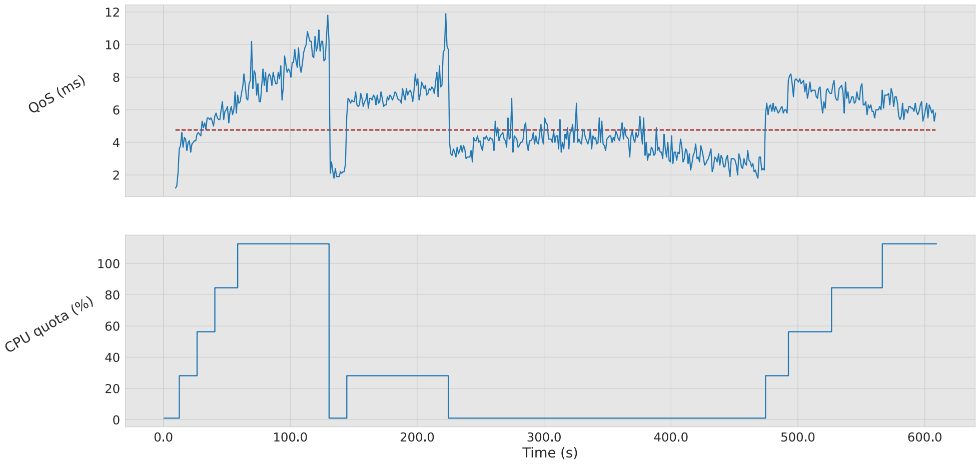


Figure 7.4: From the PRM benchmark using fotonik3d: QoS for Memcached and CPU quota for fotonik3d over time. The PRM is not able to detect QoS violations in time.

- The NaiveController algorithm is simplistic, but is for the most part not to blame for the PRM's poor performance in this experiment.

The first point is illustrated by Figure 7.4, which show the QoS and CPU quota from a PRM run with fotonik3d. We see that the PRM increases the quota even after a violation occurs within the first minute. Only when the QoS reaches about 12 ms (approx. 2.5 tardiness), is the quota reduced. Now, of course the PRM is not actually aware of the QoS violation taking place. Instead, it has to decide whether there is CPU contention, and this is based only on the measured CPU utilization. The PRM concept of contention acts as a proxy for the QoS violation, but these do not necessarily correspond.

In general, we find that when the best-effort quota is increased, Memcached's CPU utilization slightly drops because there is more CPU time contention, and because its resource requirements are the same. Still, since the best-effort workload is allowed more CPU time, the total CPU utilization of the workload colocation increases. This total is what the PRM compares to the CPU threshold. Evidently the PRM *does* respond to the CPU utilization, because the quota is scaled up around the beginning and the end of the run. This is when the load on Memcached, and in turn its CPU utilization, is the lowest. The problem is simply that the PRM's operating CPU threshold is too high.

Figure 7.5 shows the QoS, CPU quota and LLC resource levels from a PRM w/ CAT run with fotonik3d. The PRM increases the LLC resource level by one at a regular interval of 2 minutes. This period corresponds to the LLC decision interval (20 seconds) times the LLC cycle threshold (6 cycles). The LLC level is never down-regulated because cache contention is never detected by the PRM. Although this could have been caused by an issue with our profiling method, it does not significantly impact the QoS, as we discussed in the previous section. Comparing

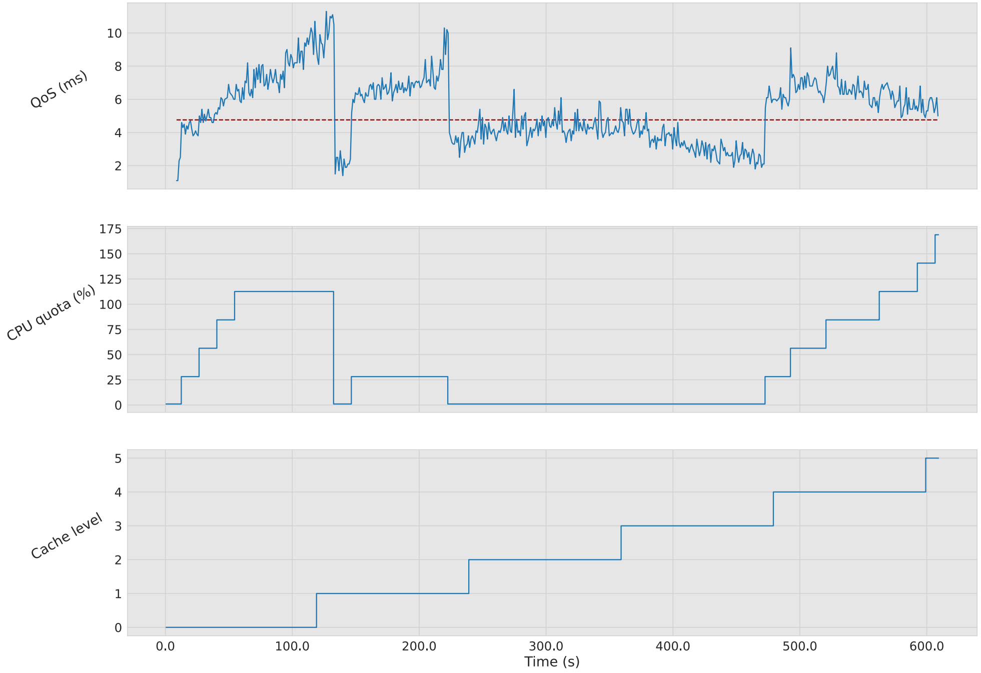


Figure 7.5: From the PRM w/ CAT benchmark using fotonik3d: QoS for Memcached, and CPU quota and LLC resource level for fotonik3d. Cache contention is never detected, and consequently the LLC level is increased monotonically.

Figures 7.4 and 7.5, the overall pattern is the same.

In general, the NaiveController controller is overly cautious about prioritizing QoS over throughput. In earlier experiments we found that when the load is low, and there is sufficient opportunity to perform best-effort work, the algorithm takes too long to reach the optimal level of CPU quota. Under the conditions here, where there is very little QoS slack, that works in favor of the PRM. At the same time, the algorithm is too reckless. After the quota has been throttled, the controller attempts to increase the quota level at the next decision interval. We see that this leads to an immediate QoS violation in the aforementioned runs. This could have been avoided by going into a cooldown period after contention has been detected.

7.2.2 Q-PRM

Proportional vs. Proportional v2

From Figure 7.6, we see that while both controllers exhibit oscillatory behavior, Proportional v2 clearly oscillates to a lesser extent. This discrepancy is explained by the fact thet Proportional v2 uses smaller CPU quota adjustments when the load is low, in order to increase resolution. Examining the behavior from $Time = 100$ to $Time = 200$ in the figure, the difference is very clear. Proportional's oscillary amplitude is much higher, which leads it to frequently violating the QoS guaran-

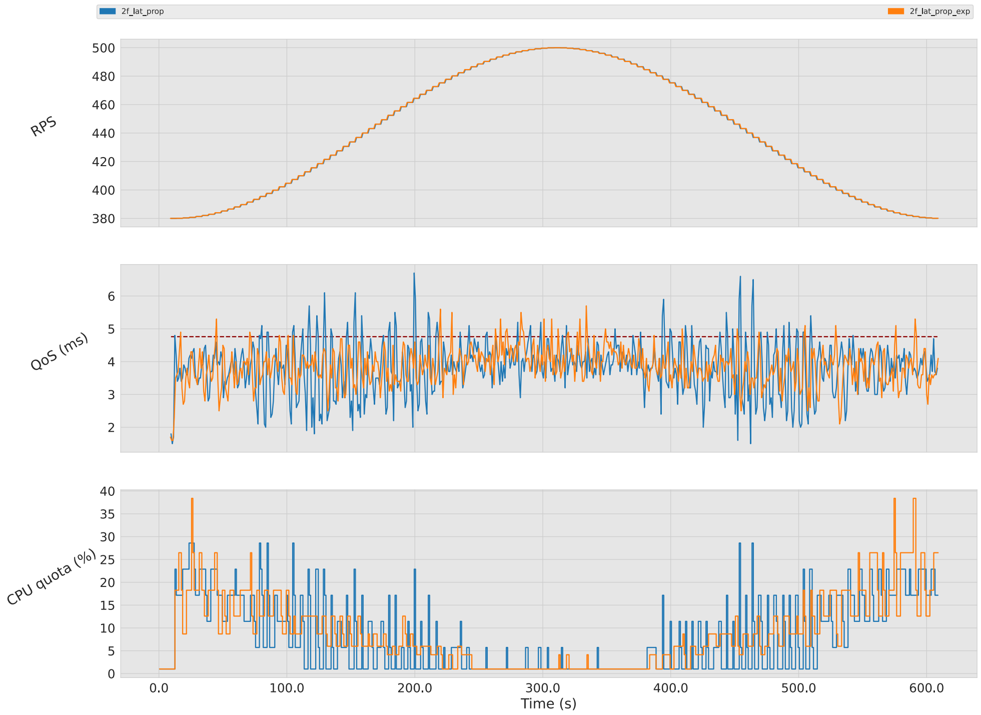


Figure 7.6: Comparison of the proportional benchmarks using fotonik3d. Proportional v2 in **orange**, and Proportional in **blue**

tee. Examining the start and the end of the time series, however, we see that they largely behave in the same manner.

Holistic Evaluation

Evaluating our own controllers, while there are differences, it should be noted that their overall results are quite similar. All of the controllers are able to perform notably better than the PRM, and they all converge upon a relatively stable CPU quota value.

There are some questions as to how much the nature of the load curve impacts our results. Step, our algorithmically simplest controller, responds very well to sinusoidal load curves due to only incrementing/decrementing the resource level by one. However, if our curve was rapidly and abruptly changing the load in a step-function manner, Step would be slow to converge and likely violate QoS guarantees to a much larger degree when the load suddenly increases. In this environment, would likely thrive, due to its awareness of not only if the QoS guarantee is breached, but also the magnitude of the violation.

The ideal controller would be able to adapt to a wide variety of load patterns. It would likely encapsulate some sort of latency history, in order to determine if the load is smoothly increasing or decreasing, which in a real-world scenario would correspond to diurnal patterns. In such a scenario, Step-like behavior would be very helpful. In case of rapid changes or load instability, it should err on the side of the latency-critical jobs, exhibiting more -like behavior.

7.3 Analysis

7.3.1 QoS vs. Throughput

The first priority of the resource manager should be to ensure that the QoS target is met. In other words, the QoS guarantee is more important than instructions retired as an evaluation metric. A resouce manager that achieves a high throughput by sacrificing the QoS guarantee is terrible. On the other hand, the maximum guarantee is achieved by not doing any best-effort work at all.

We considered quantifying the relative importance of these metrics by coming up with some sort of weighting scheme, but decided to do our evaluation qualitatively instead. One could argue that the QoS target is an absolute requirement that must be met at all costs. In this sense, a 99% guarantee with a high throughput is worse than a 100% guarantee with no work done. On the other hand, the best-effort task may be considered essential in that it must be completed at some point. The difference is that the best-effort task is not time sensitive, and is ideally worked on when the load is low. Of course, if the guarantee is the same, a higher throughput is preferable because it represents power savings. In the end, what we are really interested in is a higher total energy efficiency, and the ultimate test of a resource manager would be to compare the differences in power usage.

It should be noted that the load curve we use here is never lower than 76% of the max load. For more than half of the runtime of a benchmark, the correct decision is to completely throttle the best-effort CPU quota. This is perhaps more intense than a typical real world scenario, where either one of the workloads could be migrated to another cluster node. Such a migration would result in less contention on each server, but a lower utilization and energy efficiency.

7.3.2 What Is the Significance of Our Results?

We have shown that we can achieve a higher QoS guarantee than the PRM by using a QoS-aware approach. The main reason for this is that we are much more accurate at detecting a QoS violation than the conventional, metrics-based approach. In one way, this is almost a trivial result, because while the PRM has to estimate the QoS, the Q-PRM is provided with the answer. The Q-PRM has an advantage because its algorithm is based on the QoS target, which defines our evaluation metrics.

Our results show some degree of variation between individual runs. In other words, the same run configuration does not always produce the same results. There are many potential causes for this, and we have not investigated this in depth. For example, variation could be caused by background jobs in the operating system, interrupts, network traffic etc. We also suspect that there some inherent flaws in the CloudSuite Memcached load generator. To mitigate this situation, we took multiple samples for each benchmark, especially for the load testing benchmark to find the QoS target. However, for the dynamic benchmarks the sample size may somewhat small.

The PRM may be capable of better performance under other conditions than what we demonstrate here. In practice, the CPU utilization/quota system does all of the meaningful work. The LLC resource was never down-regulated in our experimental runs. Even if it were, it is doubtful that it would be very effective as a control mechanism. So, it is possible that we could do a different profiling, or use longer lasting benchmarks. Maybe then the PRM is able to detect LLC contention and throttle the LLC. Anyway, its main problem seems to be that it does not detect CPU contention. The only profiling parameter for this is the CPU threshold. We realized in the later part of this project that we could have manually set the CPU threshold to a lower value. This would have improved the PRM's performance significantly, because it would then be able to detect QoS violations more accurately and down-regulate the best-effort workload more quickly, leading to a higher QoS guarantee. But the scientific value of such a result is questionable. We would be artificially helping the PRM to do its job by tuning it to our specific workload and the results we observed, instead of evaluating the system itself. The entire point of using a resource manager in the first place is that is can be deployed to manage an arbitrary workload without human intervention.

It should be mentioned that in some of our earlier experiments, when a less intensive load curve was used, and `stress-ng` was used as the best-effort workload, the PRM did better than in our final results. These results were not included in this thesis only because of the practical difficulty and time that would be needed to reproduce our earlier setup. Hence these results will have to remain anecdotal. In any case, the resource managers were compared in an identical scenario in the final experiment.

Considerable effort was spent trying to set up the PRM so that it would perform at its best. The fact that it only works well in specific conditions could be considered an advantage of QoS-aware resource management. The PRM is not the most advanced conventional resource manager available, and others (e.g. Twig) may be able to generalize better. However, QoS-aware resource managers may need less profiling to work, seeing as they use the QoS directly and do not have to rely on extensive offline profiling in order to be application-agnostic. If that is the case, we would argue that our results demonstrate one of the main advantages to the QoS-aware approach.

A major weakness of our work is that we were only able to evaluate our resource manager with a single latency-critical service. Our claim is that QoS-aware resource managers require less sophistication, compared to conventional resource managers, to adapt to different workload colocations. If we had tested the Q-PRM for at least two different applications, this argument would have been stronger. Furthermore, we could have included a variety of load curves in our final experiment. Seeing as different controllers thrive under different circumstances, this would allow us to compare the resource managers and algorithms more generally.

Our results demonstrate that QoS-awareness allows for accurate detection of QoS violations. Our setup is somewhat artificial, considering that the QoS is communicated to the resource manager using a custom client on the same host. However, it should be possible to implement QoS-awareness in a production system. For example, the resource manager could estimate the QoS by timing the TCP connections established on the server.

7.4 Future Work

A lot of improvements could have been made in order to make our controllers perform better, as well as more capable of generalizing to other workload colocations. Future work could involve using more control mechanisms, for example cache occupancy and CPU frequency. Futhermore, experiments should be expanded to integrate different load patterns, in order to show that our system can deal with sudden change in the load. It would also be interesting to use another latency-critical service, in order to show that our approach is generalizable. Going even further, being able to dynamically start new latency-critical and best-effort tasks during runtime would be a boon to a practical implementation in the real world.

Chapter 8

Conclusion

In conclusion, we have presented Q-PRM, a modified verison of the Intel Platform resource Manager. We have shown that by using real-time QoS measurements, we can for the most part uphold the QoS guarantee while still performing a decent chunk of best-effort work. Our approach improved upon the existing framework in the Intel Platform Resource manager, increasing the QoS guarantee from 37.8% to 92.4%. We have also discussed which metrics we can utilize to evaluate resource managers, and the extent to which they are relevant. Lastly, we acknowledge that our model and experimental setup is not without its flaws, and openly discuss those shortcomings.

Bibliography

[1] K. Konstantinos, P. Mitropoulou, E. Filiopoulou, C. Michalakelis and M. Nikolaidou, 'Cloud computing and economic growth,' Oct. 2015. DOI: 10.1145/2801948.2802000.

[2] T. DeStefano, R. Kneller and J. Timmis, 'Cloud computing and firm growth,' University of Nottingham, GEP, Discussion Papers 2019-09, Sep. 2019. [Online]. Available: https://ideas.repec.org/p/not/notgep/2019-09.html.

[3] J. Dean and L. A. Barroso, 'The tail at scale,' *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[4] D. Byrne, C. Corrado and D. E. Sichel, 'The rise of cloud computing: Minding your p´s, q´s and k´s,' National Bureau of Economic Research, Working Paper 25188, Oct. 2018. DOI: 10.3386/w25188. [Online]. Available: http://www.nber.org/papers/w25188.

[5] 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.

[6] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso and C. Kozyrakis, 'Towards energy proportionality for large-scale latency-critical workloads,' in *Proceedings of the 41th Annual International Symposium on Computer Architecture*, 2014.

[7] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang and J. Wang, 'Perfiso: Performance isolation for commercial latency-sensitive services,' in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, Jul. 2018, pp. 519–532, ISBN: 978-1-939133-01-4. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/iorgulescu.

[8] J. Zhao, H. Cui, J. Xue and X. Feng, 'Predicting cross-core performance interference on multicore processors with regression analysis,' *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1443–1456, 2016. DOI: 10.1109/TPDS.2015.2442983.

[9] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung and V. Venkataramani, 'Scaling memcache at facebook,' in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL: USENIX Association, Apr. 2013, pp. 385–398, ISBN: 978-1-931971-00-3. [Online]. Available: `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala`.

[10] H. Kasture, D. B. Bartolini, N. Beckmann and D. Sanchez, 'Rubik: Fast analytical power management for latency-critical systems,' in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 598–610. DOI: `10.1145/2830772.2830797`.

[11] S. Chen, C. Delimitrou and J. F. Martınez, '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*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 107–120, ISBN: 9781450362405. DOI: `10.1145/3297858.3304005`. [Online]. Available: `https://doi.org/10.1145/3297858.3304005`.

[12] B. Fitzpatrick, 'Distributed caching with memcached,' *Linux J.*, vol. 2004, no. 124, p. 5, Aug. 2004, ISSN: 1075-3583.

[13] J. Yang, Y. Yue and K. V. Rashmi, 'A large scale analysis of hundreds of in-memory cache clusters at twitter,' in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 191–208, ISBN: 978-1-939133-19-9. [Online]. Available: `https://www.usenix.org/conference/osdi20/presentation/yang`.

[14] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan and C. Kozyrakis, 'Heracles: Improving resource efficiency at scale,' in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 450–462. DOI: `10.1145/2749469.2749475`.

[15] V. Petrucci, M. Laurenzano, J. Doherty, Y. Zhang, D. Mossé, J. Mars and L. Tang, 'Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers,' Feb. 2015. DOI: `10.1109/HPCA.2015.7056037`.

[16] Intel, *Intel cmt/cat*, `https://github.com/intel/intel-cmt-cat`, 2021.

[17] M. Själander, M. Jahre, G. Tufte and N. Reissmann, *EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure*, 2019. arXiv: `1912.05848 [cs.DC]`.

[18] E. PARSA. (2021). 'CloudSuite: A Benchmark Suite for Cloud Services,' [Online]. Available: `https://www.cloudsuite.ch/` (visited on 09/06/2021).

[19] R. Nishtala, V. Petrucci, P. Carpenter and M. Sjalander, ‘Twig: Multi-agent task management for colocated latency-critical cloud services,’ in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 167–179. DOI: `10.1109/HPCA47549.2020.00023`.

NTNU

Norwegian University of
Science and Technology