Achieving Service Level Objectives for Latency-Critical Cloud Services by Exploiting Various Forms of Heterogeneity in Server Clusters

By

SAMBIT KUMAR SHUKLA
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Dipak Ghosal, Co-Chair

Matthew K. Farrens, Co-Chair

Chen-Nee Chuah

Committee in Charge

2022

DEDICATION

I dedicate this thesis to my dear wife, Marry, who encouraged me to repursue my doctorate at UC Davis after being unable to complete it at UC Riverside. Despite two very difficult pregnancies and multiple heart-breaking challenges along this doctoral journey, she stepped up to take over my parental responsibilities so that I could make academic progress whenever I was falling behind. I dedicate this thesis to my two beautiful kids: my daughter, Shaarya, and my son, Shiv. 4-year old Shaarya has significant cognitive and physical disabilities and is undergoing ancillary therapies. I am hopeful that some day she will make enough cognitive progress to be able to comprehend my thesis and possibly write her own. My 2-year old son, Shiv, has a rare and terminal genetic disorder called Nemaline Myopathy Type-5 and is past the 90th percentile life expectancy for his specific condition. Thankfully, I was fortunate enough to give him my time and love during these past couple years. My kids have been both an impediment and a great motivation towards completing my research. I also dedicate this thesis to my parents, my mom, Sumana Pati, and my dad, Gyanchand Shukla, who despite their humble origins, have toiled life-long in their respective academic careers to offer me a socially secure environment along my academic journey.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I wish to extend my sincere gratitude to my advisor and Committee Co-chair, Prof. Dipak Ghosal, who has been both a mentor and a friend along this difficult doctoral journey. He was always promptly available to discuss research details. On numerous occasions, he guided me with honing my research goals when I was led astray by the complexities and technical details of my research. He has also been very patient, understanding, sympathetic and supportive during my periods of distress over the last few years. He deserves the most credits for me being able to complete this doctoral journey.

I also thank my co-advisor Prof. Matthew Farrens for financially supporting my research. His numerous hours of intense proof-reading, his eye for detail, his corrective feedback and his high standards for a doctoral thesis has helped me to elevate the standard of this thesis to a reliable literature work for guiding future research.

A word of thanks to the Graduate Advisors, Jessica Stoller and Alyssa Bates, for promptly helping me out with official matters. I would also like to thank all my fellow graduate researchers with whom I had valuable technical discussions that helped me navigate through my research.

Finally, I would like to thank my family for being so supportive of me over the years. Thanks to my wife Marry for encouraging me whenever I was demoralized. Many thanks to my Mom and Dad for their love, prayers, support and well-wishes. A special note of thanks to my father-in-law, Chandrasekhar Sahu, for eagerly supporting my family and sharing my family responsibilities over the past few years. On multiple occasions, he looked out for my family and reassured me to focus on my research. Also I would like to thank my extended family and friends for their physical and emotional support that has helped me stay afloat and complete this doctoral journey.

VITA

Sambit K. Shukla

EDUCATION

Doctor of Philosophy in Computer Science — **2022**
University of California Davis — *Davis, California*

M.S in Computer Science — **2013**
University of California Riverside — *Riverside, California*

Bachelor of Technology in Computer Science — **2008**
National Institute of Technology Rourkela — *Rourkela, Odisha, India*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2016–2022**
University of California, Davis — *Davis, California*

Graduate Research Assistant — **2009–2013**
University of California, Riverside — *Riverside, California*

TEACHING EXPERIENCE

Teaching Assistant — **2018–2020**
University of California, Davis — *Davis, California*

Teaching Assistant — **2012–2013**
University of California, Riverside — *Riverside, California*

REFEREED CONFERENCE PUBLICATIONS

Leveraging Network Delay Variability to Improve QoE of Latency Critical Services **Oct 2021**
15th IEEE International Conference on Networking, Architecture and Storage (NAS)

Understanding and Leveraging Cluster Heterogeneity for Efficient Execution of Cloud Services **Nov 2021**
10th IEEE International Conference on Cloud Networking (CloudNet)

Tuning network I/O processing to achieve performance and energy objectives of latency critical workloads **Oct 2019**
21st IEEE International Conference on High Performance Computing and Communications (HPCC)

Co-optimizing Latency and Energy for IoT services using HMP servers in Fog Clusters **Jun 2019**
4th IEEE International Conference on Fog and Mobile Edge Computing (FMEC)

SOFTWARE

HMP-ClusterSim `https://github.com/sambitshukla/Greeniac/`
Python-based simulator for service execution on heterogeneous clusters of Heterogeneous Multi-Processors (HMPs).

ABSTRACT OF THE DISSERTATION

Achieving Service Level Objectives for Latency-Critical Cloud Services by Exploiting Various Forms of Heterogeneity in Server Clusters

By

Sambit K. Shukla

Doctor of Philosophy in Computer Science

University of California, Davis, 2021

Professor Dipak Ghosal, Co-Chair
Professor Matthew K. Farrens, Co-Chair

Microservice-based deployments of Latency-Critical cloud services (LC-Services) pose a well-studied Tail-Energy co-optimization challenge: meeting Service-Level Objective (SLO) for tail latency of requests while minimizing energy consumption. Prior works have shown that CPU heterogeneity can be exploited at server-level to improve the overall energy-efficiency of LC-Services. In my research work, I show that exploiting CPU heterogeneity at the cluster-level can reap additional benefits. As a part of my research, I propose two control-plane strategies to exploit the CPU heterogeneity at the cluster-level. First, a Reinforcement Learning based technique to perform load balancing across a homogeneous cluster of Heterogeneous Multi-Processors (HMPs). Second, a heuristic-based instance scaling and load balancing technique to perform the co-optimization on a heterogeneous cluster of Symmetric Multi-Processors (SMPs). In addition, I also explore OS-level approaches to achieve tail latency predictability and perform Tail-Energy co-optimization by exploiting the CPU frequency scaling (or, as I call it, the Core-frequency heterogeneity).

During my research, I also inferred that the currently enforced SLOs for LC-Services are cloud-centric instead of being user-centric, i.e. cloud SLOs guarantee intra-cloud latency, but not the user QoE. To guarantee statistical bounds on user QoE, the control plane strategies must account for the external network delay information of requests. I propose a novel user-centric paradigm and implement a user-centric SLO-enforcement framework built on top of Kubernetes that performs QoE-aware instance scaling, load balancing and task scheduling on a homogeneous cluster. I also

demonstrate that significant energy saving can be achieved in this novel paradigm too by exploiting cluster heterogeneity.

Chapter 1

Introduction

1.1 Research Background

Growth of LC-Services In recent years we have observed rapid growth in popularity of interactive devices such as smart-phones, voice assistants, and Augmented Reality(AR)/Virtual Reality(VR) gaming consoles. With further proliferation of IoT, complex use cases involving vehicular traffic networks, warehouse management, smart city management, health care diagnostics among others are also expected to go online in the near future. Implementing these applications require back-end services hosted by cloud service providers (CSPs) on private/public cloud and fog clusters to perform compute- and data-intensive tasks on-the-fly. Typically Application owners (CSP-clients) rent cloud resources form CSPs to host the Application services and service user requests. To increase their revenue, CSP-clients expect to achieve a high Quality of Experience (QoE) for its users. Consequently, CSPs strive to guarantee the QoE expectations of End-users to maximize their revenue from its client base. Since, for interactive services, response latency primarily drives the User QoE, CSPs aim to meet certain response delay target for User requests.

Not surprisingly, human perceptual abilities [1, 2, 3] drive the response time requirements for these Latency-Critical interactive services (LC-Services). For example, response latency bounds of 10ms for AR/VR applications and 100ms for live search applications are required to guarantee an

immersive experience for users. An action-response delay under 100ms for online gaming request is crucial for continuity in visual perception [4]. Requests violating these latency bounds cause significant deterioration in a user's Quality of Experience (QoE) [5]. Historically these latency bounds have been getting lower due to rapid advancement of computing and networking technology coupled with high bandwidth and low latency demands of newer generation of application services. Cloud service deployments are being offloaded in part (or full) from cloud to fog clusters and even further to edge devices. Similarly, Telco providers expect to leverage 5G network and NFV technologies and to offer seamless, truly-interactive next generation services to users through CORD platform server clusters at the edge. In light of these developments, latency bounds are expected to further shrink from 100s to under 10s of milliseconds.

Tail Latency Since guaranteeing latency bounds on every request is impractical due to the non-deterministic nature of server-dynamics, CSPs offer to meet Service Level Objectives(SLOs) that guarantee sub-second bounds on tail latency of response time distribution. Tail latency represents the higher latency range in a statistical distribution of response latencies for a long sequence of service requests. Cloud platforms typically aim to achieve a fixed tail latency SLO target ranging between 95^{th} (or P95) to 99.9^{th} percentile (or P99.9) for hosted services. A SLO requiring 99^{th} percentile tail latency (or P99) target of 1 sec means service provider must ensure at least 99% of client requests are serviced and responded back within 1 sec of request arrival. Primary causes of high latency within a leaf node include queuing delays (at OS and application layer), OS interference (through NIC interrupts, task schedulers and OS daemons), background processes (from other VM/containers) and CPU wakeup delays from sleep states [6]. Recent works have used node level [7] and tier level [8] scheduling, OS customizations [9], sleep management [10] and interference reduction [11] to guarantee SLO.

The Tail-Energy Conundrum To meet such strict objectives, service providers must actively run many LC-Service instances across multiple servers to handle occasional load spikes from unpredictable request arrivals. But due to low energy-proportionality [12] in traditional server platforms, the under-utilized active servers are expected to continue to waste energy worth billions of

dollars [13]. Thus the LC-Services pose a **Tail-Energy conundrum** for CSPs. Actively running fewer servers can save energy but risk violating SLO during load spikes. Wheres a large number of underutilized active servers can significantly increase energy expenses. Both extremes can lead to revenue loss for CSPs and hence need to be balanced for optimal benefit.

Energy Efficiency Concerns To this end, CSPs have striven hard in recent years to maximize the energy-efficiency of cloud servers [14]. Energy-efficiency is defined as the performance achieved per unit watt consumed. Higher efficiency enables service hosting at lower energy costs. Such efforts [14, 11] have been necessitated primarily due to non-energy-proportional cloud servers [15]. More specifically, this energy-inefficiency can be attributed to traditionally used data center server processors (e.g. Intel Haswell Xeons) due to two reasons. First, their power consumption does not scale with utilization. Such cores waste enormous amount of energy during the predominant idle and low utilization phases in a data center. And second, due to their power-hungry design, they have large energy footprint when they operate at high utilization. In recent years, RISC-based ARM architectures with fewer specialized compute units and reduced legacy support, have shown to achieve significantly higher efficiency than the Xeon-like CISC-based brawny x86 server processors [16]. With multiple newly-available options to improve the efficiency of LC-Service execution, further study is needed to explore how LC-Service execution frameworks can best exploit these options to address the Tail-Energy Conundrum.

1.1.1 LC-Service Request Execution in Microservice Architecture

Current day LC-Services are typically implemented as chain of microservices[1]. Over the past decade, cloud application owners have migrated from monolithic to development-friendly microservices-based software architecture. More recently, the cloud workloads have mostly evolved from data-centric Map-Reduce based and task-graph based on-demand jobs to user-centric live-services. Several past work on cloud Services have suggested Tail-Energy optimization techniques specifically for data-centric on-line data-intensive (OLDI) applications [6, 17]. Few other works perform opti-

[1]The words service and microservice is used interchangeably throughout the thesis

mization for task-graph based jobs [18, 19]. But these cannot be applied to the newer generation of service-based deployments. To meet the latency expectation for a request in the service-based framework, it is crucial to optimize the various control-plane resource management components that impact the intra-cloud execution trajectory for a request.

Request Flow in a Microservice-based Deployment Client application requests in the form of HTTP requests are received by a frontend gateway server. CSPs may expose multiple of these frontend servers to the internet and the target frontend server is chosen via DNS Load Balancing or AnyCast IP [20]. A new TCP connection is first set up for every non-connected user client with the frontend server. The frontend servers run Load Balancer routines that serve as reverse-proxy and perform either application-level or network-level load balancing in assigning requests to the backend servers. Based on the HTTP header fields, the Load Balancer can identify the target service for the request. It then uses Service discovery mechanisms to infer the list of active instance. Subsequently, load balancing (application-layer, TCP-layer or network-layer) is performed for the target service to choose one of the instances from the list. For every new frontend connection, a corresponding TCP (backend) connection is set up to the target Service Instance. All subsequent user requests received by the Load Balancer on the new frontend connection are forwarded over the corresponding backend connection to the same target Service Instance. When the Service Instance Scaler starts/stops an instance, the instance is registered/de-registered from the list of active instance and Load Balancing is applied to the updated list for subsequent user requests. At the Service Instance, the request may be queued behind previously waiting requests. A Task scheduler schedules the request execution in a FIFO fashion from the queue. After the request has been processed, the response from the Service Instance is either sent back to the Frontend Load Balancer to be sent back to the user or forwarded to another service's Load Balancer for subsequent processing. After processing through the chain of microservices the final response is eventually sent back to the user and all intermediate backend connections are closed.

In essence, 3 control plane components determine the physical path and execution time taken by the request within a cluster: (i) A *Load Balancer* that selects the physical server to which the request is forwarded. Inefficient balancing can lead to load disparity among servers and thus long

execution time for a request (ii) An *Instance Scaler* that dynamically scales the number of service instances that are load balanced and assigns them to newer servers. Bad scaling decisions can lead to under-utilization or over-utilization of servers, again leading to resource wastage or response time delays respectively. (iii) A server-level *Task Scheduler* that selects the request from the set of waiting requests (typically in a FIFO manner) and schedules it on an idle CPU (or service instance) for execution. The scheduling algorithm, choice of servicing CPU (if multiple CPU types are available for scheduling), the server configuration and resource interference from co-executing application can all impact the execution time of a request within the server.

Addressing the Tail-Energy Conundrum for Services Majority of approaches addressing the Tail-Energy conundrum for LC-Services can be broadly classified as resource scaling and resource sharing. Resource scaling techniques include *auto-scaling* [21, 22] i.e. powering down inactive servers, *CPU scaling* [23] i.e. hosting services on sleeping cores on-demand, *power scaling* [24, 14] i.e. dynamically managing the power budget of CPUs with varying load and *frequency scaling* [25, 26] i.e. dynamically managing active core frequency through DVFS. Resource sharing approaches, on the other hand, try to co-locate non-critical tasks with LC-Service instances on active servers to maximize cloud resource utilization. Scaling of resources curbs energy wastage due to resource over-provisioning. Whereas sharing resources improves the energy efficiency of servers by performing more tasks per server. Since the two approaches are complementary, many studies combine these approaches [11, 27, 25].

An orthogonal approach is the use of energy-efficient heterogeneous computing resources. In recent years, CSPs have deployed specialized computing units such as General-purpose Graphics Processing Units (GPGPUs), Field Programmable Gate Arrays (FPGAs) and Tensor Processing Units (TPUs) for energy-efficient accelerated task execution. However, these specialized hardware accelerators cannot directly execute applications without tedious porting efforts. Use of single-ISA heterogeneous multi-core processors (HMPs) [28, 29] can also offer efficient execution platforms for LC-Service execution. Because both cores use the same instruction set, no changes are required to the existing applications.

All the above strategies address the Tail-Energy conundrum by enhancing one or more of the 3 control plane components [6, 14, 24]. Most of the works [15, 30] identified opportunities for latency reduction and energy optimization at intra-server level. These intra-server works mainly proposed various application-level *task scheduling* strategies [14, 25] that also benefit from server resource configurability. Others proposed cluster-level resource management strategies and frameworks (such as [31, 32]) to be the more generic and effective approach. These cluster level strategies involve effective *scaling* techniques, i.e. runtime load estimation, dynamic resource allocation and service instantiation. Since cluster hardware is assumed to be homogeneous, *load balancing* is traditionally assumed to be a straight-forward equal-distribution strategy and thus has not been much explored for LC-Services.

However this assumption of cluster homogeneity is not valid for current and futuristic cloud platforms. CPU heterogeneity is inevitable in present and future clusters due to the availability of a wide variety of CPU choices. Applying the existing heterogeneity-unaware strategies to these clusters leads to sub-optimal results and presents significant room for further optimization in the LC-Services as discussed below.

1.1.2 Heterogeneity in Cloud Platforms

Growth of Cluster Heterogeneity A seemingly trivial solution to the Tail-Energy Conundrum would be to bring in newer energy-efficient server processors (e.g. AMD Ryzen) with smaller energy footprint and better energy-proportionality. CSPs have been exactly doing that, some deliberately and some accidentally. CSPs must continuously acquire new servers to either expand their server fleet or replace the faulty ones. As a result, servers with newer and more efficient CPUs are installed in cluster alongside the existing older generation servers. Though this might seem to boost the overall energy efficiency of the cluster, there are two glaring caveats. First, many of the newer energy efficient multicores run at a lower base frequency and achieve lower single thread performance. Second, due to heavy capital investment in prior years, CSPs would still continue to run millions of inefficient CPU cores. As a result, a typical cloud server cluster can only grow increasingly heterogeneous over years, employing multiple generations of different

processor types with varying performance and energy efficiency metrics. However, this increasing *Cluster Heterogeneity* owing to multiple CPU types leads to heterogeneity in servicing *capacity* (or throughput) and *energy* overhead of service instances.

Other Forms of Heterogeneity in the Cloud Besides Cluster Heterogeneity, various other forms of heterogeneity could also simultaneously manifest in cloud platforms. *Server Heterogeneity* presented by servers hosting heterogeneous architectures like GPUs, FPGAs or TPUs, is also becoming rapidly mainstream due to performance benefits. *CPU Heterogeneity* presented by Heterogeneous multi-processor(HMPs) like Intel QuickIA and Arm Big.little platforms is a promising platform for efficient execution of applications [33, 34, 35]. HMPs have been tremendously successful in energy-constrained edge devices. Recently they are also being adopted in small-scale Edge and Fog clusters [36]. Researchers have also advocated for their inclusion in future cloud warehouses. Heterogeneity may also be dynamically introduced in Symmetric Multi-Processors (SMPs) servers by scaling CPU core speeds using DVFS techniques. This is termed as *Core-frequency Heterogeneity* in this thesis. A more stricter notion of heterogeneity, called *Co-runner Heterogeneity*, is presented by the authors of [37]. In it, the throughput variations among service instances due to resource interference in their host servers, is also assumed as a form of heterogeneity.

Cluster Heterogeneity and Cloud Workloads Though other forms of heterogeneity have been largely studied and shown to improve efficiency at the server-level, *Cluster Heterogeneity* has been mostly overlooked. As mentioned earlier, popular resource managers (RM) (e.g. Borg [38], Kubernetes [32, 31], etc.) scale the number of LC-Service instances and distribute user request among those instances assuming cluster homogeneity. But past research works [39, 40] have found such naive load distribution to be sub-optimal for performance. Cluster heterogeneity issue has been addressed through custom cluster-level scaling and load-balancing strategies in data-parallel Map-Reduce jobs [41, 42, 43]. It has also been exploited through customized task placements in task-graph based User jobs [19, 18]. But cluster-level strategies for efficient execution of micro-services, is missing from current literature.

Heterogeneity and LC-Services Some recent studies [33, 34, 35] have shown that increasing *CPU Heterogeneity* in cloud warehouses improves energy efficiency for LC-Services. By employing energy-efficient slow processors during low request load and fast but relatively inefficient CPUs during high load, tail latency of requests could be contained and energy usage could be reduced. Like CPU heterogeneity, *Server Heterogeneity* [44] has also shown to improve efficiency of LC-Services. However, in this respect, *Core-heterogeneity* is the most exploited form of heterogeneity. DVFS techniques have been extensively used in literature [25, 17] to reduce cloud energy usage by switching the cores to lower frequency during low load periods.

1.2 Motivation

A vast set of literature have studied the core challenge of *Tail-Energy Problem in LC-Services* over the past decade. But new research opportunities continue to emerge due to evolution of cloud hardware, software architectures, resource management frameworks, application characteristics and User expectations. Since foreseeable future generation of LC-Services will continue to be developed as micro-services, and cluster hardware is expected to get more and more heterogeneous, control plane strategies need to be adapted to meet the SLOs energy-efficiently. In this context, following are some of the challenges, opportunities and paradigms that are left unexplored in past research works.

1.2.1 Cluster-level Optimization using CPU and Cluster Heterogeneity

Most of the prior works exploiting heterogeneity for LC-Services are server-level and application-level approaches. Cluster-level control plane components remain oblivious to underlying heterogeneity. For instance, advantages of heterogeneity have been established in prior works at the server-level using HMPs, this benefit has not been exploited for LC-Services at the cluster level. By maximally utilizing all small cores and employing minimal number of large cores, further energy could be saved for a given request load. Our preliminary studies (shown in Chapter 2.3) revealed significant energy-saving could be achieved through a cluster level management of request distri-

bution among HMPs. However, unlike server-level scheduling on HMPs [33, 34, 35], requests and service instances cannot be migrated between servers without incurring significant latency overhead that can be detrimental for service SLOs. Only by coordinating cluster-level and server-level control plane strategies, this benefit could be availed on a cluster of HMPs. To implement such a generic strategy it is crucial to automatically learn about the throughput and efficiency achieved by LC-Services on the HMP servers in various possible configurations supported by Core-frequency Heterogeneity and CPU Heterogeneity. Reinforcement learning techniques [45] could be employed to learn the optimal configurations for the clusters in the run-time and then used to perform adaptive scaling and load balancing of the requests (Chapter 2).

Like HMPs, the benefits of heterogeneity could also be reaped for *Cluster heterogeneity* through cluster-level strategies. Unlike HMPs, where the slow cores are also the efficient ones, cloud clusters may host variety of CPUs varying in both dimensions: Capacity and Efficiency. A heterogeneity-aware cluster management must be generic enough to accommodate a variety of CPUs along both dimensions of heterogeneity. However, a thorough impact analysis of such *Cluster Heterogeneity* on aggregate *Tail latency* and *Energy footprint* of a LC-Service is missing from current literature. Such an analysis can lead to models and heuristics for devising efficient control plane strategies. As an alternative to generic and time-consuming learning techniques, service-specific heuristics inferred from LC-Service characteristics may be also used to develop cluster-level and heterogeneity-aware load balancing and instance scaling strategies (Chapter 3).

1.2.2 Server-level Optimization using Core-frequency Heterogeneity

Prior Server-level approaches have mostly focused on controlling the application-level execution time for user requests. Non-application-level workload-dependent latency sources in OS and hardware such as interrupts and OS queuing delays have been traditionally ignored. This is due to their non-significant microsecond-order contribution to end-system latency in earlier workload traces [14]. But this assumption must be reviewed for next generation of microservice-based LC-Service workloads. Booming popularity of micro-service based computing model in commodity clouds and network function virtualization in Telco-clouds has been shrinking the LC-Service execution time in end

systems. Hence previously neglected microsecond-order application-independent latency sources must be analyzed for their impact on tail latency. Another concern with most literature works in the domain is that their proposed approaches are customized for Hyperscale and Service-provider owned clouds; thus employing either application-specific profiling or Hardware/OS enhancements. On the contrary, majority of cloud warehouses are small private clouds hosting fewer workloads. Hence they have limited scope for energy optimization approaches such as task co-execution or Hardware/OS customization. Thus optimizing end-system service latency needs an individual analysis of each latency source along the LC-Service request path within a cloud server. But none of the previous works perform an in-depth impact-analysis of tail-elongating factors for LC-Services. The impact of these factor on latency tail needs to be studied and application-independent generic approaches need to be proposed to reduce their negative impact (Chapter 4).

1.2.3 Limitations of Current Cloud-centric SLOs

The above mentioned research opportunities benefit LC-Service execution in the widely accepted cloud-centric SLO paradigm. But to truly guarantee high user QoE, CSPs must adopt a paradigm shift and guarantee User-centric SLOs instead. Recent user feedback-based studies [46] suggest that QoE for LC-Services decreases as end-to-end delay (E2E delay) increases, following the shape of a user QoE sigmoid function (where E2E delay is the LC-Service response time measured at the user end). Hence to guarantee a minimum QoE threshold, CSP-clients expect E2E delay to be bounded by the corresponding value given by the sigmoid-shaped *QoE-E2E delay* function (Figure 5.2). However, modern CSPs only guarantee Service Level Objectives (SLOs) for intra-cloud delays (i.e., LC-Service response time measured at cloud gateway) - not the E2E delay. A typical SLO is specified as the 99th percentile (or P99) of intra-cloud delay (IC delay) not exceeding some pre-specified delay target (e.g., 1 second). But such server-centric SLO-enforcement ignores the external network delay experienced by requests. Since LC-Service users typically expect sub-second E2E delays, standard external network delays in the order of 10s-100s of milliseconds can significantly impact user QoE. Additionally since external network delays vary across users and over time, the intra-cloud delay guarantees do not translate to E2E delay guarantees. Thus there is an inherent

disconnect between user QoE and CSP's SLO. To guarantee the E2E delay bounds expected by CSP-clients, CSPs must adopt a user-centric SLO enforcement that accounts for external network delay. Such a paradigm would offers a unique opportunity to slow down the execution time of some of the requests with low external network delay in favor of the ones with higher delays that are at higher risk of QoE violation. Efficiently implementing such a paradigm requires reinventing control plane strategies for scheduling, scaling and load-balancing (as discussed in Chapter 5).

1.2.4 Exploiting Cluster Heterogeneity to Efficiently Meet User-centric SLOs

Implementing a new user-centric paradigm brings in unique challenges with cluster heterogeneity too. Unlike a cloud-centric approach, in this new paradigm, incoming requests would have non-identical service headrooms due to variable external network delay. Assigning shorter headroom requests to slower CPUs could risk SLO violation for the cluster. Thus request-load based load-distribution strategies for heterogeneous CPUs [33, 34] would not be effective in addressing our core challenge. Rather, novel headroom-aware and heterogeneity-aware load balancing strategies need to be employed at cluster-level in this new paradigm (as discussed in Chapter 6).

In the following section I enumerate the specific research problems I looked into as a part of my doctoral work.

1.3 Research Goals

Like several prior research works, the core objective of my research is to address the dual optimization challenge of minimizing energy usage while meeting Tail latency SLO for LC-Services. However the scope of my work is unique as I attempt to explore the following problems.

- In past research, only Server-level optimization strategies have been explored for execution of LC-Services on HMP-based servers (i.e. CPU Heterogeneity). Are there opportunities for further optimization through Cluster-level strategies? If so, what control-plane strategies

must be adopted to optimize the execution of services on such a cluster? How do we also accommodate for Core-frequency Heterogeneity in such cluster-level strategies?

- Cluster Heterogeneity presents a two-dimensional heterogeneity, i.e. consisting of a collection of SMPs with both varying throughput and power profiles. Thus it requires more generic cluster-level control plane strategies. What control-plane strategies must be adopted to address this 2-D heterogeneity?

- Interrupts and OS-level queuing are the two application-independent, but load-dependent latency sources that impact the performance of micro-services. What is their measure of impact on tail latency, predictability and energy efficiency of LC-Services? What Server-level OS-managed techniques and scheduling strategies can we adopt to mitigate the impact? Can core-heterogeneity (DVFS) be exploited by these server-level techniques?

- Current cloud SLOs guaranteed by CSPs guarantee Intra-cloud execution latency. Do they really meet user QoE expectations? What novel paradigm and SLO definitions must be adopted to guarantee user QoE targets? What control plane strategies must be developed to implement the novel user-centric paradigm?

- Cluster heterogeneity can also pose challenge to meeting user-centric SLOs in this new paradigm. What cluster-level control plane strategies must be adopted to meet this challenge? How can cluster heterogeneity be exploited to reduce the energy overhead of LC-Services in this new paradigm?

1.3.1 Scope of the Research

As mentioned before, the core objective throughout my research is to co-optimize Tail latency and Energy usage of LC-Services. I explore opportunities for improving efficiency at cluster level by exploiting cluster-level (servers with different generation of CPUs), CPU-level (HMPs) and core-level(per-core DVFS supported CPUs) heterogeneity [2]. This research is relevant to Cloud, Fog and Edge clusters hosting live-services for interactive use cases. It is targeted at micro-service based

[2]Note that my research does not delve into Server-level heterogeneity introduced by accelerators (GPUs, FPGAs, TPUs, etc.).

implementation of LC-Services that service queued user requests. This research is not relevant to serverless implementations performing ephemeral task executions for every request. The proposed cluster-level strategies are not applicable to data-parallel OLDI workloads or Map-Reduce frameworks such as Hadoop.

1.4 Contributions

Chapters 2 and 3 consists of two contributions to cluster-level strategies to exploit heterogeneity.

In Chapter 2, I present Greeniac, a cluster-level task manager for LC-Services hosted on a HMP-server cluster. Greeniac exploits both Core-frequency heterogeneity (DVFS capability) and CPU heterogeneity, albeit from cluster-level. Greeniac is a service agnostic task scheduler that automatically learns the best run-time distribution of LC-Service requests among HMP servers, and the cores in their HMPs, to maximize energy savings. Using Reinforcement learning [45], Greeniac learns the optimal set of cores (and servers) on which to host instances, at various load levels. Accordingly, it implements its control plane strategies (scaling and load balancing) dynamically in response to load variations. Greeniac employs several heuristics, to reduce state-space exploration time and speed up learning. My experiments show that Greeniac saves up to 28% of core energy over server-level scheduling approaches on a 4-server HMP cluster.

In Chapter 3, I address the two-dimensional Cluster Heterogeneity challenge. I show that heterogeneity-unaware Load-balancing leads to SLO-violation at lower server utilization. To address Capacity Heterogeneity, I propose a novel Scaling and Load balancing strategy that employs heuristics based on *Maximum-SLO-Guaranteed Capacity (MSG-Capacity)*. To address both capacity and efficiency heterogeneity, I propose an Energy-efficiency aware and MSG-Throughput (E-MT) heuristic-based control-plane strategy. My results show significant utilization and energy-saving benefits for LC-Services over traditional heterogeneity-unaware execution.

In Chapter 4, I dive down to the server-level and propose techniques to harness the two primary system-level latency contributors: Interrupts and OS queuing. In this chapter, I further refine

the core-objective into a 3-pronged-problem: 1) meeting tail latency targets, 2) reducing tail latency variability, and 3) improving energy efficiency of LC-Service hosts. To mitigate Interrupts, I analyze several network configurations and exploit Core-frequency Heterogeneity. I show that a dedicated frequency-scaled core for Interrupt processing can alleviate all three problems associated with interrupts at high loads for LC-Services. I identified the receive socket buffer queue as the primary source of queuing delay in kernel. Again, by exploiting Core-frequency Heterogeneity, I propose a dynamic queue length based instance scaling and frequency scaling of service instances. These server-level control plane strategies are implemented as runtime system that performs dynamic resource allocation and configuration to adapt to request traffic rates and meets the refined objectives. The runtime reduces interrupt-driven tail latency variability by up to 86% and achieves up to 16% energy saving for a variety of LC-Services.

In Chapter 5, I propose a novel user-centric execution paradigm. I propose ESLO, a novel user-centric SLO, to meet a desired User QoE target. We propose and implement an ESLO-enforcing framework in Kubernetes on ChameleonCloud Testbed [47, 48]. The framework consists of three control plane components to manage the tail QoE: (a) an ESLO-aware scheduler that employs deadline-based scheduling strategies, (b) an ESLO-aware scaler that estimates and dynamically scales the number of LC-Service instances, and (c) an ESLO aware load balancer routine that collects request traffic information at cluster-level. The load balancer assists the scheduler and the scaler components in implementing their strategies. With the proposed control-plane strategies I was able to increase server utilization by upto 15%. I also demonstrate a case study on enforcing ESLO for an Edge-cloud deployment with ultra-low latency requirements.

Chapter 6 follows up on Chapter 5 to address the challenges introduced by cluster heterogeneity in a user-centric paradigm. The objective here is to enhance the ESLO-aware Load-balancer to reduce energy overhead of services on a heterogeneous cluster. I demonstrate how external network delay variability can be leveraged at the control plane to exploit cluster-level heterogeneity. I propose a novel delay-spectrum based load partitioning between slow(efficient) and fast(inefficient) servers. I show that on a small scale deployment (mentioned in Chapter 5), we can achieve between 5-62% power saving.

1.5 Organization of the Thesis

This Thesis is organized as follows. Chapter 1 (this chapter) presents the background and motivation for my research works and lists the main objectives. Chapters 2 and 3 present cluster-level approaches to exploit heterogeneity through scaling and load balancing strategies. In Chapter 2 I use Reinforcement Learning to infer the optimal control-plane strategy for a cluster of HMP servers exhibiting CPU Heterogeneity and Core-frequency Heterogeneity. Then in Chapter 3, I develop heuristic-based strategies for a cluster of SMPs exhibiting 2-dimensional Cluster heterogeneity. Chapter 4 explores further optimization opportunities at server-level by exploiting Core-frequency heterogeneity through task scheduling and a custom runtime. Chapter 4 presents server-level strategies and OS-level runtime system to mitigate system-level latency-contributing factors by exploiting Core-frequency heterogeneity. Chapter 5 identifies that current control plane strategies do not guarantee User QoE levels. A new user-centric execution paradigm is proposed and a framework comprising of redesigned control plane components, is implemented on real Cloud testbed. Chapter 6 proposes a way of exploiting Cluster Heterogeneity in the novel User-centric framework by enhancing the Load-balancer component. An external network delay based load distribution across servers is shown to reduce energy overhead in the Heterogeneous Cluster. Finally, Chapter 7 mentions the conclusions of my research and discusses the future research directions for extending my work.

Chapter 2

Cluster-level Control Plane Strategies to Exploit CPU Heterogeneity and Core-frequency Heterogeneity using Reinforcement Learning

2.1 Brief Overview

The current generation of cloud cluster platforms waste significant amount of energy to guarantee latency requirements of Latency-critical online services. Low energy-efficiency of homogeneous multicore server processors is a major contributor to this energy wastage. The use of heterogeneous multi-core processors (HMP) [28, 49] has been advocated in recent studies [33, 34, 35] to achieve higher energy efficiency for latency-sensitive services by adapting to dynamic load changes. HMPs can process requests during heavy loads using the fast but power-hungry big cores, and switch these big cores to low-power sleep states whenever slower but energy-efficient small cores can meet the latency objectives for the request load. But these proposed CPU Heterogeneity aware intra-server task scheduling approaches can optimize energy-efficiency only at the server-level. The opportunity

for a cluster-wide task distribution to optimize energy usage in a HMP-server cluster is not explored in prior works.

In this chapter, I present Greeniac, a control-plane task manager for LC-Services on HMP-servers that leverages CPU heterogeneity at the cluster-level. By exploiting both Core-frequency Heterogeneity and CPU Heterogeneity, it employs frequency scaling and instance scaling, to maximize the energy savings from the HMP cores. Greeniac is a service agnostic task scheduler that automatically learns the best control plane strategies for HMP servers and cores to maximize energy savings. The control plane strategies involved are a scaling strategy for run-time distribution of LC-Service active instances and a load balancing strategy for run-time distribution of requests among the active instances. It first uses a *contextual bandit* reinforcement learning technique [45] to infer the most efficient instance-to-core assignments on a server at various load levels. Then, it applies a dynamic programming solution to multi-choice Knapsack Problem [50] along with a gradient descent heuristic to determine the globally most energy-efficient load distribution of requests across HMP servers. My experiments show that Greeniac saves up to 28% of CPU energy over traditional server-level heterogeneity-aware scheduling approaches on a 4-server HMP cluster. To the best of my knowledge, no such solution has been proposed before.

2.2 Background

2.2.1 Hosting LC-Services on HMPs

Recent studies [33, 34, 35] have found HMPs to be especially suitable for efficient execution of LC-Services. Unlike power-hungry multi-cores, HMP-based servers can offer improved energy efficiency by adapting to load changes. Big cores within HMPs possess several specialized speedup units and typically run at higher frequency, thereby consuming significantly higher energy compared to the smaller cores which typically use slower in-order processing and lower frequency. During high loads, fast big cores can process the load while meeting the latency objectives for the request load. At lower loads, those power-hungry big cores can be switched to low-power sleep states and energy-efficient smaller cores can process the load to save energy. Due to lower load the slower smaller cores

could also meet the latency objectives. Prior works have translated the tail-energy co-optimization problem into the following scheduling problem: For a given request load at a single HMP-server 1) how many LC-Service instances need to be run? And 2) on which cores (big and/or small) do they need to be run? By monitoring the request load at the server they determine the number of big and small core instances required to meet the SLO. The cores are accordingly activated/deactivated dynamically and the subsequent incoming requests distributed among active instances. Thus the CPU-heterogeneity of HMPs can be exploited to dynamically adapt to the unpredictable resource requirements of LC-Services. These observations and opportunities have opened up debate on the suitability of HMPs for cloud platforms.

2.2.2 A Case for Using HMPs in Edge, Fog and Cloud

Heterogeneous multiprocessor SoCs (e.g., Qualcomm Snapdragon, Samsung Exynos) have been popular with mobile device manufactures for energy-efficient processing. To achieve high efficiency in clusters, HMP-based servers (e.g. Intel QuickIA [28], Odroid-MC1 [36]) could be similarly employed. But despite the potential benefits, HMPs have not been welcomed into the cloud fleet by any major CSPs so far. I believe there are several reasons that have contributed to this decision. First, CSPs deploy high performance cores in the cloud warehouse that can perform fast processing under high loads, to provide better service to its clients and users. Energy saving is a secondary objective. Second, cloud servers execute a plethora other applications and workload, besides the LC-Services. The faster these tasks are finished, the sooner the CPUs can switch to low power idle states. Unlike in LC-Services, the instances of these tasks need not remain in an active state to serve any future requests. Thus, the benefits of HMPs may not apply to other cloud workloads. And third, newer generation processors are becoming increasingly energy efficient and energy-proportional, thereby leaving smaller margins for improvement for HMPs.

However, HMPs are ubiquitous in the mobile industry and the life-force behind mobile computing devices. The latest mobile devices use up to 3 core types. Since energy efficiency is vital for battery longevity in portable devices, CPU heterogeneity has found intense research focus in the end-user devices. The onus of computing in recent years is being gradually offloaded from massive centralized

cloud warehouses (Cloud Computing) to smaller decentralized clusters or cloudlets (Fog Computing) and to end-user devices (Edge Computing). Thus wide use of HMPs in Edge computing encourages its adoption in Fog domain too. Unlike massive CSP-backed cloud warehouses, for private fog deployments to be economically sustainable, clusters must have low infrastructural, capital and operational costs. Hence the small form factor, affordable equipment cost and low energy footprint of the modern HMP-based compute platforms are motivating factors for employing HMPs on these micro-cluster platforms. My work further demonstrates the energy-saving opportunity of using HMP-cores in small scale deployments as in fog clusters. The benefits presented on Fog clusters could potentially pave the way for inclusion of HMPs in future sustainable green (i.e. energy-efficient) cloud clusters.

2.2.3 Scope and Challenges for LC-Services on Fog Clusters

Though I see convincing reasons for the adoption of HMPs in fog clusters, the opportunities and implications of hosting LC-Services in the fog domain needs to be analyzed. Due to resource-constrained edge-IoT devices, LC-applications are typically installed on remote cloud platforms [51] as LC-Services. But unpredictable and long network delays pose significant challenges for cloud-based LC-Services. Recent studies [52, 53] show that the fog computing paradigm can address these LC-Service requirements by leveraging its network vicinity to edge-devices.

But fog computing paradigm offers several advantages and challenges for LC-Services. *First*, the reduced network delays leaves larger response time headroom and offers opportunity for more energy-efficient task scheduling at fog servers. This also widens the spectrum of LC-Services to those requiring shorter response latency or that have higher computation overhead. Hence latency objectives for services need to be re-calibrated. *Second*, fog clusters typically service IoT clients in a specific domain. This reduces resource interference effects observed in public clouds [11] both at the network and the server levels. However, instead of co-execution strategies [11] used to improve cloud energy-efficiency, *solo-execution strategies* have to be relied upon for fog servers. *Third*, since fog deployments are on a smaller scale, they can be domain-specific and use efficient, application-specific compute platforms. But the small scale and low budget of the clusters limits the

scope for any service-specific customization [17]. Hence *application-agnostic resource management* techniques are desired for efficient service execution on fog clusters. *Finally*, due to the small client base serviced by fogs, request traffic has relatively higher variability compared to public clouds. Hence the scheduling approach must *dynamically adapt* to meet the resource requirements of the *varying request load*.

2.2.4 Scheduling Services and Distributing Requests on Clusters

Multiple instances of LC-Services are typically hosted on clusters as long running tasks. Service instance scaling in a cluster is dynamically performed by cluster-wide resource managers (e.g. Amazon ECS [23], Borg [38], Omega [54]) based on preset policies. These instances execute on containers that are allocated one or more dedicated CPU cores. On a server with a multi-core CPU, multiple such instances may be hosted. The instance scaler performs resource estimation based on runtime metrics, primarily the cluster-wide request load, and determines the number of instances desired accordingly. It then coordinates with resource managers to identify servers with idle cores that meet the resource requirements of a service instance. Then the instance creation task is assigned by the scaler to a cloud-managed task scheduler. Upon instance creation by the instance scaler, the service instance is ready to process incoming requests. Thus a service scheduler combines the task of an Instance Scaler and a task scheduler.

A single user request might require processing by multiple service types defined by a Directed Acyclic Graph or DAG [55]. For each LC-Service, request are first queued at a load-balancer that forwards them to one of the service instances. Load balancing approach may be CSP-defined, and can be centralized [56] for small clusters or distributed [57] for public clouds. Within a server, requests forwarded to an instance are queued and serviced on the respective cores, typically in a FIFO order. Frequency scaling of the cores can be performed at the server level through local resource management.

Though good scaling strategies can can reduce the energy footprint of LC-Services, they need to be heterogeneity-aware in order to maximize the potential benefits of using HMPs. However,

current control plane components assume compute resources to be homogeneous. Popular cloud resource management systems, such as YARN [58], Mesos[59] and Kubernetes[32, 31], are excellent distributed resource managers, but they all treat the cores in a HMP equally instead of setting processing capabilities individually.

2.3 Motivation

As explained above, the proposed approaches to exploit CPU-heterogeneity [33, 34, 35] only perform task scheduling on a small set of big and small cores within a single server. However, the requests received at the cluster-level load balancer are expected to be equi-distributed among the servers. Thus, the previously proposed server-level local optimizations are a simple but sub-optimal approach compared to a global cluster-level approach. The energy savings could potentially be greatly improved by viewing the entire cluster as a single server and scheduling over a large set of heterogeneous cores spread across multiple HMP servers.

I chose Lucene [60] as my LC-Service test application. Apache Lucene is an exceptionally popular and most widely used high-performance information retrieval library that is used in many applications and websites to provide search functionality. A request typically consists of a set of queries consisting of keys. The search engine extracts the corresponding values from a huge in-memory database of key-value pairs and returns back the response. I adopted the test configurations of Lucenebench [61] as used by authors of [35]. The Lucene search engine was configured with 33+ million Wikipedia English Web pages. The workload characteristics as observed on a Xeon and Atom processor based servers were measured.

2.3.1 Server-level Opportunities

Exploiting heterogeneity within a HMP server requires timely scheduling of requests on big and small cores. The control plane components can employ the following two techniques to maximize energy saving as described below (and depicted in Figure 2.1).

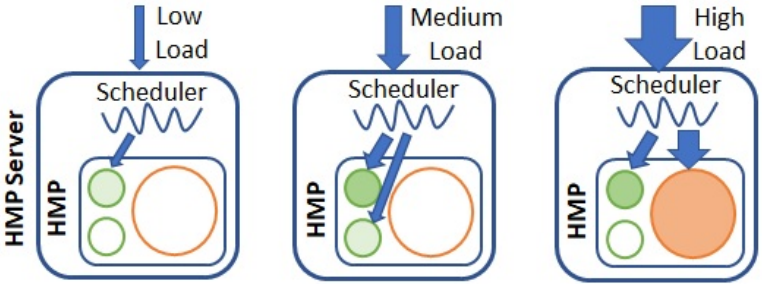


Figure 2.1: Three different representative HMP Service Configurations (SCs) optimal for three different load levels. An SC is a set of active cores in the server and their corresponding frequencies. The unshaded cores represent inactive cores on which no LC-Service thread is scheduled. Shaded cores are active cores hosting an LC-Service thread. Cores with darker shading are frequency-scaled to run at higher core frequency. At low load (left), big cores become inactive and enter sleep mode. Instance scaling with preference for small cores helps to meet SLO target optimally. At intermediate loads (center), Instance Scaling and Frequency scaling are simultaneously exploited to avoid using big cores. At high loads (right), Instance Scaler schedule LC-Service threads on big cores too. As the load further increases, Frequency Scaling is greedily employed to increase the service capacity of big cores.

Instance Scaling to Exploit CPU Heterogeneity To maximize server-level energy savings, all instance scaling options must be explored on an HMP server. For example, on an HMP with 2 big and 2 small cores, assume 2 service instances are hosted on 2 small cores. Upon a load increase, multiple scaling options exist after initiating a big core service instance - (a) terminate a small core instance, (b) terminate both small core instances, or (c) keep running both small core instances. The option that maximizes energy saving while still meeting the SLO target for loads would be the optimal choice for a good resource manager. However, identifying the optimal choice will be dependent on workload type, scheduling framework and HMP architecture. Additionally, when the big cores are idle and not hosting instances, HMPs can save energy by switching them to sleep mode.[1] The deepest sleep states are reached by powering down shared resources like LLCs and hence cannot be achieved for HMPs with even a single active core. When an LC-Service instance is scheduled on a core, it wakes up. The CPU wake up latencies, even for the the deepest sleep states, are in the sub-millisecond [62] range, typically less than 100 microseconds. This is negligible in comparison to the multi-second order granularity at which scaling decisions are typically made in the cloud frameworks. Thus switching a big core to sleep state is an effective energy-saving technique. The scheduling to handle request processing and meeting SLO targets that are in the

[1]Cores have multiple sleep state levels they can enter immediately upon becoming idle. The longer the core stays idle, the deeper it goes into its sleep levels, and the deeper the sleep state, higher the saving.

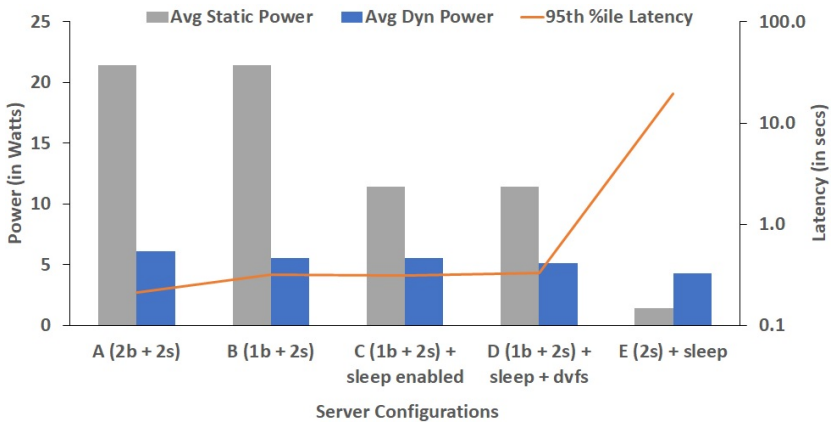


Figure 2.2: Energy saving opportunity on a 2B-2S (2 Big and 2 Small) HMP-server @12.5rps. Big core consume high static/idle power but meet tail objective. Small cores consume less idle power but might violate tail latency. Though frequency scaling does not seem to save much dynamic power in this experiment, prior studies have shown its benefits for LC-Services.

order of milliseconds.

Frequency Scaling to Exploit Core-frequency Heterogeneity In Dynamic frequency scaling (DFS or DVFS), the core clock frequency can be switched between a set of discrete frequency levels. A higher frequency level enables faster processing but consumes more power. Lower frequency levels achieve lower throughput and a lower power draw. When executing an LC-Service instance, dynamically scaling the core frequency to match the request load has been shown to achieve energy savings [25, 26]. Since frequency scaling only affects dynamic power consumption of cores, savings are relatively low compared to the Instance scaling techniques. However, since the frequency switching latency is in sub-microsecond range [63], it is much faster than the Instance scaling techniques and can be applied for fine granular power management within an HMP server. In Rubik [25], authors perform such a fine-grain power management on SMP processors to gain significant energy savings for LC-Services with service times of the order of microseconds.

Figure 2.2 shows a comparative study of tail latency and energy utilization for 5 different configurations (A through E) on a single hypothetical HMP consisting of 2 Xeon cores and 2 Atom cores. The power results were obtained by aggregating results from an Atom and a Xeon CPU server, each employing two up to 2 cores. Big (Xeon) cores are quite inefficient compared to small (Atom) cores, but are crucial to service high request loads under latency budgets. Starting from a

configuration with 4 service instance (A), it was observed that scaling down from 2 to 1 Xeon core instance (configuration B) only barely reduced the dynamic power consumption while maintaining the static power draw. The tail latency too saw a minor increase due to reduced compute resource. Enabling sleep states (in configuration C) helped significantly reduce the power consumption of the HMP. Frequency scaling on under-utilized big cores (in configuration D) is shown to save little dynamic CPU energy. However, this observation is specific to the test setup. DVFS technology has shown to save significant dynamic energy in numerous research works [64, 65, 66, 67] over past decades. Besides, lower frequency does lead to lower CPU temperatures, thereby saving cooling system energy too[2]. A significant reduction in power is observed in configuration E when both Xeon cores are disabled. However, this results in an exponential increase in tail latency since the service capacity of the two Atom cores is insufficient to handle the request rate. Hence, in a dynamic setup, instance scaling must be performed in a SLO-aware manner while trying to minimize power consumption. Due to the benefits of CPU heterogeneity and Core-frequency heterogeneity observed in Figure 2.2, I exploit both these forms of heterogeneity within a server-local scheduler to maximize HMP-server energy saving.

2.3.2 Cluster-level Opportunities

While server-level resource management can achieve energy savings on an HMP server, in a cluster with multiple servers a server-local task/resource manager can only achieve a local energy optimum. Through my studies (as shown in Figure 2.3) I show that, energy savings could be significantly increased with a cluster-level resource manager that is aware of all the compute-resources available in the cluster. To maximize energy savings, two control plane tasks must be tuned in tandem: service scheduling and load balancing.

Service Scheduling At the cluster-level, the number of possible combinations of big and small cores is large. A cluster task scheduler that can identify the most efficient core combinations for the current cluster load and elastically initiate or terminate the instances on those cores/servers

[2]In my thesis, I do not account for energy consumption contributed by non-CPU server components, cooling system and networking infrastructure. My results only depict the CPU energy saving achieved by my proposed techniques.

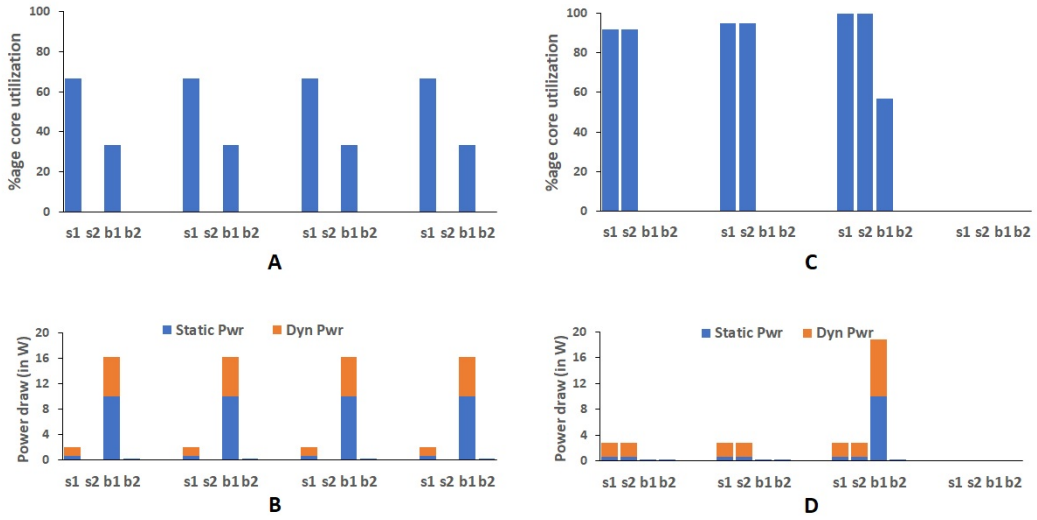


Figure 2.3: Cluster-level energy saving opportunity on a 2B-2S HMP-server cluster with 4 servers at 50 rps cluster load. With an HMP-unaware load balancing, each server needs 1 big core (b1) and 1 small core (s1) to serve the 12.5 rps server load. The 4 big and 4 small cores used in the cluster are all under utilized (shown in A) while the overall power consumption of the cluster is high (shown in B) due to the 4 big cores used. In contrast, an energy-efficient HMP-aware cluster-level load balancing and server level scheduling (shown in C) needs 6 small cores (3s1 and 3s2 cores) and only 1 big core to server the same cluster load. This significantly reduces the overall power consumption of the cluster (shown in D). Cluster-level optimization can results in use of fewer big cores, achieves high core utilization and lower aggregate power consumption.

can maximize the energy efficiency of the cluster. To demonstrate this, a load of 50 requests per second (rps) for Lucene workload was assigned to a 4-server cluster with 2B-2S HMP configurations. Assuming HMP-aware local task-scheduling (as in Hipster [33]), requests were equally distributed using a least-loaded strategy [54]. Figure 2.3(A) shows 2 service instances (1 big and 1 small core) are required to meet the SLO at each server. Note that core utilization is low (Figure 2.3(A)) and there is significant energy wastage (Figure 2.3(B)). With cluster-level HMP-awareness, a cluster-level efficient configuration consisting of more small cores and fewer big cores (Figure 2.3(C)) can be used to host services, meet the SLO and use lower aggregate cluster energy (as shown in Figure 2.3(D)).

Load Balancing Once the most efficient core combination is determined, it is important to optimally divide the aggregate cluster load across the instances to guarantee the latency requirements. Load balancing of requests across HMP servers ought to be proportional to each server aggregate

service capacity. Additionally, if each server meets the tail latency SLO for its hosted service instances, the aggregate latency objective for the entire cluster is implicitly met. In this work, I dynamically identify the optimal tuning options for both techniques within a cluster-wide scheduler to maximize cluster energy savings.

2.3.3 Challenges in Implementing a Cluster-level Strategy

Though the scope for energy saving in an HMP-server cluster is evident from Sections 2.3.1 and 2.3.2, designing a practical approach to maximizing cluster energy efficiency is extremely challenging for the following reasons.

Resource Heterogeneity: Most previous cluster-level approaches proposing energy-optimal scheduling [68, 69] assume homogeneous clusters with homogeneous multi-cores [70, 71] servers and identical service rates for all service instances. These techniques cannot be applied to a cluster of HMP servers due to the heterogeneity in service rates. Moreover, when servicing latency-critical requests with strict latency objectives, dynamic core and frequency scaling must be employed. Static and theoretical scheduling approaches based on architectural and workload assumptions are not practical in this case.

Large Configuration Space: Due to the large number of energy saving scaling options discussed in Sections 2.3.1 and 2.3.2, the configuration space (i.e., the number of available core and frequency combinations) grows exponentially with cluster size. For instance, assume a 4-server cluster with 1 big and 1 small cores per server. Assume the big and the small cores have 5 and 3 frequency levels available, respectively. Each HMP server then has 24 configuration choices (5 with big core alone, 3 with small core alone, 15 as a combination of big and small cores, 1 with both cores inactive). Across 4 servers, the number of possible configurations exponentially increases to 24^4 options. Since most clusters require multiple active servers to meet LC-Service load requirements, it is infeasible for a centralized resource manager to identify a cluster-wide optimal configuration using only a brute force search.

Dynamic Resource Availability: Both cluster resources and LC-request loads are highly dynamic. Due to the co-execution of a variety of jobs within clusters, HMP servers at times might not have all the cores available for scaling. For instance, on a 2-big, 2-small HMP, if 1 big core is pre-provisioned for another service or a batch job, then the job scheduler can only use the remaining 1 big and 2 small cores to host its service. A scheduler must also accommodate for such variations in server resource availability while optimizing cluster energy.

Service Characteristics: Different services have different service characteristics and performance requirements. Compute-intensive services usually achieve higher speedup on big cores over small cores, thereby reducing the energy efficiency gap between the core types. On the other hand, memory-intensive tasks gain little by executing on big cores and thus are more efficient on small cores. Therefore, a run-time energy optimization approach must be tailored to be service-specific.

Traffic Characteristics: Cluster request loads exhibit diurnal patterns [14] with a large load range. Classifying a large continuous load range using only a few coarse-grained load levels to determine a scheduling strategy is quite wasteful. Instead, a mapping of fine-grained load levels to the most efficient configurations that can handle the load is preferable.

2.4 Greeniac: A Smart Task Manager

Greeniac is a two-level control-plane task management system that orchestrates the instance scaling and request load balancing. It consists of three primary components: a *learning agent*, a *service scheduler* and a *load balancer*. The learning agent employs a two-level Reinforcement Learning (RL) approach and is comprised of a Server-level Learning Agent (SL-Agent) that runs on the individual HMP-servers and a Cluster-level Learning Agent (CL-Agent) which runs on the Orchestrator Node. The overall architecture is shown in Figure 2.4. For different request loads, the SL-Agent learns the service configuration (SC) that best minimizes the energy usage while meeting tail latency requirement for LC-Services. An SC is a list of active cores in the server and their corresponding frequencies.

This information is then used by the CL-Agent to learn the best cluster-wide service configuration (CSC) for different aggregate request loads observed by the cluster. The CSC is the collection of optimal SCs for all servers in the cluster. The service scheduler dynamically employs service management at the server-level to implement the configuration determined by the learning agent. Together, the learning agent and the service scheduler implement heterogeneity-aware instance scaling. The load balancer is responsible for distributing the requests among the active servers. The three components execute in a coordinated fashion to maximize the energy savings of clusters under a given SLO target. Details of these components are described in the following subsections.

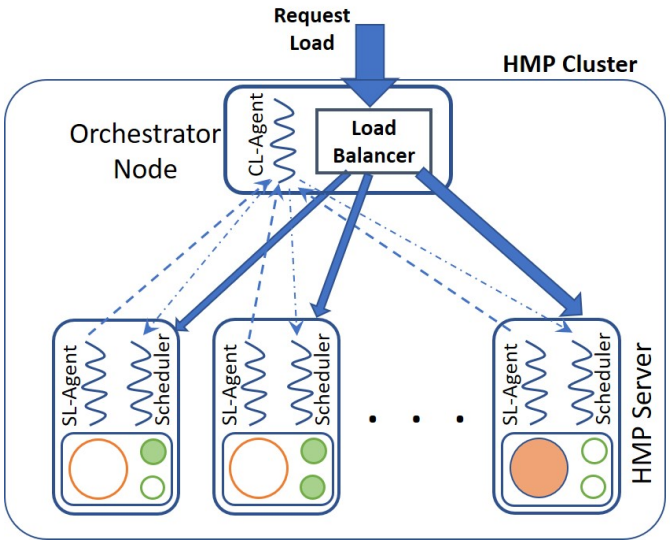


Figure 2.4: Greeniac task management on an HMP-cluster. Server-level SL-Agents learn optimal local Service Configurations (SC) and update the CL-Agent, which then learns optimal Cluster Service Configuration (CSC). CL-Agent drives local Service Schedulers and central Load Balancer to activate the optimal CSC and distributes load proportionally every epoch.

2.4.1 A Reinforcement Learning Problem

In reinforcement learning the system is modeled as a Markov Decision Process (MDP) consisting of a set of states ($s \in S$) and actions ($a \in A$). A state is a snapshot of all characteristic variables that define the system. An action enables transition between system states and there is a reward for each. I formulate the energy optimization problem as a *Multi-Armed Bandit (MAB) problem* [45]. A MAB consists of a single state (bandit) and multiple actions (arms) wherein an agent learns which action returns the maximum average reward. I define a state to be the average arrival rate

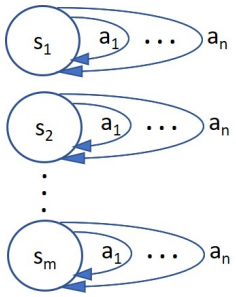


Figure 2.5: Multiple Multi-armed Bandit Problems to learn optimal action for every state. Server load represents state, SCs represent actions. The SC with maximum reward for a load is selected as optimal by the Reinforcement Learning (RL) logic.

of LC-Service requests. The actions are the choice of service instance assignments. The agent waits for a pre-defined epoch to observe the effect of an action on the tail latency and energy usage, and based on this observation it calculates the reward. The agent learns the best action (configuration) for the current state (load) from exploration of various action choices in run-time. The learning process is performed for the entire range of request loads to determine the best action (configuration) for every system state (request load). Thus, effectively, the learning problem can be conceived as a *Contextual Bandit problem* (Figure 2.5) where each MAB has a separate solution based on the context which in this case is the state or the request load.

Due to the large number of configuration choices, it is not feasible to completely explore the entire action space, particularly for larger cluster sizes. Furthermore, since learning agents must wait for multiple requests to effectively calculate the tail latency, average reward calculation even for a single state-action pair is quite time-consuming. Thus a naive RL exploration to learn the optimal task distribution would be prohibitively expensive. Finally, if all possible load values are considered then the state space is continuous and cannot be explored thoroughly. Therefore, I employed an empirically-verified heuristic to prune the action space and discretized the state space by mapping the request load values to a finite number load ranges. I describe the two-phase learning approach used at the server-level and the cluster-level in the following subsections.

2.4.2 Server-level Learning Agent

The server-level learning agent (SL-agent), hosted on HMP servers, first determines an efficiency-signature (Section 2.4.2.1) for all server configurations. Then it shortlists the efficient service configurations (e-SC) (Section 2.4.2.2) to be explored during learning phase. A throughput-sorted EC-list is used as a heuristic for the MAB RL learning phase (Section 2.4.2.3) to determine the efficient configurations (EC) for various loads based on a reward function (Section 2.4.2.4). After the completion of learning phase, a mapping (Section 2.4.2.5) between the load values and the corresponding optimal e-SC learned is generated and reported to the Cluster-level learning agent (CL-agent) described in Section 2.4.3. Since identical servers are expected to produce almost identical results for same workload, the SL-agent needs to learn the mapping from only one HMP server.

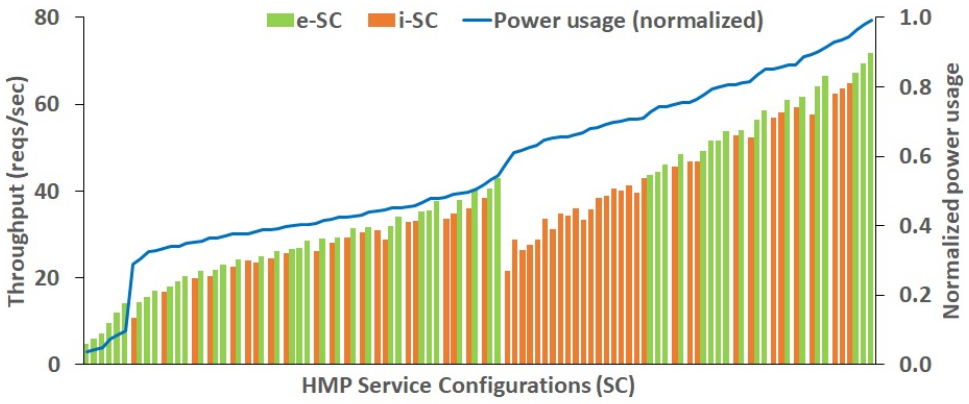
2.4.2.1 Determining Efficiency Signatures

The efficiency signature for a Service Configuration (SC) is defined as the throughput achieved and the power consumed by the SC under maximum utilization. An SC is a list of active cores and corresponding frequencies where each active core hosts a single LC-Service instance. The SL-agent lists all possible SCs for a HMP-server based on the cores' type, count and frequency levels. The local scheduler (Section 2.4.4) performs core-scaling and frequency-scaling to activate service instances for a specific SC. To obtain the signature for a target SC, an HMP server is subjected to a a high rate of request traffic[3].

2.4.2.2 Identifying Efficient Configurations

Not all SCs are efficient. For some SCs, the maximum throughput achieved is lower than at least one other SC with lower power usage. We eliminate out these inefficient SCs (i-SCs) from our RL action space. The remaining set of efficient SCs (e-SC) is sorted by the achieved throughput to create a sorted e-SC list. These are the configurations that maximize energy efficiency of a

[3]The efficiency signature of an SC may be different for different LC-Service application.



Power profile of Big core (Xeon)

Freq Level	Freq (in MHz)	Power (in W)	Freq Level	Freq (in MHz)	Power (in W)
1	3200	25.92	5	1900	19.05
2	2900	23.64	6	1600	17.74
3	2600	22.15	7	1200	16.88
4	2200	20.5	10	Idle	10

Power profile of Small core (Atom)

Freq Level	Freq (in MHz)	Power (in W)
7	1200	2.85
8	1000	2.5
9	800	2.1
10	Idle	0.7

Figure 2.6: Maximum throughput achieved while meeting the SLO by all possible Service Configurations (SC) sorted by maximum power usage for a 2B-2S HMP. The throughput is measured in requests serviced per second. Power measured at that throughput is normalized with respect to the maximum power consumption of the HMP at 100% utilization. The throughput of each SC is shown in bars and the corresponding power consumption by the blue line. Out of the 104 SCs, 54 were efficient SCs or e-SCs (shown in green) and 50 were inefficient SCs or i-SCs (shown in red). An SC is considered i-SC if its power consumption is higher but throughput is lower than at least one other SC. The learning agent only needs to explore the e-SCs to determine the most efficient SC for a load. The power profiles of the frequency-scaled big and small cores are shown.

HMP-server for a load range (L_{i-1}, L_i), where L_{i-1}, L_i are the maximum throughput for $(i-1)^{th}$ and i^{th} e-SC in the sorted e-SC list, respectively. The sorted e-SC list configurations are also sorted by maximum energy usage and is used by learning heuristics to speed up learning. Figure 2.6 shows the throughput-sorted list of configurations for our representative HMP server with 2-big Xeon cores with 7 frequency levels and 2 small atom cores with 3 frequency levels. Out of 104 SCs, only 54 are e-SCs (shown in green), while the rest 50 are i-SCs (shown in red). These e-SCs are considered for learning, while the i-SCs are ignored, thereby considerably shrinking the action space.

2.4.2.3 Heuristic-assisted Server RL Agent

The SL-agent uses the set of e-SCs (a_1, a_2, ... a_N) as its action space with uniform initial action values (set to 0). Learning is performed with an artificial trace (similar to Hipster [33]) with the

load range divided into multiple windows of arbitrary granularity. Each window corresponds to a state. A learning step is taken every fixed epoch. SL-agent predicts the average load for the next epoch and associates it with a window (state). The action selection is heuristic-assisted. Initial action for state s_j (load L_j) is chosen to be a_i, where $l_{i-1} < L_j < l_i$. Subsequent actions selected depend on the 95^{th} percentile latency observed, T_{95_obs}. To prevent violation of tail latency due to request size variations and bursty arrivals, a safe limit fraction $delta_{high}$ of target tail latency T_{95_max} is adopted as a QoS guaranteeing limit for T_{95_obs} i.e. $T_{95\_obs} < delta_{high} * T_{95\_max}$.

If T_{95_obs} exceeds this limit, a heuristic of selecting the action a_{i+1} for the next epoch is used. The heuristic is based on the assumption that an incrementally higher throughput SC could possibly meet the tail latency target with minimum incremental energy overhead. Similarly, a fraction $delta\_low$ of T_{95_max} is chosen to mark a slack limit, i.e. $T_{95\_obs} > delta_{low} * T_{95\_max}$, where $0 < delta_{low} < delta_{high} < 1$,

If T_{x_obs} falls below the slack limit, a heuristic of selecting the action a_{i-1} for the next epoch. At the end of each epoch, the tail latency and energy usage is measured for the s_j, a_i pair, and a reward is calculated (Section 2.4.2.4) and action values are updated (Section 2.4.2.5) for the state-action pair. The learning phase may be performed online on real traffic too, though at the cost of SLO violations during action exploration.

2.4.2.4 Reward Function

The reward is calculated by taking both the energy used and the tail latency achieved. To credit lower energy usage, I define an energy reward inversely proportional to the measured energy usage for the epoch (Equation (2.2)). This is taken into account to credit latency values closer to the tail latency target (Equation (2.1)). A negative reward proportional to the latency is assigned when tail latency is violated. Since meeting tail latency SLO is critical for LC-applications, to prevent tail latency reward being outweighed by a huge energy reward, both rewards are normalized as shown in Equation (2.1) and Equation (2.2). Further a heavy penalty is imposed on the reward if the desired tail latency is violated. To do this, a heuristic-based reward calculation as shown in

Equation (2.3) is adopted.

$$R_{\mathrm{TL}} = T_{95\_\mathrm{obs}}/T_{\mathrm{SLO}} \tag{2.1}$$

$$R_{\mathrm{PWR}} = 2 * (P_{\max} - P_{\mathrm{obs}})/P_{\max} \tag{2.2}$$

$$\mathcal{R} = \begin{cases} R_{\mathrm{TL}} + R_{\mathrm{PWR}} + 1 & \text{if } T_{95\_\mathrm{obs}} < T_{\mathrm{SLO}} \\ R_{\mathrm{PWR}} - 10 * (R_{\mathrm{TL}} - 1) & \text{if } T_{95\_\mathrm{obs}} > T_{\mathrm{SLO}} \end{cases} \tag{2.3}$$

2.4.2.5 Populating and Exploiting the Q-table

The Action value (Q-value) for an action is calculated by a sampled average of all rewards received for the action. An averaged reward smooths any transient rewards fluctuations due to workload variations and network/server-level interference. The action values are updated in the action value table (Q-table) for the corresponding state. After sufficient exploration using the heuristics, most relevant actions are learned by the SL-agent and the corresponding action values written to the Q-table. The Q-table is subsequently accessed in the exploitation phase to select the highest value action based on a greedy policy.

2.4.3 Cluster-level Learning Agent for Orchestrator Node

After the completion of the first phase of learning, the SL-agents forward their Q-table and e-SC lists to a cluster-level Agent (CL-Agent) running at the load distributor/orchestrator node. Given an aggregate cluster request rate, the purpose of a CL-agent is to learn (a) which/how many servers to activate, and (b) what SC should each server use. CL-agent learning problem can also be framed as a MAB RL problem where aggregate load represents the state and set of e-SCs selected for each server (Cluster SC (CSC)) represents the action. Despite the shrinking of the SC set to the e-SC set at the server-level, all possible CSCs across multiple servers can result in a huge hard-to-explore action set. Thus I again rely on heuristics to accelerate the learning. I envision optimal CSC selection as a combinatorial optimization problem (Section 2.4.3.1) and use its solution as the initial action during exploration. The statically obtained CSC is a close estimate

of the most optimal cluster configuration. Detection of observed optimal CSC is performed through exploration using a gradient method (Section 2.4.3.3) based on a reward function (Section 2.4.3.2). A cluster-wide Q-table (CQ-table) is updated with action values and eventually referenced during the exploitation phase.

2.4.3.1 Multiple Choice Knapsack Problem

Finding the least energy-consuming cluster configuration (i.e. SCs for each server) can be simplified into a combinatorial optimization problem. Assuming i^{th} HMP-server is a bin with n_i balls representing the n_i e-SCs explored. Assume the throughput and energy usage for each e-SC to be the weight and cost of the balls respectively. Let the m HMP-servers in the cluster represent m possibly non-identical bins. Let aggregate cluster request rate W that is to be distributed among the servers represent a knapsack of weight-capacity W. Then our cluster energy-minimization problem would be analogous to choosing a maximum of one of the n_i balls from each of the m bins, so as to fill a knapsack of capacity W at minimum cost. This is effectively a standard multiple choice Knapsack problem [50]. The solution obtained using a solver is an approximate optimal CSC, which is used as an initial action and is passed down to the individual SL-agents to deploy to their respective SCs. The solution tries to deploy the most efficient e-SCs across all servers and execute them at maximum utilization through appropriate load distribution (Section 2.4.5) so that aggregate energy utilization of the cluster is minimized.

2.4.3.2 CL-agent Reward Function

The CL-agent reward function used is identical to the SL-agent, with tail, power and aggregate reward calculated for each state-action pair as in Section 2.4.2.4.

2.4.3.3 Gradient Method for RL

Upon setting the initial CSC for a cluster state *s*, the CL-agent performs a training on the artificial trace with the load corresponding to the state *s*. At the end of each epoch, the cluster-wide tail latency is measured and the energy usage of all active servers in the past epoch summed. The thresholds used in Section 2.4.2.3 are again used to perform the next action selection. If the aggregate tail latency exceeds the safe limit threshold, it indicates that the current CSC is insufficient to handle the current load and thus requires either additional servers or higher throughput e-SCs to be enabled in the currently active servers. Conversely, a slack limit violation offers room for energy saving.

To speed up the learning and to avoid random exploration from a large action set, I use a gradient descent approach. The n-server cluster can be visualized as n dimensions of an $(n+1)$-dimensional space with the SCs representing units in each dimension, the n co-ordinates representing a specific CSC and the $(n+1)^{th}$ dimension representing the actual action value for each CSC (action). Each state is associated with a unique (n+1)-dimensional surface, for which the global maximum offers the optimal and most energy efficient CSC that meets the tail latency requirements. The knapsack solution sets the initial value to a CSC in the vicinity of the global maximum. From there I employ a gradient ascent approach to reach the global optimum. For instance, a unit movement of any server dimension from e-SC_i to e-SC_{i+1} would increase the aggregate cluster throughput capacity of the CSC by approximately l_{i+1}-l_i (refer Section 2.4.2.3), and is desirable upon a QoS violation in the past epoch. Conversely a change in one of the SCs from e-SC_i to e-SC_{i-1} is desirable when a slack limit is crossed. The gradient of the reward function between two epoch is used to select the next action based on a proportional throughput change.

2.4.3.4 Populating and Exploiting the CQ-table

After a fixed number of exploration steps based on gradient ascent, averaged action values are written to the CQ-table. The steps are repeated for the range of states (the aggregate cluster load range). The final CQ-table generated is used by the CL-agent when employing a greedy selection

of the highest value action for a given cluster load. After each epoch the selected CSC for the predicted load rate of next epoch is processed by the CL-agent and the SC for each HMP-server is forwarded to the corresponding SL-agent, which configures its respective servers and prepares for the next epoch.

2.4.4 Service Scheduler

The SC update received by SL-Agents at each epoch contains information about the number of big and small cores within an HMP to be activated and their corresponding core frequencies. SL-Agent relays this information to the Service Scheduler(SS) module, which is responsible for server-level instance scaling and core frequency scaling. When an updated SC requires an additional core, a new LC-Service instance is created and affinitized to the new core by the SS. Subsequently any new or waiting request is scheduled on the new service instance in an FCFS fashion i.e., the service instance with longest idle period receives next request. In order to remove an existing core from the prior SC, no additional requests are scheduled to the corresponding service instance and any executing request is allowed to be serviced till completion. Subsequently the instance is deleted by the SS when the instance becomes inactive. The SS also performs frequency scaling on the active cores based on the updated SC. Since frequency scaling using the Linux governor takes less than a microseconds to activate, the update is faster than the service time for even a single request.

Note that the service scheduler is a server-level scheduler that schedules the task of LC-Service instance creation on the cores chosen by the SL-Agent. This is unlike the traditional cloud resource managers, which perform scaling at the cluster-level since it assumes all cores to be identical. Here the CL-Agent merely informs the SL-Agents about the desired SC for the server, and the task to implement the configuration on the server is delegated to the service scheduler through the SL-Agent. In addition, the Service Scheduler is not a request task scheduler that schedules the execution of requests at an individual instance. At the individual instances, requests are processed in the traditional FCFS manner.

2.4.5 Load Balancer

A Load Balancer (LB) routine is responsible for distributing requests arriving at the gateway server across the HMP servers. The CL-Agent maintains the average throughput information of each explored cluster-level service configuration (CSC) during the learning phase. Upon identifying the optimal CSC for next epoch based on the last epoch load, it shares the throughput information for each HMP-server's SC with the LB. LB infers the load fraction of each server based on the throughput ratios between them, then fits the ratios in to a normalized scale window of 0 to 1. For every request, the LB generates a random number between 0 and 1, identifies which server it corresponds to from the load fraction window and forwards the request to the identified server.

This throughput proportional load balancing is crucial for energy-optimal load processing at individual HMP-servers. Since each server is configured for load-specific optimal SCs for the next epoch, an improper balancing can cause tail violations in some servers and lower efficiency in others. However, as shown in our studies presented later (Figure 2.10), the random number generation approach balances the load across all HMP-servers suitably to meets the latency requirements at all serviceable load ranges for the cluster.

At the individual servers, the load is distributed among the active instances from a common queue of incoming requests by a task scheduler (as in [34, 33]). This approach is unlike the traditional cluster-level load balancers where the load is balanced across all instances. Here the load is balanced across the active servers, and at each server the load is distributed among the instances by a task scheduler as per the above described policy.

2.5 Experimental Approach

Since public clouds currently offer only homogeneous processor based server instances, I had to rely on simulations to study the effectiveness of Greeniac. Furthermore, none of the available cloudlet simulators (e.g., CloudSim [72] or its extensions) support HMP servers. To support HMP-aware execution and dynamic energy usage estimation for the cluster, I developed a custom in-house

simulator, HMP-ClusterSim (described below) based on SimPy, and instrumented it with the real system measurements.[4]

For my experiments, I also addressed the fog-specific challenges discussed in Section 2.2.3. I performed my experiments on small clusters, representative of fog clusters. I avoided co-execution scenarios by dedicating a single core for each LC-Service instance, and latency objectives were also adjusted to a longer SLO of 1 sec (instead of sub-second).

2.5.1 HMP-ClusterSim Simulator

My simulator mimics a typical high-bandwidth cluster with negligible network delay. Like CloudSim, my simulator is event-driven. A single server instance acts as a gateway (or broker [72]) and remaining servers as single-HMP nodes hosting a LC-Service instance (or Task cloudlets [72]) at each core (1 core per VM/container). I assume sufficient memory and network bandwidth at each server and neglect intra- and inter-instance interference. The gateway server generates requests following a Poisson process at specified mean rate; requests are distributed across the nodes following our load balancing logic (Section 2.4.5). Within each server, a service scheduler distributes requests from a centralized queue using FIFO among the scheduler-activated cores (Section 2.4.4). To model service time variations, task length follows a log-normal distribution ($\mu = 0$, $\sigma = 0.25$) scaled by the average service time. The number of big and small cores within a HMP-server and their individual throughput are assumed to scale linearly with frequency scaling, ignoring memory hierarchy bottlenecks and interference. Responses are sent back to gateway server to record the response latency. P95 for all requests received per sampling epoch (set at 100s) is calculated at the end of every epoch. P95 of 1 sec is set as the default SLO. For RL, each state represents a bin of 10 requests per second (rps). The results discussed in Section 2.6 are based on a 4-server cluster with each server containing a single HMP with 2 big (2B) and 2 small (2S) cores.

[4]Code available at https://github.com/sambitshukla/Greeniac

2.5.2 Incorporating Real System Measurements

I performed my initial study on a high-end Xeon E5-2637 (big-core) powered Dell Power-Edge Server and an Atom C3558 (small-core) powered Super-micro Server. From both platforms, I collected the averaged service time, throughput, idle power and dynamic power values at various core frequency levels for a single service instance of Apache Lucene [60] at 100% utilization. The test configurations for Lucene were as described in Section 2.3. A Lucene client application was run on the local network in an open loop to generate a configurable request load. Since network and OS delay was negligible, the average service time and throughput were determined for the servers from the client side. The power values were measured on the servers using the Running Average Power Limit (RAPL) registers on the CPUs. These data were used in the model of the big and small core in HMP-ClusterSim. Though, I use the representative values for a single setup to demonstrate the effectiveness of my proposed approach, these statistics vary with the type of processors, server configuration and LC-Service application used. So scaling and sensitivity studies were performed for these variations and are presented in the following sections.

2.5.3 Simulator Verification

To verify the accuracy of the HMP-ClusterSim simulator, I performed two sets of tests with different service configurations. Due to a lack of a real testbed for cross-validation, I performed the validation against the expected behavior of a theoretical M/M/c queuing model. The load balancer was configured to perform a random load balancing such that, over a large epoch, the requests are equally distributed among the n servers. Since request generation follows a Poisson process with an arrival rate of λ, a random load distribution results in a Poisson arrival at each server with a rate of $\frac{\lambda}{n}$. Each individual server is modelled as an M/M/c queuing system, with c identical cores of equal service capacities. The service time distribution at each cores is exponential with a service rate of μ. The service rateμ is set to the mean service rate observed from real measurements (discussed above). The cluster arrival rate λ is varied between 0 and a maximum value equal to the aggregate service capacity of the entire cluster. To verify the accuracy of the simulator, I compared the mean response time of requests measured in the simulator against the calculated mean response

time for the corresponding queuing system.

Cluster Utilization ($\rho = \lambda/n\mu$)	Expected Mean Response Time (in msecs)	Simulated Mean Response Time (in msecs)		
		n = 1	n = 2	n = 4
0.1	41.1	41.3	41.4	41.2
0.2	46.2	46.1	46.4	46.4
0.3	52.9	52.7	52.8	52.8
0.4	61.7	61.6	61.6	61.7
0.5	74.0	73.5	74.3	74.1
0.6	92.5	92.1	91.9	91.4
0.7	123.3	123.8	122.3	122.2
0.8	185.0	186.2	184.1	185.7
0.9	370.0	369.1	370.0	367.0

Figure 2.7: This table compares the expected and the simulated mean response times for 3 different cluster sizes. Each server has a single big core and represents an M/M/1 queuing system. The number of servers in a cluster (n) is varied. The arrival rate is accordingly varied to achieve different cluster utilization levels. The load balancer is expected to perform a random and uniform distribution of requests among the servers so that each server also experiences a Poisson arrival with a rate of $\frac{\lambda}{n}$. The expected and simulated results are quite close, thereby verifying the end-to-end functionality of the simulator with regards to load balancing.

First, I verified the functionality of the load balancer at scale as number of servers were increased. Each server employed a single active fast core. In this setup, each server is expected to behave as an M/M/1 queuing system. The number of servers, n, was varied. The cluster utilization, ρ, was calculated as $\frac{\lambda}{n\mu}$. The values obtained for 3 different cluster sizes are shown in Figure 2.7 along with the theoretical mean response latency. All the three configurations are shown to achieve similar mean response latency as the expected mean response latency at all utilization levels, thus verifying the functionality of the load balancer.

Next, I verified the functionality of the server-level scheduler at scale as number of cores were increased. A cluster with a single server hosting multiple identical fast cores was considered. In this setup, the server is expected to behave as an M/M/c queuing system with a single queue and c cores consuming requests from it. The number of cores, c, was varied. The cluster utilization, ρ, was calculated as $\frac{\lambda}{c\mu}$. The theoretical mean response latency of an M/M/c queuing system varies with the value of c (as shown in Figure 2.8). The expected and simulated values were found to be

Cluster Utilization (ρ = λ/cμ)	Expected Mean Response Time (in msecs)		Simulated Mean Response Time (in msecs)	
	c = 2	c = 4	c = 2	c = 4
0.1	37.4	37.2	37.4	37.2
0.2	38.5	37.8	38.7	38.0
0.3	40.7	38.8	40.5	38.6
0.4	44.0	40.5	44.3	40.6
0.5	49.3	43.2	49.7	42.9
0.6	57.8	47.4	58.2	47.6
0.7	72.5	54.8	72.1	55.8
0.8	102.8	69.9	101.4	68.5
0.9	194.7	115.9	192.5	114.7

Figure 2.8: This table compares the expected and the simulated mean response times for 2 different server configurations. The cluster has a single server with c big cores that represents an M/M/c queuing system. The value of c is varied and the arrival rate is accordingly varied to achieve different cluster utilization levels. The cores pull requests from a single server queue with the help of the scheduler. For a given cluster utilization level, the expected mean response time varies with c. The expected and simulated results are quite close, thereby verifying the end-to-end functionality of the simulator with regards to server-level scheduling.

similar for both server configurations. This verifies the functionality of the server-level scheduler.

Together, both the results demonstrate that the simulator accurately mimics the basic functionality of the scheduling and load balancing for Poisson arrival and exponential service time based system. For the experiments performed in this and the following chapter, I simulate more complex configuration with heterogeneous servers and clusters, non-uniform load balancing, and biased scheduling strategies. These variations are hard to verify because they do not represent a standard queuing system. However, verification of the end-to-end functionality for the basic building blocks of the simulator offers significant confidence to the results shown in the following section and in Chapter 3.

2.6 Results and Observations

In this section, I first demonstrate the higher energy saving achieved by Greeniac compared to popular cluster-level scheduling approaches. Then I study the impact of scaling and LC-Service application characteristics on energy saving.

2.6.1 Cluster-level Energy Saving

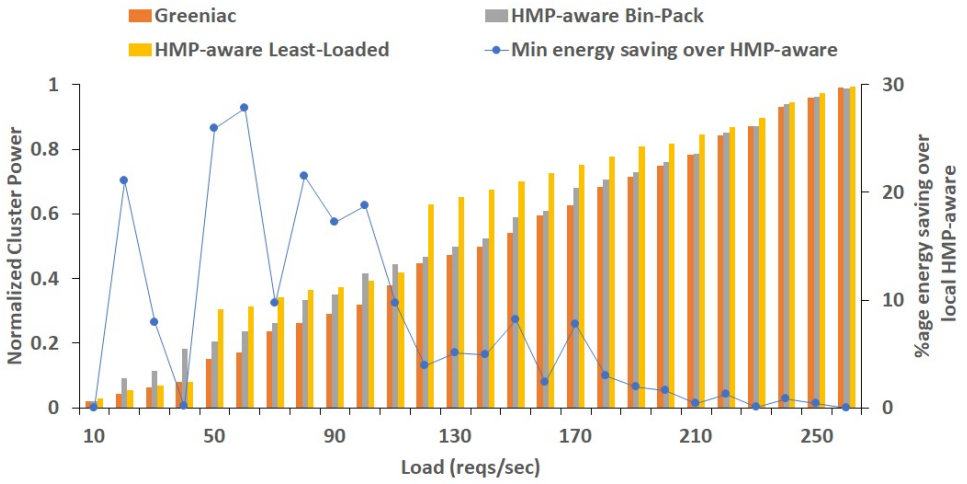


Figure 2.9: Power usage for various Server-level CPU-Heterogeneity-aware scheduling approaches. Power consumption (shown in bars) is normalized (shown in left y-axis) with respect to the aggregate power consumption of the HMP cluster at 100% utilization of all 4 servers. Percentage of power saving achieved over the most efficient alternative approach for a given load is shown as the blue line (right y-axis). Greeniac has higher energy saving (up to 28%) over Server-level CPU-Heterogeneity-aware approaches for all load ranges

To demonstrate the potential of cluster scheduling I compared Greeniac to two popular cluster service scheduling approaches: bin-packing and least-loaded. Bin-packing attempts to minimize the number of active servers by maximizing the utilization of each active server. It schedules instances on all cores of an active server before scheduling services on an inactive server. Requests are proportionately distributed across the active servers by the load balancer in the Bin-packing approach. In contrast, Least-loaded instantiates a service on the least-loaded server and subsequently forwards request to the least loaded instance. Thus it tries to distribute services and request equally across all servers. For a fair comparison, I implemented the Hipster learning [33] at the individual server level for both scheduling approaches to ensure scheduling at each server will be HMP-aware, SLO-guaranteed and energy-optimal for the observed server load. In Hipster [33], the authors determine the most efficient service configuration of a single HMP server for a given load using their own heuristics-based RL technique. Thus both the bin-packing and the least-loaded approaches are unaware of heterogeneity at cluster level, i.e. for scaling decision, but at the server-level the service schedulers are CPU-Heterogeneity-aware, i.e. services and requests are distributed among cores taking into account their CPU-heterogeneity.

Figure 2.9 shows the energy usage for the 3 approaches normalized by the maximum cluster energy usage for HMPs. The line shows the percentage energy saving of Greeniac with respect to the most efficient locally heterogeneity-aware scheduling strategy at the corresponding load. I observed that Greeniac achieves up to 28% higher energy saving over the best case server-level HMP-aware scheduling. Overall, Greeniac gained high HMP power saving for the low and mid range load values that typically represent the majority of load traffic periods, while at very high load, most HMP cores in all servers execute LC-Service instances with high utilization and all cluster scheduling techniques perform almost identically. At low loads, least-loaded distributes tasks equally across all 4 servers with each receiving $1/4^{\text{th}}$ of the cluster load, which can be serviced by small cores within tail latency constraints. As the load is increased, at a certain threshold cluster load level every server must schedule services on big cores to meet its tail target, resulting in poor power savings due to a large number of under-utilized big cores being active across the cluster. On the flip side, bin-packing tries to maximize core utilization for active cores resulting in fewer big cores being active across the cluster. However, since bin-packing tries to maximize utilization of all cores within a HMP before scheduling tasks on an inactive HMP, it schedules services on the big cores of active HMPs while ignoring the availability of low power cores on inactive servers.

The optimal CSCs and the corresponding load distribution for each load level, as identified by Greeniac, are shown in Figure 2.10. The upper plot of Figure 2.10 shows the SCs or the active cores and their corresponding representative frequency levels[5] across all 4 servers. The lower plot shows the fraction of cluster load received by each of the 4 servers and the aggregate cluster power consumption at each load level. At the lowest tested load of 10 rps, two small cores, one each from HMP-1 and HMP-2 are employed at 1.2GHz and the load is equally distributed between them. With increase in load, additional small cores from across the cluster are activated and the load distributed in proportion to the processing capacity of the SCs. At 50 rps, Greeniac activated a big core on HMP-1 at 1.6 GHz, leading to a major fraction of load being forwarded to HMP-1 and a marked increase in power consumption of the cluster. At subsequent higher loads, additional big cores with increased frequencies were deployed on other HMPs too.

Greeniac employs fewer big cores across the entire cluster and tries to maximize utilization of all

[5]see Figure 2.6 for exact CPU frequencies corresponding the representative frequency levels

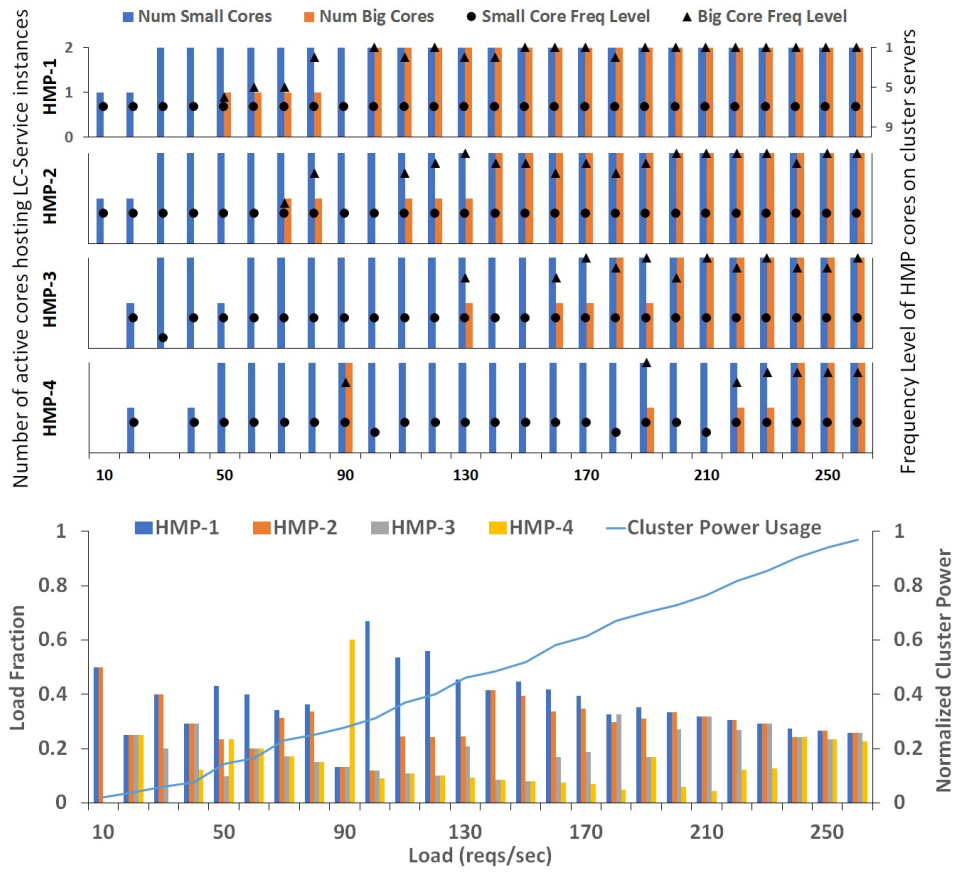


Figure 2.10: The optimal CSCs (top plot) and the corresponding load distribution (bottom plot) for each load level, as identified by Greeniac. For the top plot, each CSC shows the 4 SCs for the 4 HMP servers. Each SC shows the number of small cores and big cores (left y-axis) and their corresponding frequency levels[5] (right y-axis). The bottom plot shows the load distribution across the 4 HMP servers for the corresponding CSC. The fraction of cluster load forwarded to each server is shown by the left y-axis and the aggregate cluster power consumption (normalized to maximum cluster power) is shown by the right y-axis. For any given load, Greeniac employs fewest possible big cores across the entire cluster and maximizes utilization of all active cores while still guaranteeing SLO.

active cores. Using least possible number of big cores also results in lower power consumption at all load levels. Note that, though the CSCs learnt by Greeniac might be optimal for a particular load level, Greeniac's CL-Agent has no locality-awareness about the services. Using the CSCs, as is, in a dynamic control plane can result in bad performance. For instance, at 80 rps, all 8 small cores and 1 big core each from HMP-1 and HMP-2 are employed. But at 90 rps, Greeniac learns employs the 2 big cores from HMP-4. At 100 rps, the optimal CSC learnt consists of 2 big cores hosted on HMP-1. In a real production cluster, with frequent load variations, naively implementing these learnt CSCs can result in wasteful service tear-down and creation efforts. Such performance

issues could be addressed using context and locality awareness before implementing the CSCs at the control plane. But such performance enhancement for a dynamic implementation are left for future work. In this work, I focus on demonstrating the energy benefits of my proposed cluster-level control plane strategy at static load settings.

2.6.2 Scaling Studies

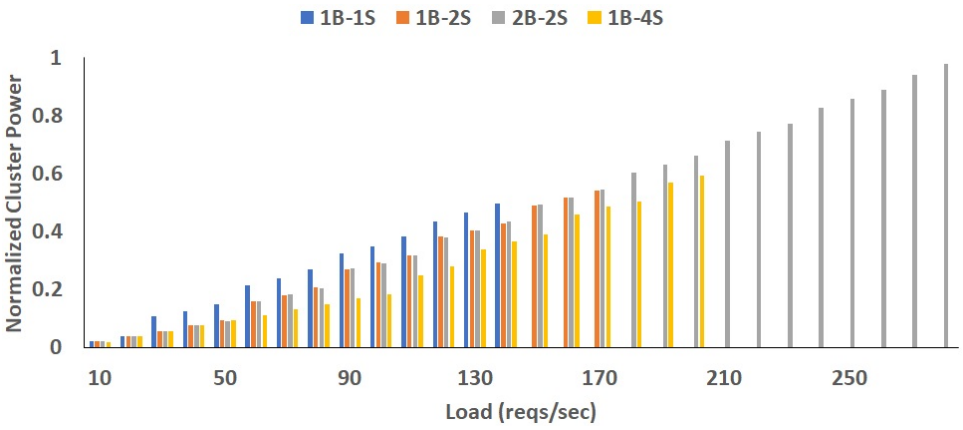


Figure 2.11: Greeniac-enabled power usage for various HMP-configurations in a 4-server cluster. Lower the power consumption greater the energy saving. HMPs with more small cores enable greater cluster power saving at the cluster level. However, HMPs with more big cores achieve higher throughput while meeting latency SLO.

Processor Heterogeneity To test the impact of degree of heterogeneity on the power-saving, I executed Greeniac for 4 different HMP clusters with different combinations of big and small cores in the HMPs. Greeniac achieves higher energy savings with HMPs having more small cores. Hence, in Figure 2.11 1B-4S maximizes saving. Since Greeniac selects the optimal cluster configuration from a global pool of HMP cores, HMPs with the same number of small cores (e.g. 1B-2S and 2B-2S) achieve similar cluster power savings. However, a larger number of big cores with greater service capacity (as in 2B-2S HMPs) can meet tail latency requirements for higher cluster loads.

Cluster Size Larger clusters that can service higher loads also offer opportunities for more energy saving with Greeniac, which can be seen in Figure 2.12. At 100 rps, a 16-server cluster consumes the least power compared to a cluster with fewer servers. This is because at higher load, Greeniac learns to utilize a larger set of the small cores which are available in larger clusters. At low loads,

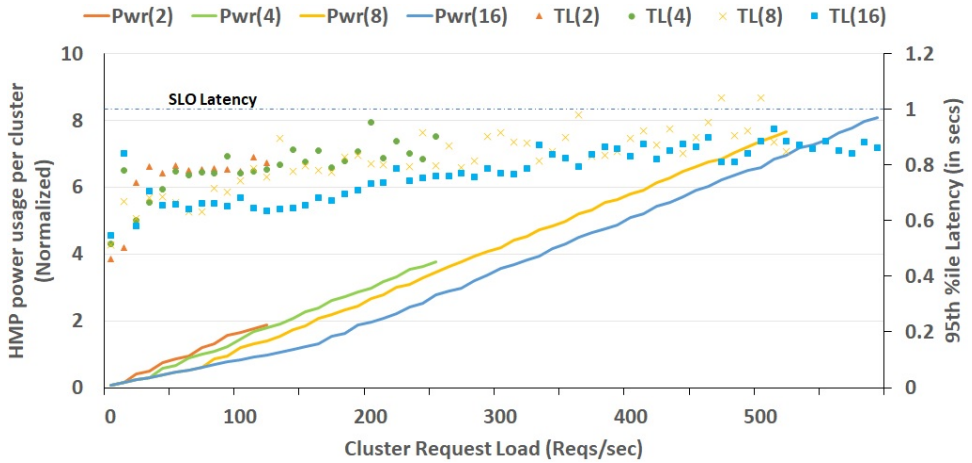


Figure 2.12: Normalized Power usage and Tail latency for various cluster sizes in a 2B-2S HMP cluster. Lines represent the power consumption of each cluster normalized with respect to the maximum power consumption of each HMP. Various types of dots correspond to the 95th %ile tail latency (denoted as TL and shown by the right y-axis) for each cluster at various load values. Smaller clusters can meet the tail latency SLO target of 1 second (black dotted line) up to a smaller load. For the load levels that are supported by multiple clusters, Greeniac consumes lowest power in case of larger clusters. Greeniac can achieve this greater energy saving by exploiting a larger set of small cores and avoiding use of big cores in larger clusters.

different sized clusters consume similar amounts of power since Greeniac learns to activate the small cores on fewer servers. For instance, Greeniac learns that, at 20rps, using 2 small cores each on 2 servers can meet the SLO targets irrespective of cluster size, thereby resulting in the same aggregate power usage. As the load increases, Greeniac schedules services on the big cores of one or more active servers on smaller clusters. Thus at any given load, smaller clusters have a higher number of big cores hosting LC-Services.

2.6.3 Impact of LC-Service Characteristics

Service Time Distribution LC-Services may vary in service time distribution. My default service workload had a low standard deviation (= 0.25). In order to evaluate the sensitivity of Greeniac to various service time distributions, I compare it against a service workload with identical mean service time but higher deviation (= 0.8). Figure 2.13 shows that Greeniac consumes higher power for LC-Services when they have a larger standard deviation. Larger service time variations implies a wider service time range on a small core, and therefore a request scheduled to be serviced

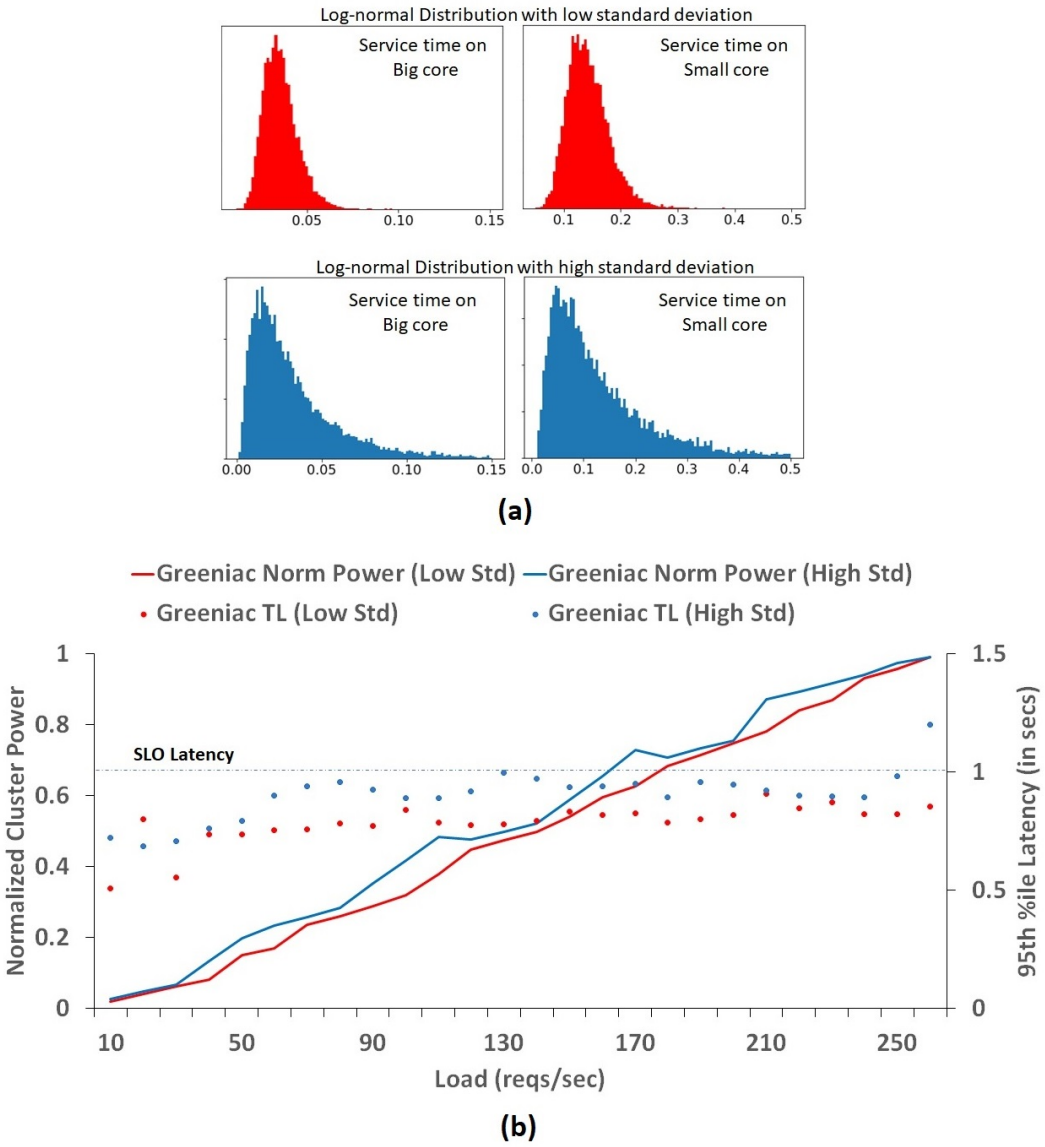


Figure 2.13: Effect of service time distribution on aggregate cluster power. Plot (a) shows the service time distributions on big and small cores for two applications with log-normal distribution and identical mean service time. For various load levels, Plot (b) shows the cluster power consumption normalized to the maximum cluster power as lines (left y-axis) and the corresponding 95 %ile tail latency as dots (right y-axis). Since slower cores achieve higher utilization while still meeting SLO for service distributions with low variation, Greeniac learns to employ less big cores for such cases. Thus the configurations learnt by Greeniac for applications with higher variations in service times (shown in blue) are more power consuming than for applications with lower variations (shown in red).

on a small core is more likely to violate the SLO target. Greeniac learns this service property and finds service configurations with big cores more suitable when there is a higher service time deviation. This results in higher cluster power usage compared to services with lower service time

deviation. Increased variability of service latency is also the reason behind the variation in tail latency and violation of SLO at a lower load as shown in Figure 2.13.

Application Characteristics Compute and memory characteristics of services can lead to different power characteristics and average service time ratios between big and small cores (B:S). Compute-bound applications are known to compute much faster on big cores [73], while the service time for memory bound applications are less affected by processor speed since applications spend a large number of CPU-independent idle cycles on memory waits. The service time ratio between big and small cores and the power characteristics can be affected by other factors as well, such as cache characteristics, memory access pattern, compute units used, etc. But even if two applications have similar power consumption for a given load, both on big and small cores, difference in their service time distributions results in Greeniac learning different optimal cluster configuration for the load. In such cases, average service time ratio (B:S) is the only application-specific factor that affects the choice of optimal cluster configuration.

To test the impact of application characteristics, I consider 3 applications with different B:S ratios at 1.2 GHz core frequency: the Lucene application (A) with B:S as 0.66, a memory intensive service (B) with lower B:S ratio of 0.8 and a compute intensive service (C) with B:S of 0.5. The 3 test applications are configured to have identical service characteristics on the big core. Thus the only difference between them is shown by the variations in service characteristics on the small core as shown in top plots of Figure 2.14. While CPU-intensive application tasks take longer to execute on slow cores, memory-intensive tasks are relatively faster with lower variations. Because of the lower service rate for C, slow cores violate the SLO targets for a lower load compared to A and thus aggregate load serviceable by cluster is lower. On the contrary, slow cores can service higher loads for B, leading to a higher cluster load support as shown in Figure 2.14. Greeniac implicitly learns these application characteristics and identifies the optimal configuration for each application service at various loads. It consumes lower cluster power for B by exploiting small cores, and consumes higher cluster power for C due to relatively greater use of big cores for the same load.

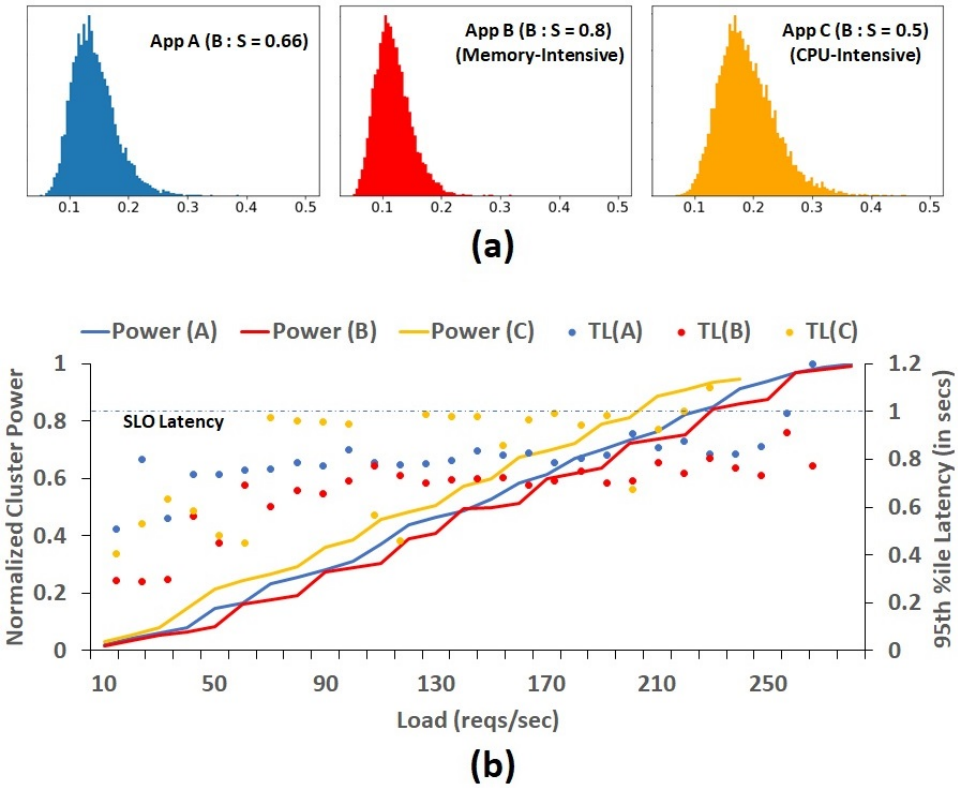


Figure 2.14: Effect of application characteristics on cluster power and tail latency. The 3 applications have identical service characteristics on big core. The variation in service times (in secs) on the small core due to the difference in application characteristics are shown in (a). CPU-intensive services run relatively slower while memory-intensive services run relatively faster on small cores with respect to big cores. The normalized power consumption of the cluster and the tail latency for the three applications at different load levels are shown in (b). For a given load, Greeniac learns to use more big cores for CPU-intensive compared to memory-intensive services. Thus power consumption is lower and SLO is met up to a higher cluster load for memory-intensive service B.

SLO Latency Greeniac could also learn the optimal configurations with different SLO targets for the 95%ile latency. For an SLO target of 500ms, the 4-server cluster could only service loads up to 100rps before violating SLO targets. To meet the stricter latency target Greeniac had to rely on big cores and consumed much higher power compared to when the default SLO target was 1 second. Since CPU power usage is proportional to utilization, I observe that, for 0.5s SLO, cluster utilization could barely reach 60% before violating the SLO margin. Thus stricter SLOs cause Greeniac to use big cores with low utilization and leads to higher power usage and lower cluster efficiency.

I also observed that an extremely lenient SLO of 10s does not gain any significant energy saving

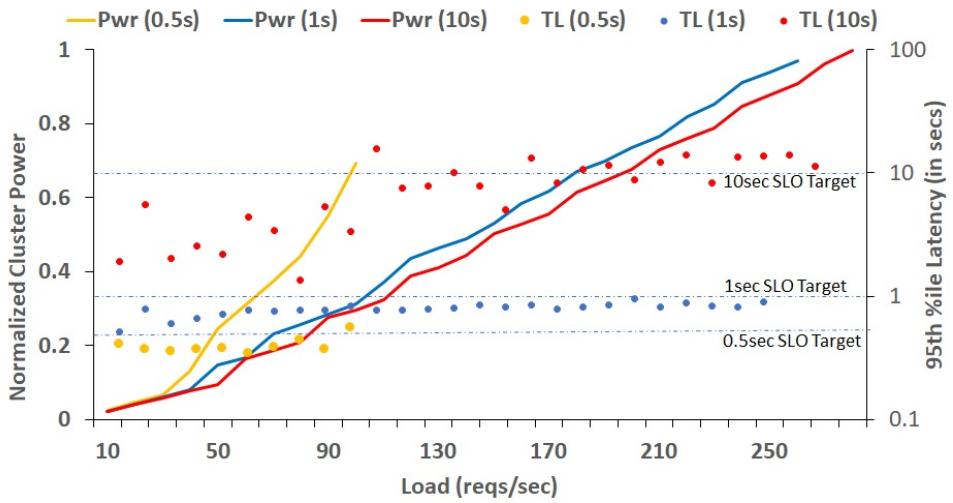


Figure 2.15: Effect of SLO target on cluster power and tail latency on 2B-2S HMP cluster with 4 servers. Same service was executed with 3 different SLO targets (shown as dotted lines) for the 95 %ile latency (right y-axis). The power consumption for each case is normalized against the maximum power of the cluster. With stricter SLOs, cores start violating SLO at lower utilization levels. Thus lower the SLO target higher the number of cores required for servicing a given load. Greeniac learns to use big cores at stricter tail latency targets, thus consuming higher power and meeting SLO target up to a lower cluster load.

for the cluster because it is harder for Greeniac to learn the optimal configurations that can meet latency targets. As shown in Figure 2.15, the tail latency for the optimal configuration inferred by Greeniac for the 10s SLO case is irregular for various loads. This is because a long latency target allows for a long request wait queue and can render the system unstable if the service rate drops below the arrival rate. Besides, the gradient descent described in Section 2.4.3.3 can get very steep for large SLOs since a slight change in CSC can greatly affect the tail latency and hence the reward function.

2.7 Related Works

Resource scaling [14, 25] and resource sharing [11, 25, 71, 27] techniques proposed in recent years for improving energy efficiency of LC-Services are limited in their effectiveness due to the lack of energy-proportionality in server processors. More recent works such as Hipster [33], Octopus-Man [34] and Haque et.al. [35] propose the use of HMPs, which have proven to be energy-efficient and widely deployed on mobile platforms. Octopus-Man [34] uses a PID controller to estimate

the best HMP service configurations based on recently observed tail latency. Hipster [33] models the server system as a MDP to learn optimal HMP configurations that meet the tail target for a single server. Haque et.al. [35] employ a PID controller to estimate the threshold latency for switching requests from smaller to bigger cores. These works all try to perform energy-aware scheduling only at the server-level, thereby leaving room for significant savings at the cluster-level, achieved by Greeniac. Some other works [71, 74] leverage server heterogeneity to associate different types of LC-Services to suitable server types. However, these approaches do not perform CPU-heterogeneity-aware scheduling.

Energy-aware task scheduling has been widely studied [75] for homogeneous clusters. Auto-scaling [21] techniques using bin-packing scheduling are popular in commercial cloud platforms. Tail latency-aware server scaling has shown to provide significant energy saving for low energy-proportional (EP) servers. However, with servers becoming more energy-proportional [15], energy saving from such scaling techniques are significantly lower. Due to the lack of use of HMPs in clusters, such studies have not been performed for HMP-servers. Energy efficient scheduling on HMPs has been studied in [76] but tail latency targets were not considered.

Machine learning techniques have been proposed to minimize energy usage on clouds [77]. Reinforcement learning [45] is recently gaining popularity due to its ability to learn models for complex systems with enormous state spaces. But to the best of our knowledge it has not been applied before to cluster level energy saving in any prior work. Hipster [33] employs RL only at a single server-level scale.

Heterogeneous architectures such as GPUs [55] and FPGAs have been employed to host LC-Services. Such accelerators can provide orders of magnitude improvements in latency and energy saving, but they are only suitable for specific applications and require an enormous porting effort making them unsuitable for generic online services.

2.8 Conclusion and Future Works

I developed a cluster-level control plane task manager for LC-Services called Greeniac. Greeniac uses reinforcement learning to achieve minimal energy utilization while guaranteeing tail latency constraints. I exploit the CPU heterogeneity in an HMP to save energy by using small cores instead of big cores at a cluster level whenever possible. My proposed work gains up to 28% power saving over previous works which perform HMP-aware local scheduling with a traditional cluster scheduler. I performed several tests to verify the functionality and perform deeper analysis.

As future work I see several opportunities and challenges for Greeniac. First, due to the generic nature of Greeniac, it can be extended to heterogeneous clusters made of both HMP and non-HMP cores. Second, to support a truly dynamic cluster environment, Greeniac must support learning in a co-execution environment when an LC-Service shares HMP resources with other batch tasks and services. Finally, a further in-depth study of Greeniac for different cache characteristics and micro-architectural properties used by HMPs would be insightful for optimal HMP configurations with multiple classes of LC-Services.

Chapter 3

Cluster-level Control Plane Strategies to Exploit Cluster Heterogeneity using Heuristics

3.1 Background

In the previous chapter, I demonstrated the opportunity for energy saving that exists if CPU heterogeneity is exploited. I also presented control-plane strategies to make the most out of these opportunities. But due to the lack of popularity among CSPs, strategies meant for heterogeneous CPUs can currently only be applied to edge and fog domains. However heterogeneity, as a challenge, is also growing in the cloud warehouses in the form of Cluster Heterogeneity. Over the years, CSPs have sought to acquire more compute resources, both to meet growing compute demands and to replace worn out resources. With the arrival of newer, faster, larger, energy-efficient and energy-proportional processors, CSPs continue to heterogenize their clusters by employing these newer generation processors.

Unlike CPU heterogeneity discussed in the previous chapter, cluster heterogeneity is not one-dimensional. In the case of HMPs, it is typical that the bigger faster core is less energy efficient

that the smaller slower core. But *Cluster Heterogeneity*, which uses multiple different CPUs, leads to two-dimensional heterogeneity in both *throughput* and *power* consumption for service instances. Traditional resource managers (RM) such as Kubernetes scale the number of LC-Service instances and distribute user requests among instances assuming cluster homogeneity. My study reveals that naively assuming identicality for instances hosted on heterogeneous clusters can lead to both SLO violations and energy wastage. An energy-optimal LC-Service execution requires adapting the control-plane strategies, i.e. both *Instance Scaling* and *Load Balancing* of request traffic, to be heterogeneity-aware.

Though the challenges presented by cluster heterogeneity have been acknowledged by numerous past research works [78, 37, 79, 80] and addressed specifically for data centric Map-Reduce tasks [80, 81, 82, 83], such research is lacking for latency-critical microservices. Prior research approaches cannot be applied directly to microservices-based implementations due to their significantly different control plane components. To this end, a thorough impact analysis of the *Capacity* and *Efficiency* heterogeneity among serving CPU cores on the aggregate cluster-level *Tail latency* and *Energy footprint* of an LC-Service is missing from the current literature. Such an analysis can lead to models and heuristics for more efficient heterogeneity-aware cluster-level instance scaling and load balancing strategies.

The main contributions of this chapter are:

1. I show that heterogeneity-unaware request distribution among server instances on a heterogeneous cluster can lead to poor server utilization. Based on my experimental observation, I propose a novel Maximum-SLO-Guaranteed-Capacity (MSG-Capacity) proportional request distribution technique to address the capacity heterogeneity among CPUs, and show that it can achieve higher utilization.

2. To address efficiency heterogeneity, I propose a simple Efficient-first (E-first) heuristic for request distribution among the heterogeneous instances.

3. To address both capacity and efficiency heterogeneity, I propose an Energy-efficiency aware and MSG-Throughput (E-MT) based request distribution heuristic that maximizes utilization

while minimizing the energy footprint of the target LC-Service.

3.2 Impact of Heterogeneity

3.2.1 Cluster Heterogeneity: Scope of the Work

Heterogeneity in cloud servers can be across many different attributes of the system configuration, including general and special purpose compute units, memory, network, and others. In this study, I do not consider heterogeneity in memory and network resources, assuming they have sufficient capacity. Rather, due to market changes, I focus on heterogeneity among the compute units within a cluster. Further, I do not consider workload distribution for accelerators or HMPs. Rather I consider heterogeneity in server clusters made of general-purpose symmetric multiprocessors (SMPs) which are heterogeneous in terms of the core performance capacity and energy efficiency as discussed below. I assume that all servers with identical CPUs to be of identical system configuration and that there is no interference from co-executing applications. In this work I do not consider or exploit CPU heterogeneity or Core-frequency heterogeneity.

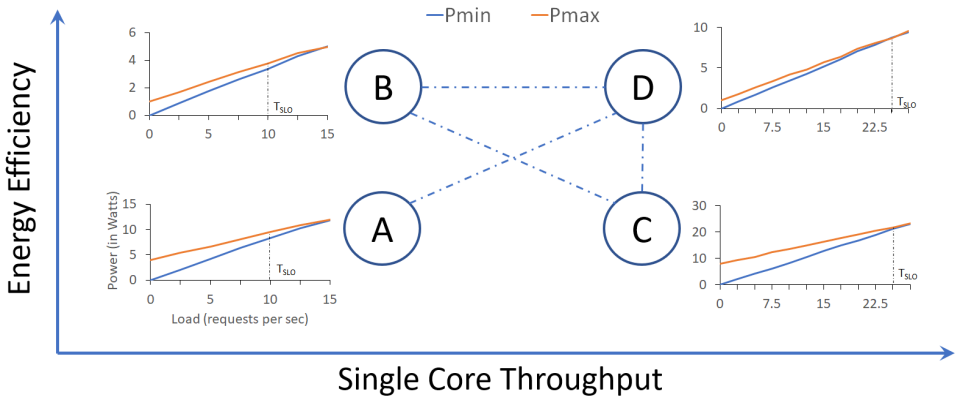


Figure 3.1: Heterogeneity across server processors in two dimensions - performance capacity (or throughput) and energy efficiency. 4 processors are considered, for each of which the power consumption is measured as LC-Service load varies from 0 to 100%. The power characteristics of each processor is shown as a range between the ideal (throughput-proportional) power P_{min} and maximum (real) power P_{max}. The T_{SLO} refers to the MSG-Throughput of each processor which is described later in the chapter. C and D achieve twice the maximum throughput of that of A and B in requests per second. B and D are almost equally efficient and achieve close to 2.5 times the energy efficiency of A and C in Joules per request. Thus, for each dimension I consider two levels - low and high.

I consider heterogeneity among CPUs along two dimensions, namely, Capacity T and Efficiency E. T denotes the average number of requests serviced per second at maximum utilization. E is defined as the requests serviced per Joule at maximum utilization. Both Capacity and Efficiency values are application-specific for every CPU core type. For a specific LC-Service instance, its throughput (i.e. average request service rate) and energy footprint depend on the Capacity and Efficiency of the hosting CPU core. I assume the instance throughput and energy consumption are linearly proportional to core utilization, and I also assume that for identical CPU core types, Capacity and Efficiency metrics are identical. Hence for instances hosted on identical core types, the throughput and energy footprint are assumed to be identical. The goal of this study is to answer the following questions. (i) What are the benefits/disadvantages of cluster heterogeneity? (ii) Given a set of CPU cores of 2 or more types, what mix of heterogeneity optimizes the Tail-Energy problem for a specific service at a given request load? (iii) What is the ideal request distribution strategy across the service instances that meets the SLO while minimizing energy usage? An analysis of these questions can help CSPs understand the potential trade-offs associated with increasing cluster heterogeneity along Capacity and Efficiency dimensions. Additionally, the analysis helps us infer optimal instance scaling strategy over the mix of cluster CPU cores.

3.2.2 Research Strategy

Setting up a real large scale heterogeneous cluster can be quite expensive. It would also be quite restrictive in terms of the mix of processors deployed and the scale of the heterogeneity. Before investing heavily in acquiring newer varieties of servers, CSPs need to understand how a potential micro-service would perform on the target heterogeneous platform. To perform a comprehensive study across a wide spectrum of processors by varying the level of heterogeneity, we perform a model-based simulation study.

For each of the two aforementioned CPU dimensions we consider two levels, low(l) and high(h). Consequently, consider 4 hypothetical processors (A-D) as shown in Fig. 3.1. A(T_lE_l) represents inefficient and slower legacy processors (e.g. legacy Xeon Server processors). B(T_lE_h) is slower but efficient processors (e.g. Atom or ARM processors). C(T_hE_l) is the set of fast but less effi-

cient processors (e.g. Haswell Xeon Server processors) and D(T_hE_h) represents the latest or future processors with high performance capacity and efficiency (e.g. AMD Ryzen, Intel Core-i9). These characteristics of the processors are based on their single thread performance and typical TDP reported in [84]. To start with, I perform a model-based analysis of 4 possible heterogeneous scenarios comprised of 2 CPU types, namely, A-D, B-D (equivalent to A-C), B-C and C-D (equivalent to A-B), as shown in Fig. 3.1.

3.2.3 Simulation Model

Design: For the initial analysis I extended my custom simulator, HMP-ClusterSim (described in Chapter 2.5.1), to simulate a heterogeneous cluster with multiple server types as shown in Figure 2.4. In general, each server can be a heterogeneous multiprocessor (HMP). However, for this work, I configured each server CPU to be an SMP consisting of a number of identical cores, which can be one of the four types (A through D) shown in Figure 3.1. Each server may host multiple service instances. I assume each instance requires a maximum of 1 CPU core and is mapped to a single physical core. Each service instance inherits the capacity and efficiency characteristics of the assigned CPU. I also assume negligible resource interference from co-executing applications on the cluster. For a given micro-service, a number of service instances across multiple non-identical CPUs may be instantiated. A single proxy-server (Gateway Node) receives all user requests and distributes them across multiple server instances using a Load-Balancing Strategy (LB). The Gateway Node is assumed to be not limited by In-cast or Out-cast bottlenecks for the maximum service load. Request arrivals at the gateway follow a Poisson process and the service time of requests on each core follows an exponential distribution.

Setting Capacity Parameters: Again, I consider the Lucene [60] workload (as also in previous chapter) as my target workload for obtaining practical Capacity and Efficiency parameters for the CPUs (A through D). To obtain these values, I measured the single thread throughput for Lucene on an Intel Haswell(Xeon) E-2637 processor at various CPU utilization levels. To better understand the challenges and impact of heterogeneity, I assume the capacity metrics of cores C and D to be

identical, i.e. instances on C and D have same throughput at any given utilization level. Similarly, I assume the capacity metrics of A and B to be identical. The performance capacity (i.e. maximum throughput for hosted instances) of A and B are conveniently chosen to be half of C and D, to assist analysis of simulation results. This is a valid assumption because along the capacity spectrum of processors [85], it is possible to find several pairs of processors with performance capacities in the ratio of 1:2.

Setting Efficiency Parameters: To get the energy characteristics of the processors, the power values gathered while executing Lucene service instances are measured on an Intel Haswell (Xeon) E-2637 processor. The obtained power values at various request load levels, ranging from 0 to maximum throughput, determine the power characteristics of processor C. The same process is repeated for an Atom core to obtain the power characteristics of B. The energy footprint of A is chosen such that its efficiency is similar to that of C. I calculated the energy efficiency as the ratio of throughput to power consumption. Since C is assumed to have twice the throughput of A, the power characteristics of A are appropriately scaled for its supported load levels. Similarly the power characteristics of D are obtained by assuming it to have similar efficiency as that of B The rationale behind these assumption is to study the one-dimensional capacity heterogeneity in A-C (T_lE_l-T_hE_l) or B-D (T_lE_h-T_hE_h) cases (Section 3.3). The efficiency measure of the 4 processors at their respective MSG-Capacity (explained below), are 1.21 (A), 2.95 (B), 1.15 (C) and 2.88 (D), as per their individual power characteristics shown in figure 3.1. Thus, B (and D) achieve approximately 2.5 times the efficiency of A (and C).

T_{SLO} denotes the maximum request load that meets the SLO guarantee for a processor type. T_{SLO} for a processor is application-specific and can be experimentally determined or derived from queuing-theoretic simulations as described in 3.3.2.1. I will refer to this as the *Maximum SLO Guaranteed Capacity (MSG-Capacity)* in the remainder of the chapter and define it more formally in the next section. Even though the parameters chosen for core types A, B and D are hypothetical in my simulation, their performance and power characteristics are quite realistic and closely match the single thread performance and TDP metrics of some existing processors [84, 85, 86].

3.2.4 Terminology and Key Metrics

Prior to my analysis, I clarify some of the terms and define the key metrics that have been used in this study:

1. Each application service instance is uniquely assigned to a single virtual processing node. Henceforth, I will refer to each service instance as a server. Furthermore, since each virtual processing node is assumed to be uniquely mapped to a single physical core, the terms core and server are used interchangeably where required. Multiple servers may be instantiated by a resource manager to be hosted on a physical *server node*.

2. The set of all active service instances (servers) for an application service form a *cluster configuration*. For my analysis, I explore cluster configurations consisting of at most two types of servers. For example, a 2B1D cluster configuration is a multi-server configuration consisting of 2 B-type and 1 D-type servers. The cluster configuration for a micro-service changes dynamically with instance scaling.

3. Maximum-SLO-Guaranteed (MSG)-Capacity of a core represents the maximum throughput that can be achieved by a server while still meeting the SLO for a single-server cluster configuration.

4. P99 is defined to be the latency for which 99% of the received requests have latencies less than P99. P99 has been evaluated at cluster-level and at individual server-level. For a specific server, P99 is statistically evaluated from latencies of requests serviced by that server. In this study, the SLO is *cluster-level P99 of 1sec*, unless otherwise stated.

5. All the results are shown with respect to a normalized load, which is the request rate normalized by the aggregate performance capacity of all cores in the cluster. For the static analysis in Section 3.3- 3.4, a cluster configuration also represents the cluster size. Hence a 1C1D configuration is assumed to be hosted on a cluster with only 1 C-type and 1 D-type core.

6. The terms request and task are used interchangeably throughout the chapter.

3.3 Analysis of Capacity Heterogeneity

For analyzing capacity heterogeneity, I consider a cluster of $B(T_lE_h)$ and $D(T_hE_h)$ type cores. The cores have identical efficiency but differ in capacity.

3.3.1 Heterogeneity-unaware Request Distribution

To demonstrate the overhead of heterogeneity-unaware request distribution, I first consider a 1B1D cluster configuration. The incoming requests are uniformly (randomly and equally) distributed to the two servers. The maximum supported load that can be serviced by this configuration is measured by measuring its throughput at 100% utilization. For a generic representation of results, the incoming request load is normalized with respect to the maximum supported load. I scaled the normalized load from 0 to 1. For various load levels, I measured the core utilization and P99 latency for requests serviced at each server. Additionally, I also measured the cluster-level P99 latency, denoted by P99(B-D), which represents the effective P99 response latency perceived by end-users.

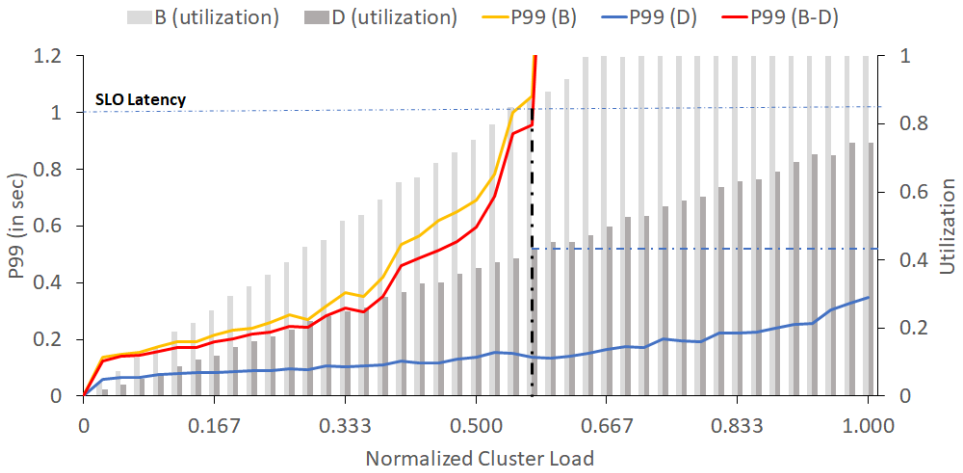


Figure 3.2: Impact of uniform and random request distribution on a heterogeneous 1B1D cluster. The lines shown as P99(B), P99(D), and P99(B-D) are P99 latencies (left y-axis) achieved in the B-server, the D-server, at the cluster-level, respectively. The bars represent the utilization level (right y-axis) of the B and D servers at different cluster loads. The cluster load is normalized with respect to the maximum service capacity of the cluster at 100% utilization of all servers. For a uniform load distribution, cluster-level tail latency closely follows the tail latency of the slower processor since its utilization saturates at a lower load.

From Figure 3.2 I observe that P99(B-D) could reach only 56% of cluster performance capacity before exceeding the SLO. At this point, the D-type server was operating at only 44% utilization. This happens because, with equal request distribution, both servers B and D service similar number of requests. However, since server B registers only half the throughput of D (as shown in Figure 3.1), its P99 latency (i.e. P99(B)) violates the SLO target at a lower server-level load than D, which is still underutilized and has a low P99 latency (P99(D)). Beyond this load level, a rapid increase in high latency requests serviced by B pushes the cluster-level tail latency beyond the SLO margin. The two intuitive observations from the above results are : (a) homogeneous load distribution across a heterogeneous cluster can saturate the slow servers while under-utilizing the faster ones, and (b) The cluster-level P99 latency lies between the P99 values of the most SLO-violating core (i.e. core B) and the least SLO-violating core (i.e. core D). In the above study, the SLO violation at 56% is caused by high P99(B), which is primarily due to queuing delays on saturated B-type servers. Since 50% of requests are serviced by core B, failure to meet the SLO target by even 2% of its serviced requests can still result in an SLO violation by the cluster.

3.3.2 Heterogeneity-aware Capacity-based Request Distribution

Since the cluster P99 latency is driven by the P99 latency for the most violating core, it seems obvious that redistributing the load towards less violating cores can reduce the cluster P99. To introduce such heterogeneity-awareness into the Load balancer, I first repeated the above simulation using an intuitive heuristic: Capacity-based Request Distribution. Here I divide the cluster load among the cores in proportion to their capacity. Since server D has twice the performance capacity of B (Figure 3.1), the cluster load was distributed randomly in a ratio of 1:2 between servers B and D.

3.3.2.1 Key Insights

Capacity-based distribution enabled the cluster to operate at 85% of capacity while meeting the SLO. As expected, capacity-proportional distribution led to equal utilization for both cores at all

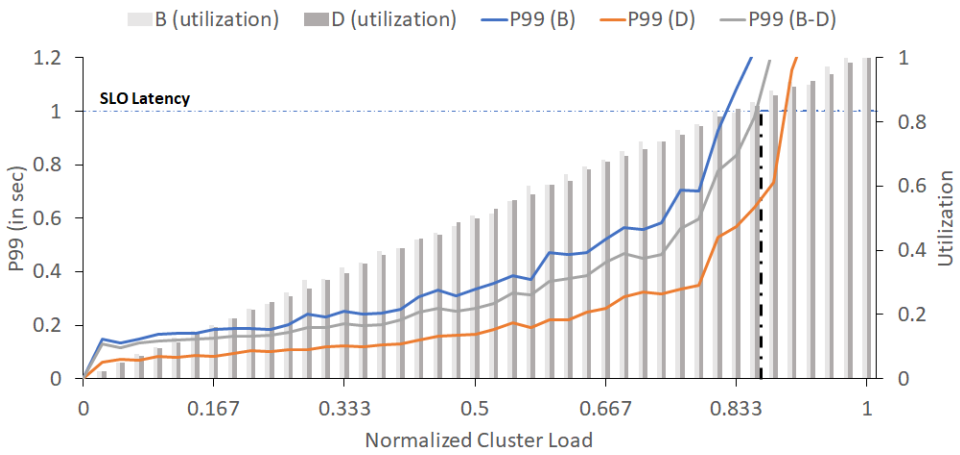


Figure 3.3: Impact of heterogeneity-aware capacity-based task distribution on a heterogeneous 1B1D cluster. P99(B), P99(D), and P99(B-D) are P99 latencies observed at the B-server, the D-server, and at the cluster-level, respectively. Capacity-based distribution forwards more requests towards faster processor. The cluster-level tail latency meets SLO up to a higher cluster load. But since faster processors start violating SLO at higher utilization level than the slower processor, the cluster level utilization can be further improved by further skewing the load distribution in favor of the faster processor.

load levels. But interestingly, despite equal server utilization, P99(B) was still higher than P99(D) at all load levels. Server D attained a higher utilization before producing the same number of late responses as server B, which led to few key insights:

Insight 1: Cores with different performance capacity violate the SLO at different average utilization levels. Higher performance cores can be maintained at higher utilization levels without violating SLO requirements. This behavior is a characteristic of M/M/1 queuing systems wherein the probability of a request being serviced within a delay target of T is given by Equation 3.1. For a 99th percentile latency, this probability(p) is 0.99. The core utilization level when P99 equals T (the SLO latency target) is given by ρ_{slo} in Equation 3.2. ρ_{slo} is the maximum utilization achieved by a core while guaranteeing SLO. This will be referred to as *Maximum SLO-Guaranteed Utilization* or *MSG-Utilization*. Since the service rate mu scales linearly with core performance capacity and D has twice the service rate of B, it has higher MSG-utilization as given by Equation 3.2 and seen in Figure 3.3.

$$P(S < T) = 1 - e^{-\mu(1-\rho)T} \tag{3.1}$$

$$\rho_{slo} = 1 - \frac{ln(1-p)}{T\mu} \tag{3.2}$$

$$\frac{d\rho_{slo}}{d\mu} = \frac{ln(1-p)}{T\mu^2} \tag{3.3}$$

Insight 2: Another key observation can be drawn from Equation 3.3. For a given SLO target latency T, the utilization gap between B and D increases rapidly as the service rate reduces for the same capacity ratio. Hence the disparity between the ρ_{slo} for B and D, would greatly increase at lower service rates, even for the same capacity ratio. The fraction of late responses would be much higher for core B at low service rates, resulting in cluster-level MSG-utilization being relatively more influenced by MSG-utilization of server B than server D.

Insight 3: In Equation 3.2, $1/T\mu$ refers to the ratio of the average service time to the SLO target latency. The higher the ratio, the longer the queuing delay that a request can endure without resulting in a late response. A lower SLO latency target can accordingly reduce the MSG-utilization for cores, with a slower core achieving a lower MSG-utilization value.

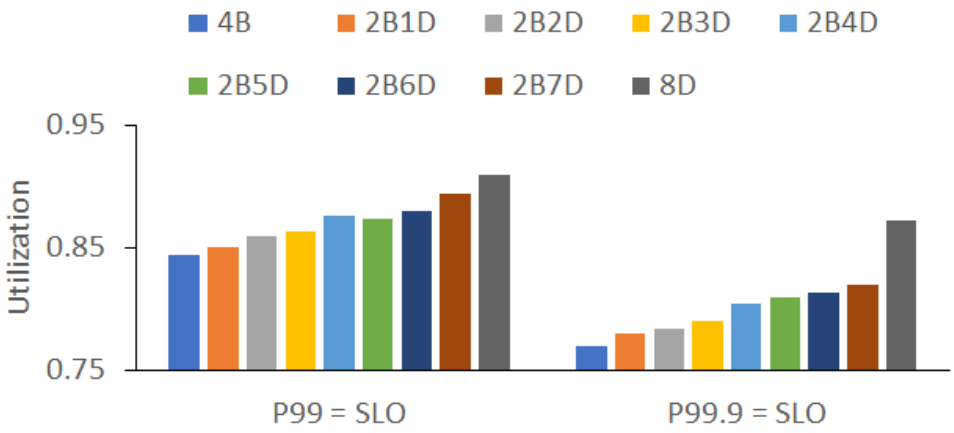


Figure 3.4: Variation in MSG-utilization for various heterogeneous clusters at different SLO strictness levels with a capacity-based load distribution. As the number of high-capacity servers are increased, the MSG-utilization gradually increases, maximizing for the homogeneous fast core cluster 8D. For a stricter P99.9 SLO, clusters employing slower processors (B) achieve much lower utilization. Even a few slow servers in 2B7D significantly lowers the MSG-utilization compared to the 8D configuration.

Insight 4: Increasing the strictness of the SLO target (i.e. p) also negatively impacts MSG-utilization for a core, as given by Equation 3.2. Figure 3.4 shows the MSG-utilization slack (i.e. $1 - \rho_{slo}$) of a homogeneous cluster (4B) increase by 50% by increasing the strictness from 99% to 99.9%. Additionally, with an increase in SLO strictness, cluster utilization is expected to become more sensitive to heterogeneity and influenced more by the slow servers. To verify this I compared the MSG-utilization of 9 clusters having a mix of B and D servers. Starting with a 4B cluster where all requests are served by the B servers, I gradually increased the number of D servers, ranging from 2B1D up to 2B7D. For the 8D cluster, all requests are served by the D servers. For a 99%ile strictness the MSG-utilization steadily increases with increase in the number of D servers. There is only a 1.5% gap in the MSG-utlization levels of 2B7D and 8D clusters. However, for a 99.9%ile strictness, the gap widens to 6.5%. Note that this degradation is observed for 2B7D cluster despite having several fast cores and few small cores. This implies that even a few low performing servers can pull down the MSG-utilization of a large cluster configuration which uses a capacity-based request distribution. This is expected since for a stricter SLO, even fewer late requests can cause an SLO violation for the entire cluster.

Insight 5: Capacity based distribution is clearly not the ideal load-balancing strategy. As shown in Figure 3.3, P99(B) is still higher than P99(D), implying that B is still the most SLO-violating core. By diverting more requests towards D, both P99(B) and P99(B-D) may be able to sustain greater cluster load. Specifically, if every core received a fraction of load such that its utilization does not exceed its MSG-utilization, the P99 latency of every server would be the same when the cluster level MSG-utilization is maximum. For such a load distribution, the number of late requests would also be equally occurring at both cores. This intuition served as the motivation for my proposed MSG-Capacity based load balancing strategy, discussed below.

3.3.3 MSG-Throughput based Request Distribution

To maximize the MSG-Utilization of cluster, I exploited the idea of capping the utilization of a core at its MSG-Utilization. Performing this requires restricting the core load to the Maximum

SLO-Guaranteed Throughput or $MSG\text{-}Throughput(\lambda_{slo})$ given by Equation 3.4 (from Equation 3.2).

$$\lambda_{slo} = \mu - \frac{ln(1-p)}{T} \tag{3.4}$$

This ensures the P99 latency for every core meets the SLO latency target, thereby guaranteeing the SLO is met for the entire cluster. Thus I propose a MSG-Throughput based load distribution strategy in which the request load is divided among servers using their corresponding MSG-Throughput.

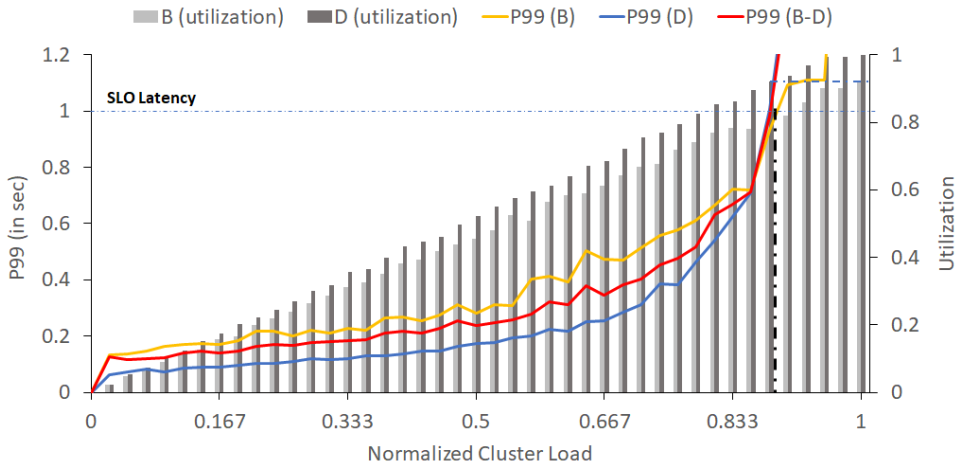


Figure 3.5: Impact of heterogeneity-aware MSG-Throughput-based task distribution on a 1B1D cluster. P99(B), P99(D), and P99(B-D) are P99 latencies achieved in the B-server, the D-server, and at the cluster-level, respectively. Since faster servers have higher MSG-utilization, this distribution is skewed further towards the faster server as compared to the capacity-based distribution. The cluster achieves higher utilization as both servers individual tail latencies merge at the same cluster utilization level.

To test the effectiveness of this approach I repeated the experiment from Section 3.3.2 with load balancing based on the MSG-Throughput ratios of cores B(T_lE_h) and D(T_hE_h). This ratio is different from the performance capacity ratio and is more biased towards fast cores. For example, while the capacity ratio of B to D in the previous experiment was 1 : 2, the MSG-Throughput ratio was found to be 1 : 2.27. Results show that at lower cluster loads, the P99 of slow server (B) and fast server (D) differ by a smaller margin compared to the Capacity-based distribution in Figure 3.3. However, at higher loads, when the server-level P99 approaches the SLO latency target, the P99 latency at both servers equalize. This new distribution enables the cluster to service up to 88% of the aggregate load capacity, a 3% improvement over the prior Capacity-based distribution. Additionally, D cores could now achieve up to 92% utilization, a 7% increase.

As observed, my proposed approach causes fast servers to run at a higher utilization level for the entire range of load. It does not try to achieve an identical P99 for fast and slow cores for the lower and medium ranges - rather as an individual server load approaches its corresponding MSG-Throughput level, the load balancing strategy ensures that each server gets enough load such that its P99 latencies get closer as they reach the SLO margin. At the threshold point, where the cluster-level P99 reaches the SLO target of the cluster, this load balancing ensures that the P99 latency of all servers are equal. This can be also theoretically verified from Equation 3.5 which provides the number of late responses for a cluster. Here p, p_B and p_D correspond to the probabilities that a request is serviced within the SLO target latency at cluster-level, at server B and at server D, respectively. λ and λ_B correspond to the arrival rates at cluster level and at server B respectively. For a given SLO (i.e. p $\leq$ 0.99), λ peaks when $p = p_B = p_D = 0.99$.

$$p\lambda = p_B\lambda_B + p_D(\lambda - \lambda_B) \tag{3.5}$$

When λ equals the MSG-Throughput of the cluster, B and D are at their corresponding T_{SLO}. Since we assume B and D have equal efficiency at their corresponding T_{SLO}, the energy footprint is also optimized at the MSG-Throughput of the cluster. This heuristic provides a simple load-independent partitioning strategy that maximizes Utilization for clusters with Capacity heterogeneity. Note that the disparity in P99 between the cores is inconsequential since the primary objective (the Cluster-level SLO) is met despite the P99 imbalance. Another important observation is the closing in of the P99 gap when the load reaches MSG-Throughput levels. This crucial feature of the proposed strategy enables the load balancing strategy to meet the SLO despite load variations, as long as the cluster-level load is within the aggregate MSG-Throughput at Cluster-level.

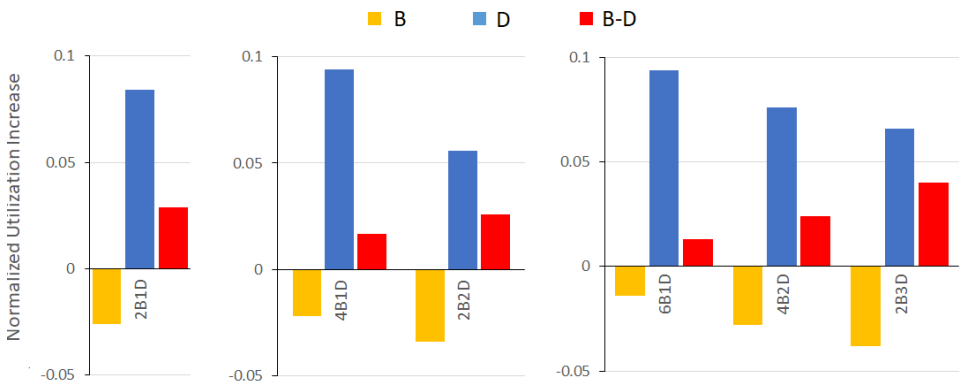


Figure 3.6: Improvement in maximum utilization observed for MSG-Throughput based distribution in 3 sets of clusters. Each cluster in a set has identical capacity but a different mix of heterogeneity. Clusters with higher number of fast servers (D) achieve higher cluster-level utilization improvement (shown in red) and greater reduction in utilization for slower (B) servers (shown in yellow).

3.3.4 Sensitivity Analysis of MSG-Throughput based Load Balancing

3.3.4.1 Better Resource Utilization

I observed that the proposed load balancing strategy results in an improved cluster utilization compared to a capacity-based distribution. By reducing the average load at slow servers and increasing it at fast servers, the overall number of requests processed increases. To determine the impact of cluster configuration on the server and cluster utilization, I determined the utilization levels for 3 sets of clusters, each with identical capacities (shown in Figure 3.6). For the first set I performed the study on a single cluster 2B1D to observe that the utilization of slower (B) server (shown as yellow bars) decreased, while the utilization of faster D server (shown as blue bars) increases. This led to a significant overall improvement in utilization at the cluster level (shown as red bars). Note that, here, the improvements shown at the cluster-level are for the maximum SLO-guaranteed utilization. In this proposed approach, the individual servers operate at their MSG-utilization levels when the cluster achieves maximum SLO-guaranteeing throughput. Thus the change in maximum utilization of B and D, as shown in Figure 3.6, refers to the difference between the MSG-utilization of a server and its utilization level achieved in a capacity-based distribution when the cluster attains its maximum SLO-guaranteeing throughput.

For the second set, I considered two clusters, 4B1D and 2B2D, both of equal capacity but higher

than that of the first set. I observed that by increasing the number of slow servers (from 2B1D to 4B1D), the cluster utilization reduces, despite the faster D server servicing slightly greater number of requests. When increasing the number of fast servers for the same capacity (from 4B1D to 2B2D), the cluster level utilization improves significantly. Even though the utilization improvement for individual fast servers is lower, together they serve a greater fraction of cluster load, thereby further reducing the slow server loads and increasing the overall cluster utilization.

Similar trends were observed for the third set of even higher capacity. For the three identical-capacity clusters 6B1D, 4B2D and 2B3D, the clusters gained greater utilization improvement with increase in the number of fast servers. Overall, among all tested clusters, 6B1D was seen to achieve the lowest improvement while 2B3D achieved the highest improvement in cluster utilization. Thus higher the share of slower server capacity in a cluster, lower the improvement, and, higher the share of fast server capacity, higher the cluster utilization improvement. On the other hand, server utilization of fast servers increases more on slow-server dominated configurations (e.g. 6B1D), since fast servers are more under-utilized in such configurations when using capacity-based load distribution (as discussed in Section 3.3.1).

3.3.4.2 Adapting to SLO Strictness

MSG-Throughput-based load balancing also adapts to the SLO requirements. To demonstrate this, I applied my proposed approach to the 2B7D configuration explored in Figure 3.4, which showed a drastic fall in utilization with increased SLO strictness for Capacity-based distribution. When the load is distributed based on MSG-Throughput, heterogeneous configurations achieve close to the homogeneous fast cores' utilization and throughput. It is worth noting that the MSG-Throughput for both servers change with SLO requirements. With stricter SLO (P99.9), the MSG-Throughput ratio shifts in favor of fast servers. Thus the stricter the SLO requirement, the closer the utilization achieved by the proposed approach gets to a homogeneous fast server configuration's performance.

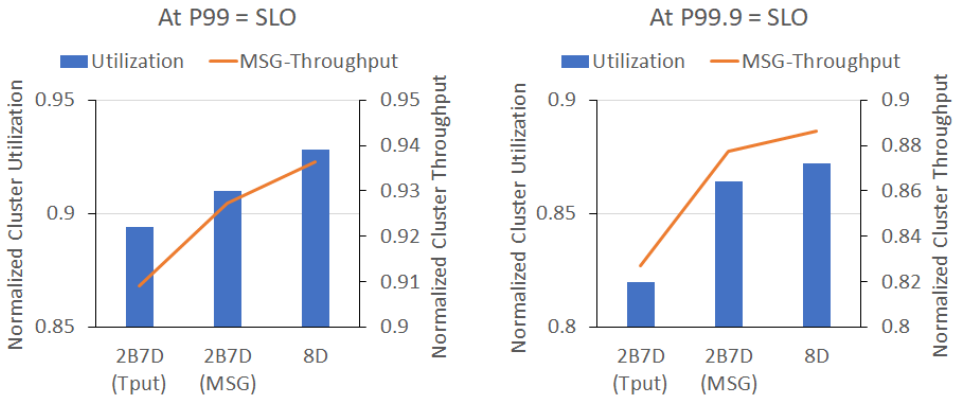


Figure 3.7: MSG-Throughput based distribution is more adaptive to SLO strictness compared to a capacity-based distribution. A 2B7D cluster achieves higher utilization level (blue bars measured by the left y-axis) with MSG-Throughput based distribution, even with stricter SLOs. The utilization levels of 2B7D get closer to an equi-capacity homogeneous 8D cluster with increasing strictness. The MSG-Throughput achieved by the cluster too follows a similar trend. Here the MSG-Throughput of cluster is normalized with respect to the total capacity of the cluster (given by the right y-axis) and is shown by the orange line.

3.3.5 MSG-Throughput based Instance Scaling

Though this section focuses on the load balancing aspect of the MSG-Throughput based control plane strategy, it must also be complemented with an instance scaling strategy to effectively service requests in a heterogeneous cluster. My proposed MSG-Throughput based scaling strategy is motivated by a HMP task-scheduling strategy employed in recent works [34, 33]. As per this scaling strategy, instances are hosted only on high capacity D-cores if the SLO could be met with an equal-distribution load balancing. Additional instances are scheduled on lower capacity B-cores only if SLO is expected to be violated for the current cluster load. Thus if no more high-capacity cores are available to the resource manager, additional instances are scheduled on available lower-capacity B-cores. When the cluster load decreases the Instance Scaler first deactivates the instances hosted on low-capacity B-cores, and D-core instances are deactivated only when the cluster configuration has no active B-core instances. Since fast cores have high MSG-Utilization, with this scaling strategy, the average utilization of the configuration is maximized. In addition, since both B and D are assumed to have identical efficiency, the scaling strategy should have no impact on the overall efficiency of the cluster configuration. For scaling up, if the resource manager finds an available D-core, the Instance Scaler first instantiates the microservice on the D-core and then deactivates

an instance hosted on B-core, if any. The number of active instances required to meet the SLO is calculated by the scaler based on the MSG-Throughput of the cores. CSPs may choose to maintain a buffer of active instances to handle the load spikes in [21]; however, such works to fine-tune the scaling routine is out of scope of my research.

3.4 Analysis of Efficiency Heterogeneity

For this study, I chose the case C-D (T_hE_l-T_hE_h) that possesses one-dimensional efficiency heterogeneity (i.e., clusters with throughput homogeneity but heterogeneity in energy efficiency). Due to the equal performance capacity of both core-types, the MSG-throughput based load balancing strategy results in equal request distribution across all cores. For a 1C1D configuration using MSG-Throughput based load balancing, the relatively inefficient server (C) consumes the majority of cluster power usage at any load. But the effective cluster efficiency varies with the configuration - the fewer the C-cores, the higher the efficiency. Similarly, increasing the number of D-cores can improve cluster efficiency when using an equal load distribution scheme.

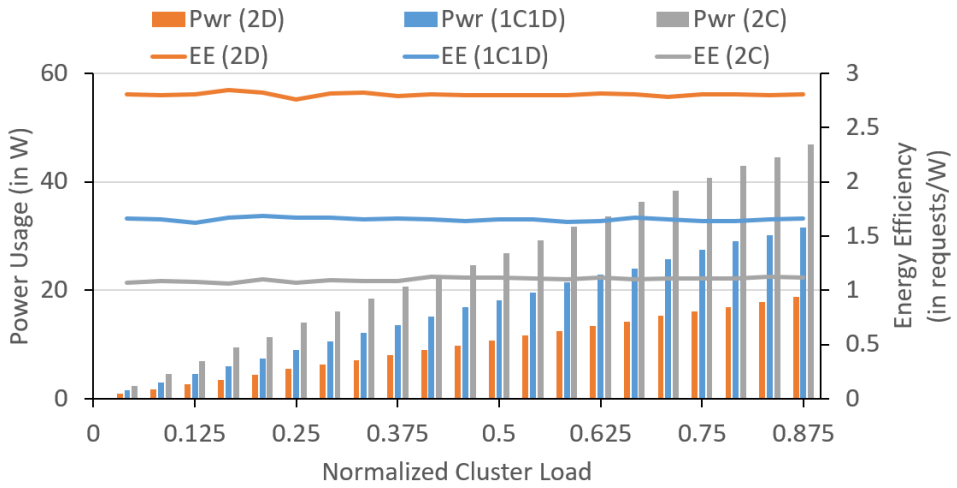


Figure 3.8: Influence of individual servers of a 1C1D configuration on the net power consumption and efficiency of the cluster at various cluster load levels and equal load distribution. Figure shows a comparison of 3 equi-capacity clusters: 2D, 1C1D and 2C. The bars represent the average power consumption (in Watts) shown by the left y-axis. The lines represent the average energy efficiency achieved (in requests serviced per Watt) shown by the right y-axis. The 1C1D cluster consumes more than the efficient 2D cluster but less than the inefficient 2C cluster. Due to an inverse relation with power, the energy efficiency of 1C1D is closer to that inefficient 2C cluster.

I measured the aggregate power usage for a 1C1D configuration over a range of cluster loads. I compared the results with homogeneous configurations of C and D which had equal performance capacity, i.e., 2C and 2D. As expected, the heterogeneous 1C1D cluster consumed less power than relatively inefficient 2C cluster but more than more efficient 2D cluster at any load. Though the power consumption of 1C1D is close to an average of the 2C and 2D clusters, its energy efficiency (represented by the blue line in Fig. 3.8) is relatively closer to the inefficient 2C cluster. This is due to the inversely proportional relation between energy efficiency and power consumption. For a generic nCmD configuration, the average efficiency of the cluster would be higher than C but lower than D for all load ranges. Intuitively, configurations with more efficient servers achieve higher efficiency. Thus, for any C-D cluster configuration, even though a load balancing strategy may not cause a tail latency violation, it does affect the cluster efficiency. Similarly, over the period of LC-Service execution, a good *Instance scaling strategy* can manage instantaneous cluster configuration for an LC-Service and improve its average efficiency dynamically while meeting the SLO requirements.

3.4.1 Efficient-First (E-First) Control Plane Strategy

Ideally, CSPs would like to maximize C-D (T_hE_l-T_hE_h) cluster's efficiency by getting as close to a D-only cluster as possible. To achieve this I employed an Efficient-First (E-First) strategy which relies on both efficiency-aware instance scaling and load balancing. Like in the previous section, the E-first instance scaling heuristic is motivated by HMP task-scheduling strategy employed in recent works[34, 33]. In this scaling strategy, instances are hosted only on efficient D-cores if the SLO can be met with an equal-distribution load balancing. Additional instances are scheduled on inefficient C-cores only if the SLO is expected to be violated for the current cluster load. If no more efficient cores are available, additional instances are scheduled on inefficient C-cores. When the cluster load decreases, the instance scaler first deactivates the instances hosted on inefficient cores. D-core instances are deactivated only when the cluster configuration has no active C-core instances.

While the E-First scaling maximizes efficiency, it must operate in tandem with the load balancer

to guarantee the SLO. Since for a C-D cluster all cores have equal capacity, they have equal MSG-Throughput as well. Hence no core should be operating at higher than their MSG-Utilization level. When serving requests to a homogeneous D-only type configuration, the load must be equally distributed to maximize cluster throughput while meeting the SLO. In contrast, when serving requests to a heterogeneous nCmD configuration, all of the m D-cores must operate at their MSG-Utilization level. This ensures that efficient D-cores serve the maximum possible number of requests without risking the SLO violation. The remaining load must be distributed equally across the n relatively inefficient C-cores. It is crucial that information about the load at the C-cores, as observed by the load balancer, be shared with the instance scaler. Since commercial CPU cores are relatively inefficient at lower utilization levels, this information will enable the Instance scaler to scale-down the number of C-cores when their load is low. Scaling down the number of C-cores increases their individual utilization and thus improves the cluster efficiency.

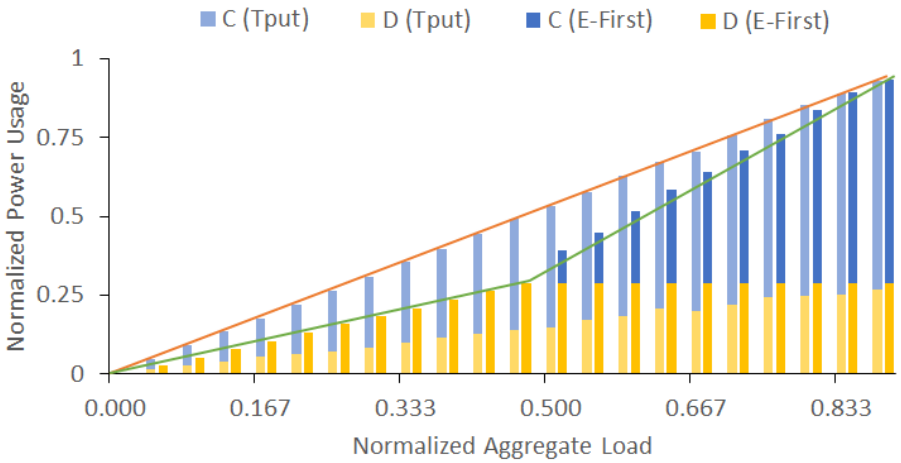


Figure 3.9: Power saving achieved by E-first load balancing strategy over a Capacity-based load balancing strategy (Tput) in a 1C1D cluster at various load levels. Individual power contribution of C and D servers are shown by blue and yellow bars respectively. Power usage shown is normalized with respect to the maximum aggregate power usage of the entire cluster at 100% utilization. Load levels are also normalized with respect to the maximum service capacity of the cluster (i.e. cluster throughput at 100% utilization). E-First employs only efficient D servers to service lower loads until their MSG-Capacity is reached. At higher loads, inefficient servers are employed and the load balancer maintains the load levels of efficient servers at their MSG-Capacity to maximize cluster efficiency. The power saving (denoted by the triangular delta region) is highest when only efficient D servers are employed and they operate at their MSG-utilization levels.

I applied the E-First strategy to a 1C1D configuration. As shown in Figure 3.9, unlike for the

capacity-based distribution (Tput), E-First performs a fully biased distribution in favor of efficient servers. At low load, only efficient servers are employed, until the load reaches the MSG-Throughput level of the efficient D-servers. At any given load, this strategy saves energy by employing efficient servers to perform the maximum processing possible while meeting the SLO target. Lesser processing time and lower energy is spent by relatively inefficient servers.

3.5 Analysis of Bi-dimensional Heterogeneity

I used inferences from previous sections regarding MSG-Capacity-based and E-first strategies to develop a two-dimensional heterogeneity-aware Energy-efficient MSG-Throughput (E-MT) strategy. E-MT involves using only the efficient servers for service execution when the server load level is below the aggregate MSG-Capacity of the efficient servers. Only the minimum number of servers needed to meet the current load are employed at any time, provided that an individual instance load does not exceed MSG-Capacity. Upon further load increase, the minimum number of additional inefficient servers are employed such that none of them receives more load than their MSG-Capacity. Since efficient servers improve the aggregate cluster efficiency, E-MT maintains a load of MSG-Capacity on them, while the inefficient servers may remain underutilized.

In the previous section, I demonstrated the efficacy of E-First strategy with C-D (T_hE_l-T_hE_h) heterogeneity. Since B-D (T_lE_h-T_hE_h) is considered homogeneous with respect to energy efficiency, using the E-MT strategy would result in effectively the same energy usage, cluster utilization and cluster throughput as using only the MSG-Capacity based distribution. While the two prior cases only presented one-dimensional heterogeneity, in this section we explore the effectiveness of E-MT with two-dimensional heterogeneity i.e. with B-C (T_lE_h-T_hE_l) and A-D (T_lE_l-T_hE_h) heterogeneity.

3.5.1 E-MT on B-C Heterogeneity

As shown in Figure 3.10, E-MT uses only the efficient server (B) during the low load phase. However, when the cluster load reaches approximately 30%, the load at B reaches its MSG-Throughput level.

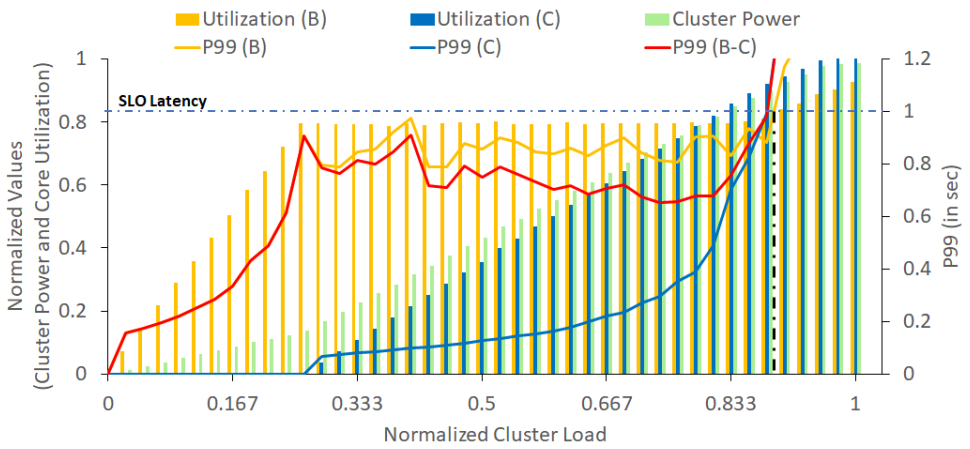


Figure 3.10: Performance of E-MT on a 1B1C cluster. Utilization of servers B and C at various cluster loads are normalized with respect to their respective maximum utilization and shown as bars. The cluster-level power consumption is normalized with respect to the maximum cluster power and is show as green bars. These normalized values are shown along the left y-axis. The lines denote the P99 latency values (shown by right y-axis) for individual servers B and C and of the entire cluster. As load increases E-MT employs only more efficient B servers until their MSG-Capacity is reached. At this point, the P99 latency of the B-server (and the cluster) reaches close to SLO target limit (blue dotted line). Upon further load increase, E-MT forwards additional load to the less efficient C servers due to which the cluster power consumption starts increasing more rapidly. Due to a higher MSG-Capacity of the faster C server, E-MT can keep meeting cluster-level SLO up to a much higher load beyond the MSG-Capacity of the B server. P99 latencies start violating the SLO when the cluster load exceeds the MSG-Capacity of the cluster (black dotted line).

Upon further load increase, E-MT begins employing inefficient servers and diverts any additional traffic load towards them. The figure shows that the cluster power starts increasing more steeply due to the use of inefficient servers. The load-balancing strategy ensures that the B-core instances do not receive more than their MSG-Throughput worth of load even as the cluster load increases. Due to this cap on the server load, the P99 for the slow B-server is observed to remain just below the SLO as the cluster load increases further, until the server load at the fast C-server reaches its corresponding MSG-Throughput level. Beyond this point the load is distributed based on the MSG-Throughput ratio of B to C, though it does result in violation of the SLO for cluster. Since aggregate capacity is the same as previously discussed in the B-D case, E-MT achieves the same MSG-Capacity (88% of aggregate capacity) as it did for the B-C case. It is worth noting that though E-MT employs slow servers first in case of B-C heterogeneity, this does not cause under-utilization of fast servers. The E2MT instance scaling ensures that additional fast cores are used only when the cluster load cannot be serviced with all active cores functioning at their MSG-utilization levels.

3.5.2 E-MT on A-D Heterogeneity

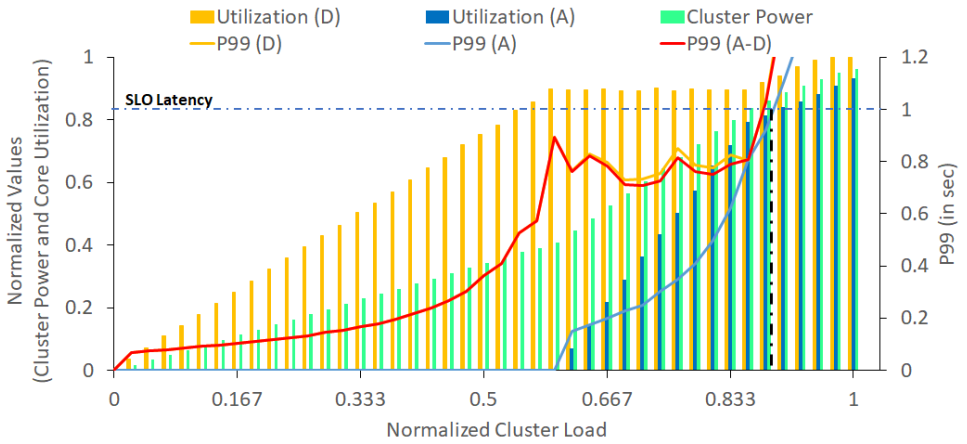


Figure 3.11: Performance of E-MT on a 1A1D cluster. Utilization of servers A and D at various cluster loads are normalized with respect to their respective maximum utilization and shown as bars. The cluster-level power consumption is normalized with respect to the maximum cluster power and is show as green bars. These normalized values are shown along the left y-axis. The lines denote the P99 latency values (shown by right y-axis) for individual servers A and D and of the entire cluster. As load increases E-MT employs only more efficient D servers until their MSG-Capacity is reached. At this point, the P99 latency of the D-server (and the cluster) reaches close to SLO target limit (blue dotted line). Upon further load increase, E-MT forwards additional load to the less efficient A servers due to which the cluster power consumption starts increasing more rapidly. Due to a lower MSG-Capacity of the slower A server, E-MT meets cluster-level SLO for a relatively small additional load beyond the MSG-Capacity of the D server. P99 latencies start violating the SLO when the cluster load exceeds MSG-Capacity of the cluster (black dotted line).

For A-D heterogeneity, E-MT employs only efficient (D) servers during low load (as shown in Figure 3.11). D servers, being fast and having higher MSG-Capacity, sustain a higher cluster load. After the cluster load reaches D's MSG-Throughput, E-MT begins employing the less efficient A server. Server A, being slow, reaches its own MSG-Throughput at a small server load. When both A and D's server loads reach their corresponding MSG-Throughput, their P99 and the aggregate cluster P99 starts to violate the SLO at any further load increase. The above results are a demonstration of the fact the E-MT strategy can be successfully applied to any server heterogeneity type to maximize cluster level energy efficiency, throughput and server utilization for LC-Services hosted on a heterogeneous server cluster.

3.5.3 Employing E-MT on Real Multi-level Heterogeneity

In a large-scale heterogeneous cluster with more than 2-levels for each degree of heterogeneity, E-MT must maintain an ordered set of servers sorted by their energy efficiency. The load distribution strategy again starts employing efficient servers first at very low loads, until all such available servers are operating up to their corresponding MSG-Throughput. With further load increases, additional servers from the next efficient server in the ordered set are employed. The strategy ensures that (i) at a given time only the most efficient available servers in a cluster are employed, (ii) the slower servers do not cause under-utilization of faster servers, and (iii) the aggregate load serviced by the cluster is maximized while meeting the SLO.

3.6 Considerations for Control-plane Implementation

While the heuristic-based E-MT strategy may seem simple, applying it at the control plane can be challenging for a variety of reasons.

3.6.1 Challenges in Determining MSG-Throughput

1. MSG-Throughput for a core is application-specific. Shorter tasks result in larger throughput, so the throughput for a microservice needs to be measured for each core type before applying any control-plane strategies.

2. The MSG-Throughput value derived from Equation 3.2 is based on the assumption of a Poisson arrival and exponential service time distribution. Though these are widely accepted theoretical assumptions, real world distributions are noisy and often do not fit any standard distributions. Therefore online measurements need to be performed on each core type with sample traffic loads to obtain the real MSG-Throughput of each core.

3. In this work I did not consider Core-frequency Heterogeneity. However, most current processors support dynamic voltage-frequency scaling (DVFS), which enables them to run at lower

frequency and save energy during periods of low utilization. MSG-Throughput measured for my experiments assumed a fixed base frequency for cores.

4. Additionally, interference from co-executing workloads can result in incorrect determination of MSG-Throughput for microservices.

3.6.2 Challenges in Determining Energy-efficiency

1. Although energy-efficiency is shown (in Figure 3.9) to be fixed for the entire load spectrum, this study assumes a lack of idle power consumption in cores. While most cores enter low power sleep states upon being idle, many are not configured or capable of doing so. Thus efficiency can vary with load level.

2. As discussed before, different frequency levels due to Core-frequency heterogeneity result in different levels of energy-efficiency. Additionally, recent studies have shown that DVFS-capable systems demonstrate energy super-proportionality [15, 87], wherein peak efficiency in a system is achieved at intermediate frequencies.

3. Most modern CPUs support multiple sleep states, each with different power levels and inter-state transition delays. Depending on the inter-request arrival time and traffic pattern, CPUs might end up spending a variable amount of time in low power sleep states despite having the same idle time [88]. Thus the traffic pattern can also result in variations in the energy efficiency of a core. Moreover, CSPs may choose to disable some of the states to reduce wakeup delays, so the same core type might achieve different efficiency values based on host server configurations.

4. CSPs are concerned about the aggregate server power consumption which also includes the power used by components other than the CPU. The efficiency of a server can vary based on the efficiency of its non-CPU components, e.g. memory, network card, storage, etc. An efficient core may still result in an inefficient server if it has inefficient server components.

5. Inferring an energy efficiency value for a multi-core platform can be very challenging. With a shared Uncore containing LLCs, memory controller, etc., and multiple workloads potentially

executing on a multi-core CPU, energy footprint due to a single service instance would be hard to infer. Thus effective efficiency of a core can also be dependent on co-executing workloads using shared core resources.

6. Finally, the energy-efficiency of a core is application-specific. A memory-intensive microservice would have a different power profile than a compute-intensive one. Efficiency metrics for each service may need to be measured on-the-fly on production servers before applying any control-plane strategies.

3.6.3 Instance Scaling Considerations

In a large-scale cloud deployment, scaling is performed by cluster-wide instance scaler components such as Amazon EC2 AutoScaling [89]. To implement E-MT for a service, scalers must obtain runtime metrics for cluster-level loads from the services' load-balancer. Using the runtime load-metric and pre-computed MSG-Throughput of cores, the instance scaler checks if the current set of cores hosting active service instances can meet the SLO. If not, the instance scaler invokes the resource manager to obtain an available core. To implement E-MT, per-core-type efficiency metrics for the service must be provided to the resource manager, which must then return the most efficient core available. The instance scaler then schedules an instance creation on this core. The instance scaler also checks if any active core with efficiency lower than the newly added core can be released without violating the SLO. If so, the instance hosted on the low-efficiency core is deactivated after the new instance is brought up on the high- efficiency core. Note that, the instance scalar may employ multiple available efficient cores, if needed, both to meet the SLO and to minimize the number of less efficient active cores.

If, however, the aggregate MSG-Throughput is substantially higher than the cluster load, the instance scaler needs to perform a scale down and deactivate one or more instances. To do this, the least-efficient active core type is deactivated first. The instance scaling strategy relies on the load balancer to effectively distribute the requests among active instances. Large-scale instance scaling is a complex task requiring several additional considerations, such as avoiding oscillations, identifying the optimal time interval for metric measurements and scaling decisions, workload prediction for

the next epoch, etc.. However, policy, algorithm and parameter selection for such aspects are independent of an E-MT implementation.

3.6.4 Load Balancing Considerations

The application layer L7-Balancers and network layer balancers such as Amazon Elastic Load Balancers (ELB) [90] and HAProxy are typically employed for distributing request traffic across LC-Service instances on large cloud platforms. The proposed E-MT strategy can be applied to these L7-balancers to minimize the aggregate energy footprint of all the server instances. With scaling, the load ratio must be adapted for each service instance, thus a weighted round-robin or dynamic round-robin load-balancing technique would serve as the best strategy to distribute the load across the instances as per proportions determined by E-MT. As mentioned previously, instance scaling updates must be relayed to the load balancer immediately. The load balancer must then dynamically adapt the load ratios so that all efficient cores receive up to the MSG-Throughput load, while the rest of the load is distributed across inefficient cores as per the E-MT strategy.

Some cloud L7-balancers may also support additional functionality such as rate pacing or queuing, which can affect inter-packet delays and traffic patterns at a server. This in turn can result in different tail behavior at the server and cluster level. However, studies regarding different load balancing [57, 56] and traffic shaping strategies are beyond the scope of this work.

3.7 Related Works

Motivated by the seminal works regarding cloud efficiency concerns by Barroso et.al. [12, 91], a huge amount of research [92, 10, 27, 93, 59, 94, 24, 17, 14, 11, 95] has been performed to improve the energy efficiency of cloud platforms while guaranteeing LC-Service QoS. However, these works do not explore the effect of cluster heterogeneity in cloud servers.

Many works have explored the benefits of server heterogeneity created by introducing heterogeneous compute platforms such as GPUs [96], TPUs [97] and FPGAs [98] into the cloud. LC-Services have

also been shown to significantly reduce the energy footprint on these platforms [99]. But such benefits are application specific, and cannot be experienced by generic applications. Some other works [34, 35, 33, 76] have demonstrated improvements in cloud efficiency by exploiting CPU Heterogeneity by using heterogeneous multiprocessor (HMPs), such as Arm Big.little processors [29] or Intel QuickIA [28]. Though HMPs are an attractive option, they are seldom popular in cloud servers. Instead, heterogeneity in the cloud is more prevalent across servers at the cluster level. Exploiting this heterogeneity to improve energy efficiency of LC-Services was the goal of this chapter.

Some previous works by Delimitrou et.al [100, 74] have explored ways to exploit server heterogeneity. These works propose machine learning techniques to identify co-locatable workloads, and map them to servers on which they achieve high efficiency. But they do not provide a detailed picture about the server-level study of tail latency and energy efficiency in a heterogeneous cluster. My work dives deeper, to perform a generic study about the impact of capacity and efficiency heterogeneity existing across servers.

In this work, I also propose the use of a load distribution strategy suitable for heterogeneous clusters. Traditionally such tasks are performed by complex centralized [101] or de-centralized [101] load balancers in large cloud platforms. But they lack heterogeneity awareness which, as shown in this research, can reduce the application service energy footprint. CSPs employ complex large scale resource managers [54, 102, 32, 69, 38] to assign CPU and memory resources to cloud workloads. My work relies on such managers to provide compute resources, albeit in a heterogeneity-aware manner to assist in implementing the proposed E-MT strategy in the control plane. In my research, given a set of compute resources, I try to exploit their heterogeneity. However greater gains could be achieved by integrating the E-MT strategy into existing resource managers to exploit the large scale of heterogeneity available on a cloud warehouse. Research works such as [103, 104, 44] try to improve resource utilization using job packing to reduce unallocated resources on heterogeneous clusters. However, unlike them, I do not attempt to improve utilization using co-location techniques. Such works are orthogonal to mine, since my work attempts to improve utilization of core resources already allocated to the LC-Service.

3.8 Conclusion

In this chapter, I considered two types of core heterogeneity - capacity and energy efficiency, and I performed a detailed study of their impact on server utilization and energy usage under different load distribution strategies. I proposed a heterogeneity-aware control plane strategy that accounts for heterogeneity in both capacity and energy efficiency. Using a model-based and a simulation-based analysis I demonstrated the effectiveness of this strategy to achieve both a high server and cluster utilization, while still meeting the SLO. I also discussed the challenges associated with implementing the proposed E-MT scaling and load balancing heuristics on a large-scale cluster. This work provides the required foundation and motivation to perform an experimental analysis on large-scale cloud platforms.

Chapter 4

Server-Level Control Plane Strategies to Exploit Heterogeneity

4.1 Background

With evolution of software architectures for cloud-hosted applications from traditional monolithic to multi-tier applications to microservices, the fundamental units of deployment have been gradually shrinking from seconds to microsecond-order service times. To effectively meet the milliseconds order response latency requirements of end-users, it is critical to achieve predictable latency bound for the deployment units (or the LC-Service instances in case of microservices). To achieve predictability for such low latency service tasks, a finer look into the server-level latency sources for the LC-Services is required. Thus, in this chapter, I further refined my objectives for LC-Services beyond the Tail-Energy challenge to also include tail predictability. The goal is to service LC requests at the lowest energy cost with a predictable tail latency that meets the tail latency SLO requirements. Upon further review of microservices, I inferred that the tail latency problem comprises of three inter-twined sub-problems: 1) meeting tail latency targets, 2) reducing tail latency variability, and 3) improving energy efficiency of LC-Service hosts.

To meet the above goal, it is important to quantify the contributions of various factors that impact

the tail latency characteristics. As shown in prior studies [30], most factors that impact the tail latency characteristics are node-configuration and/or workload-characteristics dependent and could be optimized at application-level for an LC-Service. However, interrupts-induced delays and queuing delays are dependent on incoming request traffic and require dynamic configuration of network processing to optimize tail latency performance. Thus, in this chapter, I first quantify the impact of interrupts and queuing on the tail latency performance for various workloads and network stack processing configurations. Based on the results, I propose a server-level runtime system that leverages directed and controlled exploitation of Core-frequency Heterogeneity (i.e. frequency scaling) to mitigate the three-pronged tail latency problem.

The contributions of this work are as follows:

- I have analyzed the performance of several end-system network processing configurations for Network Interface Card (NIC) drivers, with an objective to reduce the tail latency problems that are caused by interrupts. I show that a single NIC rx-queue using centralized IRQ processing is the most suitable configuration for most request loads. I also show that using a dedicated frequency-scaled core for IRQ processing can mitigate the interrupt-induced tail latency problems manifesting at high request loads for LC-Services.

- I have identified the receive socket buffer queue as the primary source of queuing delay in the Linux kernel. I propose a dynamic frequency scaling strategy based on instantaneous buffer queue length to achieve an optimal, energy efficient and SLO-guaranteeing service rate for the application cores.

- I have implemented a runtime system, that performs dynamic resource allocation and configuration to adapt to request traffic rates while minimizing the tail latency problems caused by interrupts and queuing delays. This system reduces interrupt-driven tail latency variability by up to 86% and achieves up to 16% energy saving for a variety of LC-Services.

4.2 Performance and Energy Efficiency Objectives

As mentioned in the previous section, achieving the performance and energy objectives of LC-Services requires addressing three inter-twined sub-problems: meeting tail latency targets, reducing tail latency variability, and improving the energy efficiency of LC-Service hosts. While two of these issues have been previously discussed in Chapter 1, I briefly revisit them and describe the three sub-problems below.

4.2.1 Meeting Tail Latency Targets

Tail latency is defined as the higher range of latencies in a statistical analysis of response latencies for a large sequence of service requests. In a statistical distribution of response latencies, a 95^{th} percentile (or P95) tail latency refers to the value such that 95% of the request latencies are lower than it. In large data centers a 95 percentile (P95) to 99.9 percentile (P99.9) latency is considered a decent tail metric, which is expected to process/aggregate most of the responses. The primary causes of long task latency within a server have been identified as queuing delays, OS interference, background processes and CPU wakeup delays. The OS interference is attributed to network interrupts, task schedulers and system daemons. One or more of these factors can cause a few response latencies to be pushed into the tail zone. These delays are not node-specific and can probabilistically occur in any node, so generic system-level and dynamic request traffic-aware latency-reduction techniques need to be employed which can be applied to all the nodes to prevent unforeseen tail elongation beyond SLO cutoffs.

Interrupts and OS-queuing related issues are typically ignored due to their limited impact on performance in traditional cloud deployments. However, with micro-service execution times shrinking towards the sub-millisecond range and the increase in traffic loads per service instance, these factors can become significant and need further attention.

4.2.2 Reducing Tail Latency Variability

Tail latency can vary across multiple time slots while being subjected to the same request loads. The source of tail latency variability is the unpredictability in the extent to which various latency-causing factors impact the request latency. High variability prevents administrators and developers from guaranteeing response time to the end-user. Variability can also lead to an increased number of Hedged requests [6]. A Hedged request is a redundant request sent out in modern cloud frameworks to alternate servers whenever the response for the original request is delayed beyond a certain threshold. Variability can cause Hedged requests to be sent out unnecessarily to mirror servers, leading to wasted energy and computing resource. Reducing this variability will help insure a predictable tail latency, as well as provide energy saving opportunities by allowing execution to be performed at a lower speed while still guaranteeing SLO.

4.2.3 Improving Energy Efficiency of LC-Service Hosts

Energy efficiency is defined as the number of instructions retired (or number of requests processed by an LC-Service) per unit energy consumed by the server. But achieving optimal energy efficiency for LC-Service is quite challenging. Lack of energy proportionality in traditional servers lead to lower efficiency in under-utilized servers [12]. Whereas, over-utilized [30] or power-capped servers [105] risk SLO violation. Dynamic Voltage Frequency Scaling (DVFS) [17, 14, 24, 25] and opportunistic background application consolidation [26, 25] have been proposed to improve energy efficiency; however next-generation intra-cloud networks supporting traffic beyond 40Gbps are expected to witness very high request loads at servers. Additionally, previously ignored low-latency Hardware/OS delays will constitute a significant fraction of server latency for the next-generation of micro-services. Thus, energy saving options for situations where high traffic loads cause higher interrupt loads and OS queuing delays need to be explored.

4.3 Sources of Tail Latency

Of the various sources that contribute to tail latency, interrupt and OS queuing are dependent on the request traffic and independent of the application implementation, and hence are of primary concern in this work.

4.3.1 Interrupt Processing

In a typical end-server system, request packets for LC-Services are received at the NIC hardware buffer and then transferred into one of the FIFO receive ring buffers (Rx-buffer) using a DMA engine. NIC Receive Side Scaling (RSS) maps each NIC ring buffer to a set of cores. Upon a hardware interrupt being generated due to an incoming request, the HardIRQ (and subsequently softIRQ) handler is scheduled on the interrupted core, preempting any LC-task executing on the core; the LC task is put in a wait queue until the protocol processing completes. This high priority interrupt results not only in higher service latency but also higher variability in the service latency. These are both exacerbated during periods of high load and/or bursty arrivals. In some experiments I observed as much as 50% variation in P99.9 for Memcached service latency over multiple runs using same request rates. Such a large variation in the tail latency greatly impacts the predictability of server tail latency. Boosting the core frequency increases the service rate and reduces latency [26, 14]. However the speed up is required for only the interrupt context execution and not the application context execution. Needlessly speeding up application context execution consumes higher energy without additional SLO guaranteeing benefits.

4.3.2 Packet Queuing

As shown in Figure 4.1, the request and response packets traverse through multiple queues in the cloud server node. The NIC ring buffers and hardware queues contribute little to the latency since they are processed quickly. The transmit socket buffer queue is a transit queue used for copying response packets to kernel memory. Queuing Discipline (QDisc) Queues are policy controlled and

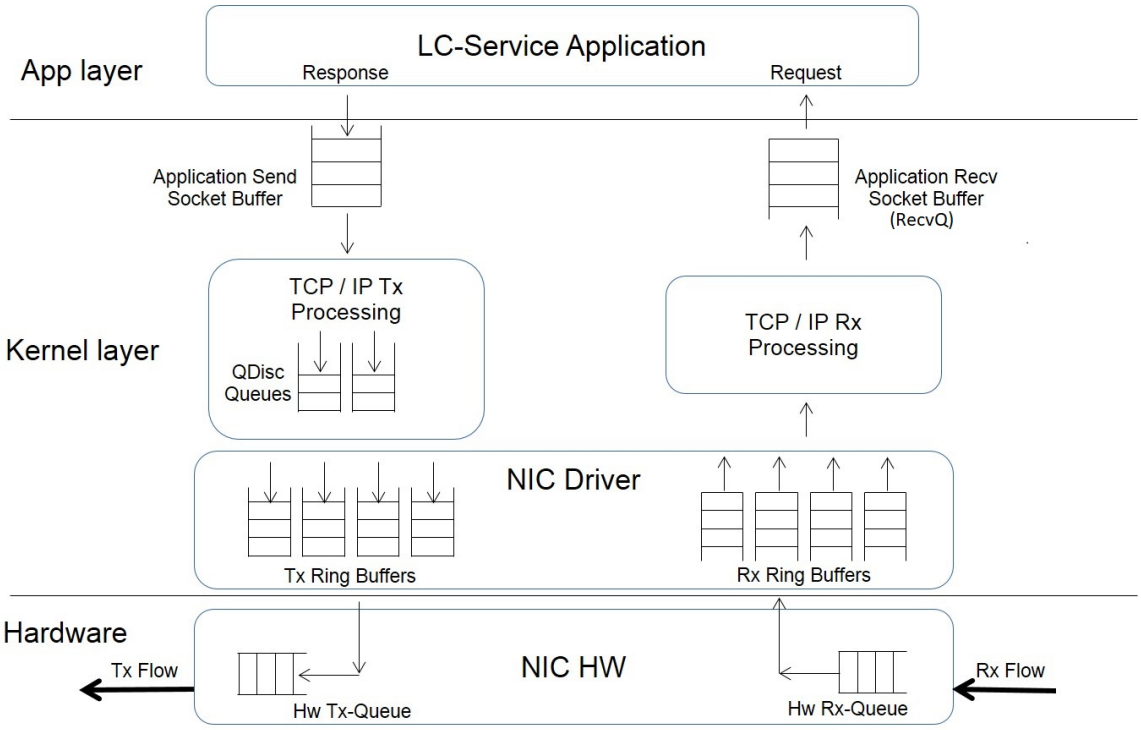


Figure 4.1: Request-response path within a cloud node. A query travels through multiple queues along the receive and transmit stack path. Queuing delays can lead to long tail latency.

configurable for low latency. My observations show that in fact the Application Receive Socket Buffer (or RecvQ) is the primary source of queuing delay in the request-response path. The RecvQ is the kernel-level buffer for the application socket where the incoming requests are queued after completing Rx stack processing. When request rates occasionally exceeds aggregate service rates, the RecvQ builds up. This means the request must wait in kernel buffers to be received by the LC-application, resulting in large delays. In past, researchers have proposed using DVFS [25, 33] to increase server efficiency by monitoring application-managed user level queues and speeding up the service threads only when SLO is threatened. In this chapter, I propose such a technique, but for kernel-level socket buffer queues which are inaccessible to application developers. Hence our proposed approach is not application-specific.

4.4 Proposed Approach and Methodology

4.4.1 Reducing Inter-Process Interference

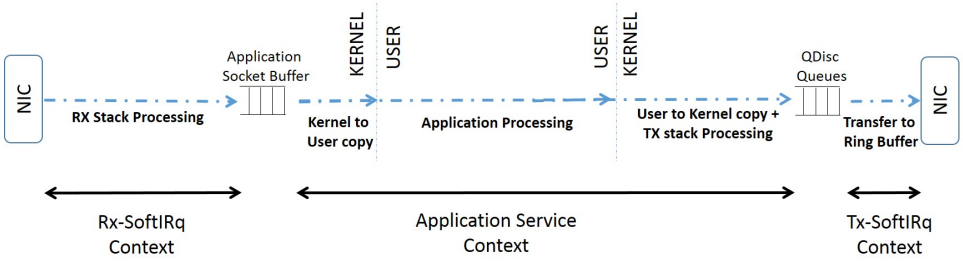


Figure 4.2: Request flow through multiple process contexts. Co-execution of Rx and Tx softIRQ and application contexts on the same core causes interference.

The request-response flow within a server passes through three process contexts: 1) the Rx-softIRQ kernel context, 2) the application context, and 3) the Tx-softIRQ kernel context. These are shown in Figure 4.2. Though all of the LC-Task processing occurs in a single process context (or container instance), the request-response path through the node stretches across multiple contexts. The Receive stack processing on the request packets is performed by the softIRQ handler kernel thread in a kernel process context, while the Transmit stack processing for the response payload is performed in a kernel context by a separate kernel thread.

Of the three contexts, the application context is the most time-consuming, usually requiring orders of magnitude (over 1000 times for long LC-tasks) longer than the Rx-softIRQ processing time. However, for short tasks (e.g., memcached application tasks), as the request rate increases the corresponding softIRQ processing requirements grow proportionally, eventually reaching significant levels on the application cores and thereby reducing the co-executing LC-task service rate. I propose to eliminate this inter-process interference problem by isolating the application thread from the receive and transmit softIRQ thread. I employ the following two techniques to achieve this isolation.

Interrupt Mapping: The Receive Side Scaling (RSS) enables interrupts associated with a NIC Rx-ring buffer queue to be distributed among the mapped CPUs specified by the SMP-affinity

bit-mask. By default, the interrupts are equally distributed across mapped cores. Since a target Rx-ring for an incoming packet is configurable based on packet identifiers, RSS must be applied to all Rx-rings to enable all traffic from various sources and destined for the LC-Service TCP port to be diverted to the mapped core. Alternatively, all incoming requests can be centralized into a single ring buffer queue and subsequently interrupt the mapped core. The RSS technique enables identifying the target Rx-buffer through an on-NIC hardware indirection-table using a hash-based filter on packet identifiers. Modern OS's support Receive Flow Steering (RFS), which enables softIRQ execution on the same core as application. Advanced NICs also support Accelerated Receive Flow Steering (AccRFS), which enables kernels to update the indirection-table with target cores running the consuming application thread. RFS techniques help reduce future cache miss delays by leveraging data locality in local CPU caches, but they also risk increasing the request-response delay due to context-switching as described in the previous subsection.

CPU Affinitization: CPU affinitization can be used to control the placement of tasks on a subset of cores. Unlike pinning, affinitization provides the scheduler with a list of cores to which a task may be scheduled (affinitized). This allows the scheduler to schedule tasks on other lightly loaded cores, if required, thereby improving core utilization. The technique is used to isolate or restrict some processes from interfering with some others. Also affinitizing multiple tasks in an architecture-aware manner, such as sharing a memory hierarchy level (L2, L3, NUMA), helps improve performance. Thus by scheduling softIRQ and LC-Service threads on cores sharing last-level caches, overall server latency can be reduced.

4.4.2 Exploiting Core-frequency Heterogeneity

DVFS has been widely used as a reactive strategy to achieve energy efficient SLO-guaranteed execution for varying request loads [14, 24, 25]. I observed that socket buffer queue lengths can be accessed during runtime to pro-actively scale service rates in response to request rate variations. In this chapter I present a dynamic energy efficient strategy for millisecond order LC-Service tasks potentially suffering from OS-queuing delays. During high loads and bursty phases, LC-task threads

can be run at high frequency to meet latency SLO; this enables the LC-task threads to consume the buffer queues quickly and reduce the response latency for tail-elongating back-of-queue requests. At lower loads, LC-task cores must proportionally slow down to lower frequency levels to reduce energy wastage. The LC-tasks can be then completed at lower power consumption while still meeting QoS deadlines.

It is worth noting that the server CPUs cannot usually enter energy-saving deep sleep states for two reasons. First, the wake-up delay from deep sleep states are on the order of 100s of microseconds and result in further delays due to cold cache misses. They have been shown to cause tail elongation in LC-Services [30]. Second, multi-core CPUs can only enter deep sleep states when all LLC-sharing cores are idle. But in a cloud server hosting several container instances, such low utilization is seldom observed, resulting in continuous idle power consumption. For the same reason, Race-to-Sleep strategies would not be suitable for LC-Services. However, since current CPU hardware require only 10s of microseconds to update a core frequency, implementing a queue-length dependent scaling strategy for short sub-millisecond order LC-tasks (or microservices) is a feasible approach.

4.4.3 Testing Methodology

4.4.3.1 Benchmarks

I used six (6) applications from the TailBench suite [106] for my study of OS queuing overhead. These applications, with open-loop request-response characteristics, represent the compute-intensive and On-Line Data-Intensive (OLDI) data center workloads. These applications have average service latency ranging from 100 microseconds to 500 milliseconds. To study the impact of OS queueing at high request loads, I employed multiple clients and increased the request rates for each TailBench application service instance to levels on par with the maximum aggregate service rates that can be supported by server cores. Due to the relatively low service rates of the order of 100s - 1000s of requests per second and long service times, interrupts are unlikely to affect tail latency of these applications. This way only the queuing related overhead would be observed for the applications.

To study the impact of interrupts, I tested two closed-loop low-latency applications, Memcached (memc) and CustomRPCServer (rpc). Memcached is a general-purpose distributed memory caching system that has been extensively used big data operators to speed up data lookup. It maintains an in-memory key-value store and efficiently retrieves data values upon a lookup request. The Memcached server is designed and implemented as a pool of worker threads with event-driven data receive and send. Memcached is run with numerous client threads and few server threads.

The customRPCServer is a version of Remote Procedure Call (RPC) Server application that is widely used by websites to serve client requests. It receives a connection request from a remote client, establishes connection, and then continuously services requests arriving from clients and responds back. My customized version, customRPCServer, receives a request, performs a dummy processing task on a server CPU and then responds back a standard response to the client. CustomRPCServer has a one-to-one connection between each thread on the client and a corresponding thread on the server. The request and the response sizes were set to 256 bytes. Both these applications have a median latency on the order of 10s of microseconds, so they can sustain a very high query rate and are ideal for studying the impact of interrupts on the tail latency.

4.4.3.2 Node-level Timing

Client side application latency results have been shown to be biased by network stack and queuing delays [107] at the client side. In my experimental infrastructure too, the client application and TCP buffers became the bottleneck for many applications and introduced client side latency into the LC-request-response path. To correctly identify the sources of bottlenecks in the cloud server nodes, I collected the latency results at 3 layers for the closed-loop testing setup using the customRPCServer. The customRPCServer request packets had placeholders for collecting timing information along the request-response path. I inserted netfilter kernel hooks at the IP layer on both client and server side to timestamp every ingressing and egressing packet on the client and server side at the Layer2-Layer3 kernel interface. The closed-loop system enables correctly timestamping a request and response packet based on the unique client side port number, without worrying about packet reordering.

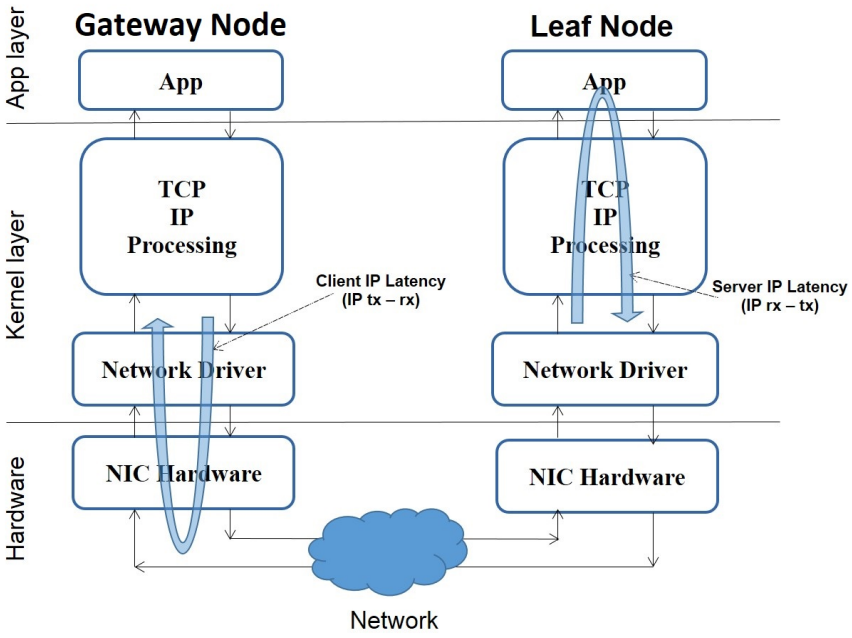


Figure 4.3: Test platform configuration to collect server node-side latency statistics. Gateway node (Client) IP Latency measures the complete server processing time including NIC and transmission latency. I use IP latency for my study to exclude Gateway node's application-layer latency, while including the full stack latency for Leaf node (Server).

Response latency for a request is calculated in the client thread using the timestamp information inserted into the response payload along the query path. Server-IP-latency (shown in Figure. 4.3) includes most of the software level delays incurred by the request in the network stack along with the LC-task application time. In addition, Client-IP-latency also includes the network transmission and NIC delays on both sides. Finally, App-timing is the latency that the client perceives. My experiments showed that the difference between the Client-IP-latency and the Server-IP-latency was small (100s of nanoseconds) and stable, regardless of the request rates. Hence, for our work, I chose to use the server IP layer latency extracted from response packets at the client application layer. Tail latency is then statistically calculated by the client over 1000s of consecutive response latencies collected over a fixed time interval. It is worth noting that tail latency in commercial data centers is measured at the client side running on a gateway node. Thus, latency may be introduced both at gateway and server nodes. But in this work, I limit the scope to server node latency.

4.4.3.3 Test Platform Configuration

For the server node I used a Dell E370 server with two quad-core Haswell CPU nodes running the Linux 3.19 kernel. The server nodes were connected by 40 Gbps ConnectX-3 Pro Mellanox NICs, for which driver-driven polling (NAPI) was enabled. The clients were run on dedicated cores running at 3.5 GHz across multiple networked machines. The number of clients used were varied depending on the desired workload at the server node. I used Running Average Power Limit (RAPL) counters available on Haswell CPUs to obtain CPU energy usage. I consider only CPU energy since CPUs consume the majority of the power in highly energy-proportional cloud rack servers[15]. Power consumption by memory and other peripheral components are beyond the scope of this work. I configure the server core frequency using the Linux Userspace frequency governer.

4.5 Reducing Interrupt Side-effects

To test the impact of interrupts on tail latency, I used memcached and customRPCserver which can handle query rates on the order of 100K Queries per Second (QPS) on a single core. To eliminate the impact of queuing delays, closed loop clients were used for request generation. For my tests, 6 client threads were executed. The findings of my study are discussed in the following subsections.

Table 4.1: Server configurations for 4 memcached threads on an 8 core server node. All application cores operate at 1.2 GHz.

Configuration	**RFS off**	**RFS on**	**Details**
Distributed-IRQ (D-IRQ)	C1	C2	There are 8 rx-rings. Each IRQ is mapped to a unique core
Mapped-IRQ	C3	C4	There are 8 rx-rings. All IRQs are mapped to a single isolated IRQ-core
Centralized-IRQ (C-IRQ)	C5	C6	There is a single rx-ring. All IRQs are mapped to a single isolated IRQ-core
Centralized-IRQ with frequency scaling	C7	C8	There is a single rx-ring. All IRQs are mapped to a single isolated IRQ-core running in turbo-boost mode

4.5.1 Identifying Interrupt-friendly Configurations

To identify configurations that best mitigated the impact of interrupts, I executed Memcached on several server configurations. In Figure 4.4 - 4.7, I present the results for the eight (8) configurations listed in Table 4.1. There are four (4) configurations corresponding to different IRQ-mapping; each of them is set with Request Flow Steering (RFS) on or off. RFS is enabled by default on most systems and causes part of the SoftIRQ processing to be performed on the application core. As a result, the network stack processing is performed partly on the interrupted core in a SoftIRQ context, and partly by a kernel thread on the application core in the kernel context. Setting RFS off for each of the 4 IRQ configurations ensures SoftIRQ processing is performed entirely on the interrupted core.

From my results I observed that, in general, when RFS is enabled, the softIRQ processing load is reduced (Figure 4.5), the interrupt count increases (Figure 4.4), but the throughput reduces (Figure 4.6). The reduction in throughput leads to reduced tail latency (Figure 4.6) and lower energy efficiency (Figure 4.7). The RFS-enabled configurations were also observed to incur much higher context switches and application thread migration between the 8 cores and across the two CPUs. Hence, as shown in Figure 4.5, the 4 memcached threads mostly executed on the same CPU (on Cores 0, 2, 4 and 6) for the non-RFS configurations (C3, C5 and C7). Whereas, the RFS-enabled configurations (C4, C6 and C8) had the application threads distributed across both CPUs. These stalls resulted in lower utilization (Figure 4.5) and lower throughput or QPS (Figure 4.6) being achieved for RFS-enabled configurations.

Problems with Distributed-IRQ Configuration: Configuration C1 performs the softIRQ processing for packets on the same core as the ring-buffer queue which received the packet. I observed that softIRQ processing overhead (shown as green bars in Figure 4.5) being higher on the cores that receive higher number of interrupts (shown in Figure 4.4). In my experiments, the interrupt distribution across cores was relatively imbalanced in C2, as compared to C1 (Figure 4.4). This caused an adverse impact on the tail latency of LC-tasks (Figure 4.6). But in general, for the D-IRQ configurations (C1 and C2), since the 4 service threads are not pinned to any core, they

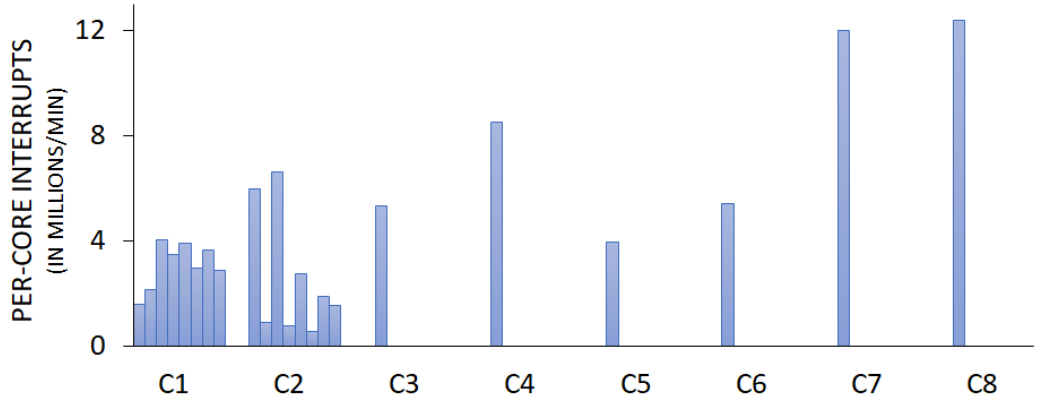


Figure 4.4: Interrupt distribution for 4 memcached task threads on 8 cores for the 8 configurations shown in Table 4.1. Distributed IRQs (C1 and C2) cause interrupts to be received on all 8 cores (Core 0 through Core 7). Centralizing the interrupts (C3 through C8) causes interrupt handling on a single core (Core 1). Due to IRQ coalescing, the number of interrupts received need not be proportional to the request rate

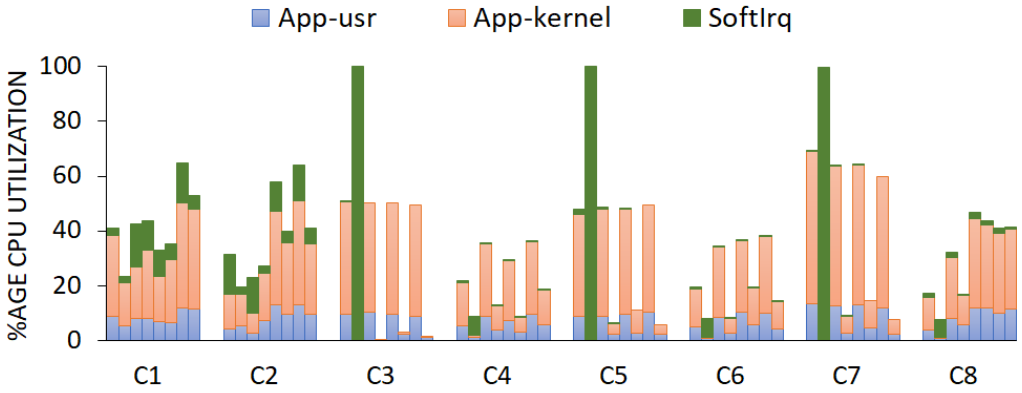


Figure 4.5: CPU Utilization breakdown for 4 memcached task threads on 8 cores for 8 configurations. Unlike D-IRQ configurations (C1 and C2), C3 through C8 perform softIRQ processing primarily on the interrupted core Core 1. D-IRQ configurations see CPU utilization being spread across all 8 cores due to frequent interrupt-induced context switching and task task migrations. In contrast, the configurations C3 through C8 observe the task threads mostly run on the non-interrupted CPU, i.e., CPU 0 (cores 0, 2, 4, 6). A 100% core utilization for IRQ-core indicates the softIRQ processing becoming the bottleneck for the non-D-IRQ configurations. By scaling up IRQ-core frequency to turbo-boost mode (in C7) application core utilization could be improved from the C-IRQ configuration C5. since memcached tasks have short service times, a small fraction of the thread execution time is spent on the application level execution, whereas majority of the processing occurs at the kernel level as transmit-side processing.

may migrate to other available cores upon being interrupted. This can be verified from the core utilization that is distributed across all cores for C1 and C2, as shown in Figure 4.5. This negatively impacts data locality, exerts higher pressure on the memory hierarchy and hence results in lower

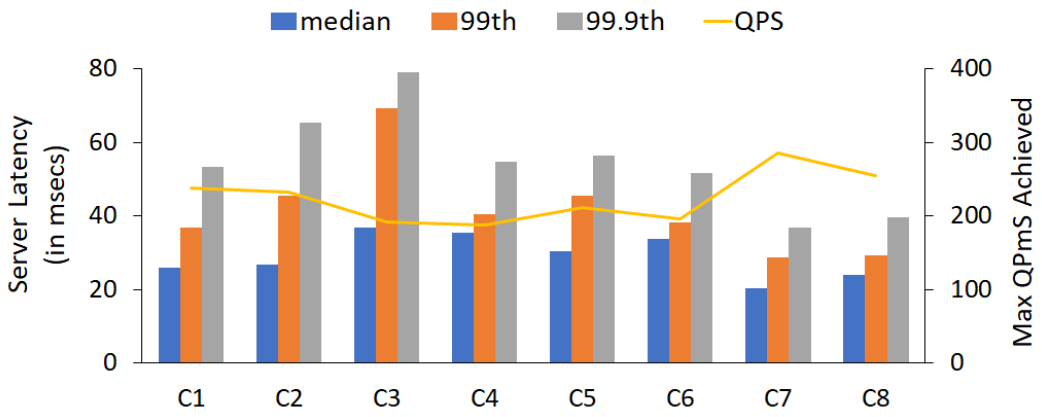


Figure 4.6: Server latency and maximum QPS supported by memcached for the 8 configurations. D-IRQ configurations (C1 and C2) observe a fair amount of latency variability, i.e., a 99.9 %ile tail latency that is twice that of the median latency. C3 and C4 face higher median and tail latency because of softIRQ processing for 8 rx-rings becoming a bottleneck on a single IRQ-core, thus achieving lowest QPS. The throughout slightly increases and latency reduces for C5 by using a single Rx-ring for all requests. But increased thread migration caused increased median latency and lower throughput for the RFS-enabled C6 configuration. Speeding up a single IRQ-core showed increased throughput with C7 achieving higher QPS than the D-IRQ counterpart C1. Additionally, C7 observed reduced tail latency and reduced latency variability compared to C1.

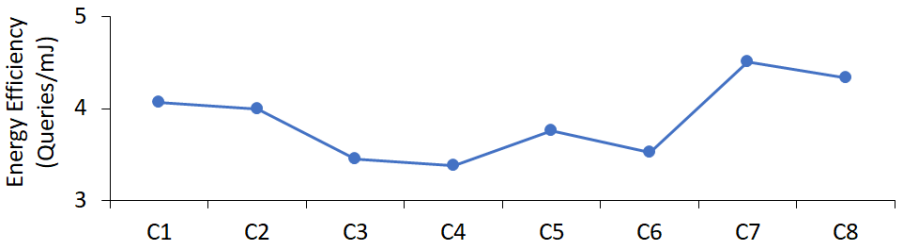


Figure 4.7: Maximum CPU energy efficiency (queries processed per unit CPU energy) achieved for memcached in 8 configurations. C7 could achieve highest efficiency by registering high throughput while speeding up a single CPU core.

than optimal energy efficiency (Figure 4.7). I also observed that the 99.9th %ile latency was more than twice the median latency for the D-IRQ configurations (Figure 4.5), thereby indicating some latency variability. This is primarily due to the unpredictability in interrupt distribution across cores. From this observation I inferred that, to improve the predictability of latency, i.e., to get the tail latency closer to the median latency, interrupt distribution must be centralized.

Note that the latency variability effect gets worse on open-loop systems, such as in a real microservice instance hosted on a cloud server, due to two reasons. First, due to arbitrary and bursty traffic, the instantaneous interrupt rate could be much higher causing more context switches on all cores

for a D-IRQ configuration. Second, higher traffic rate would require longer softIRQ processing times. Since softIRQs have higher priority, LC-tasks would have to wait longer to be scheduled for further execution, thereby making the tail longer and more unpredictable.

Another key observation was that the aggregate number of interrupts received across all cores are much lower for non D-IRQ configurations (C3 through C8 in Figure 4.4). This is because NAPI polling is activated upon an interrupt and remains active for a fixed timeout window after the latest packet is received. With all packets arriving to a single Rx-ring, NAPI polling would incur fewer timeouts. This causes fewer interrupts and subsequently fewer LC-Task context switches. On the other hand, the D-IRQ configurations receive relatively fewer packets at every Rx-ring, resulting in more frequent NAPI timeouts and thus a higher number of aggregate number of interrupts. Note that since NAPI is enabled, at very high loads very few interrupts may be observed, leading to fewer context switches. But such high loads could also be beyond the service capacity of nodes, thereby causing SLO violations.

Diverting all Interrupts to a Single Core: To reduce the impact of interrupts, I mapped all 8 interrupts to a single dedicated core (referred to as the IRQ-core) which is not assigned any LC-task threads. These are configurations labelled C3 and C4 as shown in Figures 4.4 - 4.7. Although this successfully isolated the interrupts from the applications (Figure 4.5), the IRQ-core was overwhelmed with servicing the interrupts and became a bottleneck. In addition, interrupts due to requests received in different queues have different IRQ numbers and so could not be coalesced (despite being destined for the same core). This resulted in a higher softIRQ processing delay and higher tail latency compared to corresponding D-IRQ configurations. RFS (in C4) reduced the load of softIRQ processing thereby reducing the high 99th %ile tail latency observed for C3 (Figure 4.6). However, compared to C3, a higher interrupt count for C4, thread migrations and subsequent resource stalls caused the median latency to stay similar for C3 and C4. However due to under-utilization of cores (Figure 4.5), C4 achieved the least QPS (Figure 4.6) and energy efficiency (Figure 4.7) amongst all configurations.

Using a Single Rx-ring Buffer Queue: To eliminate the drawback of configurations C3 and C4, I centralized all incoming requests to a single NIC Receive queue (Rx-ring buffer). These are labelled configurations C5 and C6 (RFS-enabled). I mapped the corresponding IRQ to a single IRQ-core. This helped significantly reduce the interrupts for C5 (compared to C3) through coalescing (Figure 4.4), while marginally increasing the maximum throughput (or QPS) of C5 as compared to C3 (shown in Figure 4.6). A similar improvement is achieved for C6 when compared to C4. Though C-IRQ achieved minor throughput improvement over Mapped-IRQ, it significantly improved the tail latency and reduced the latency variability. For instance, the difference in latency between the median and 99.9th %ile significantly reduced for C5, compared to C3 (Figure 4.6). The tail latency too significantly reduced. However, compared to the D-IRQ configuration C1, C5 only achieved at par variability (Figure 4.6), slightly higher median and tail latency, lower throughput and lower efficiency (Figure 4.7). However, on the bright side, the IRQ-core appears to be the bottleneck that restricts the performance of C-IRQ configurations.

Reducing IRQ-core Bottleneck: The IRQ-core (Core 1) for C5, shown as the green bar in Figure 4.5, runs at 100% CPU utilization for C5. To remove this bottleneck, I created two new configurations, C7 and C8, in which I turbo-boosted the IRQ-core frequency. This greatly increased the CPU utilization by memcached task threads for C7 (running on all cores except Core 1) as compared to C5 (shown in Figure 4.5). The reduced median latency (Figure 4.6), translates to significantly higher energy efficiency for C7 and C8, as compared to C5 and C6 respectively (Figure 4.7). Additionally, both C7 and C8 observe significant reduction in latency variability compared to other configurations. Thus, compared to D-IRQ configuration C1, the C-IRQ configuration with a scaled-up IRQ-core as in C7, could achieve lower median and tail latency(Figure 4.6), lower variability in latency (Figure 4.6), higher App-core utilization (Figure 4.5), higher throughput (Figure 4.6) and higher energy efficiency (Figure 4.7). Boosting a single IRQ-core could achieve a low and stable response latency. But the IRQ-core can still become the bottleneck despite supporting higher QPS rates. For instance in figure 4.5, the IRQ-core still runs at 100% utilization while the App-cores remain under-utilized. Thus a higher QPS rate would either require multiple IRQ-cores or D-IRQ configurations with all cores frequency scaled to meet the tail latency SLOs.

Using Multiple IRQ Cores: To further reduce the IRQ-core bottleneck under high request loads, I dedicated 2 cores to IRQ processing through NIC rx-queues. For this configuration, application threads were affinitized to non-IRQ-cores. In the following subsections, I demonstrate the effectiveness of this approach. However, C-IRQ configurations with 2 IRQ-cores can be best utilized when the interrupts are equally distributed. An skewed interrupt distribution between IRQ-cores can lead to lower energy efficiency and the reappearance of softIRQ-bottleneck problems. For larger many-core systems, even more cores can be dedicated to handling interrupts to serve higher request rates, which helps in further reducing the tail (Figure 4.8). But when the IRQ-cores are running at a higher (boosted) frequency, App-cores running at the base frequency (1.2 GHz) start becoming bottlenecks leaving the IRQ-cores underutilized. This explains the dip in energy efficiency for higher IRQ-core frequencies, as observed later in Figure 4.9.

4.5.2 Reducing Tail Latency Variability using C-IRQ

Like memcached, I also executed 6 customRPCserver service threads (described in section 4.4.3.1) for three configurations: C1, C7 and a third configuration C7-dualIRQcore, which is similar to C7 but employs 2 IRQ-cores. For all three configurations, I collected latency statistics for 100 execution instances, each of 100 seconds. Figure 4.8 shows box whisker plot for the variation in P99.9 latency over the 100 runs. Since tail latency is more sensitive to latency variations, minimizing the variability of tail latency would increase the predictability of tail latency as well as median latency. Thus, in this experiment, I study the impact of the 3 configurations on latency variability at different core frequencies.

The figure shows that the Distributed IRQ configuration C1 (shown as the red box plot) causes wild variations in the tail latency. The variability is reduced by increasing the CPU-core frequency of all App-cores. At higher core speeds, the softIRQ processing for interrupts is completed faster. But setting all application cores to a high frequency is wasteful. Executing few cores at higher frequency will still cause large variations in the tail latency due to the slow running cores.

On the contrary, the C-IRQ approach in configuration C7 achieved comparable but very stable tail

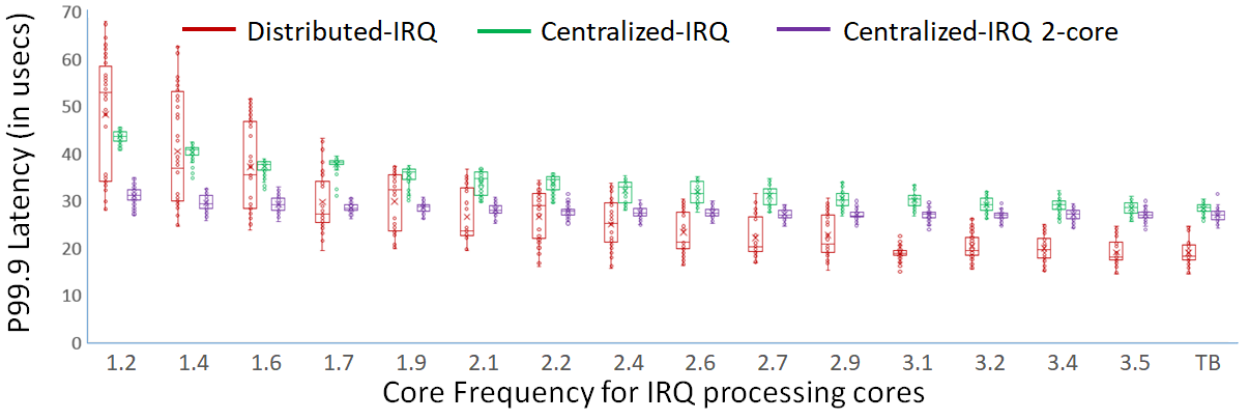


Figure 4.8: Variation in P99.9 latency of customRPCserver for D-IRQ and C-IRQ configurations at various core frequencies. Each box plot represents the P99.9 latency value for 100 test runs, each running for 100 seconds. Longer the box plot higher the variability in latency and greater the unpredictability. The variability is shown to reduce with Centralized IRQ processing. Comparable latency is achieved in centralized configuration by boosting only the IRQ handling core compared to the distributed configuration where all cores are simultaneously boosted. Using 2 IRQ-cores further reduces the median latency at lower frequencies when a single IRQ-core becomes a bottleneck.

latency with minimal variability. As observed before, IRQ-core can become a bottleneck at lower frequencies. By boosting the frequency of a single IRQ-core, while running the App-cores at the lowest frequency, I could observe the reduction in tail latency for C7. C7-dualIRQcore achieved a similar low variability while further reducing tail latency at low frequency levels. The sharing of softIRQ processing load between the 2 IRQ-cores enabled this reduction in tail latency. The increase in frequency of the IRQ-cores results in only minor reduction in variability. Unlike our closed-loop test, on an open loop system with higher loads, D-IRQ would introduce even higher variability while C-IRQ would suffer longer average latency. There a C7-dualIRQcore configuration would be able to further share the softIRQ processing load, observe higher IRQ-core utilization and further reduce tail latency.

4.5.3 Improving Energy Efficiency with C-IRQ

To study the energy benefits of C-IRQ, I measured the average energy consumption per user request/query for the three configurations C1 (Distributed-IRQ), C7 (Centralized-IRQ) and C7-

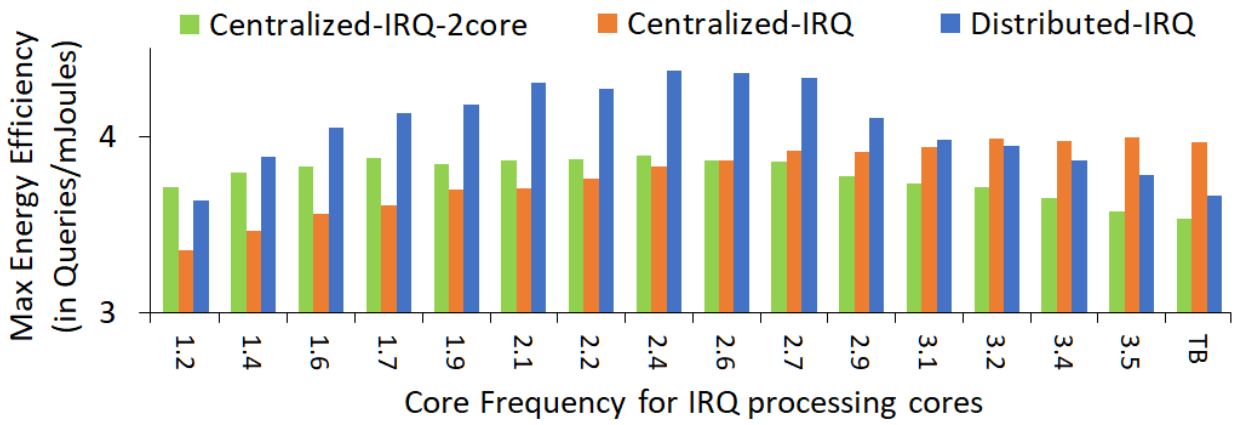


Figure 4.9: Maximum energy efficiency achieved (queries processed per unit CPU energy) for customRPCserver by C1, C7 and C7-dualIRQcore at different frequency levels. The C-IRQ configurations have all server threads running at 1.2 GHz, with only the IRQ-cores being frequency-scaled. The D-IRQ configuration has all server threads frequency-scaled. The frequency on x-axis refers to the frequency at which all frequency-scaled cores of a configurations ran. C7 achieves higher energy efficiency than C1 at higher frequencies as the IRQ-core bottleneck reduces. Efficiency of C1 increases initially with frequency as the throughput increases, but falls at higher frequencies due to the higher energy overheads. The C7-dualIRQcore configuration achieves better efficiency than C1 at lowest frequency because of its 2 IRQ-cores. At higher frequencies the C7-dualIRQcore becomes the most inefficient configuration due to higher energy consumption of IRQ-cores.

dualIRQcore (Centralized-IRQ-2core). All LC-Service instance threads were frequency scaled for D-IRQ, whereas they were executing at lowest frequency (1.2 GHz) for C-IRQ configurations (C7 and C7-dualIRQcore) and only their IRQ-cores were frequency scaled. Figure 4.9 shows the energy efficiency achieved by the 3 configurations as the scaling frequency was varied. C7 is shown to suffer from relatively lower energy efficiency for lower IRQ-core frequencies due to the IRQ-core bottleneck (seen in figure 4.5). The energy efficiency of C7 increases as the frequency of IRQ-core is increased. On the contrary, when frequency of all App-cores in case of D-IRQ are gradually increased, efficiency increases quickly until an optimal intermediate frequency range and then falls below C7's efficiency for higher values since power consumption varies super-linearly with core frequency [108].

C7-dualIRQcore could improve upon C7's energy efficiency and is even better than C1 at lowest frequency because the 2 IRQ-cores shared the softIRQ processing overhead to eliminate the bottleneck, thereby achieving lower tail latency (as shown in figure 4.9). However with increase in IRQ-core frequency, no significant reduction in tail latency (figure 4.9)) with increase in IRQ-core

power consumption leads to only minor efficiency improvement, quite lower than that of C1. At higher frequencies beyond 2.7 GHz, C7-dualIRQcore efficiency starts degrading due to sharply increasing IRQ-core power consumption. Since C7 achieves similar latency (and hence throughput) at higher frequency levels, it is observed to be gaining higher efficiency compared to C7-dualIRQcore as the IRQ-core approaches the maximum frequency levels. Note that figure 4.9 demonstrates the maximum energy efficiency at a given frequency by considering maximum QPS achieved by a configuration. Unlike the best case results shown in Fig. 4.9, during non-bursty phases the energy efficiency is much worse for C1, whereas its only slightly lower for C-IRQ since the App-cores continue to execute at lowest core frequencies.

4.5.4 Observations and Inferences

Clearly, C-IRQ with IRQ-core frequency adaptation can deal with the side-effects of interrupts. Depending on the application, the interrupt processing to service processing ratio may vary. To prevent either the LC-Service or the softIRQ from becoming a bottleneck during a higher request load, the CPU resources must be scaled for both. In a nutshell, the results show that when the interrupt load is low (i.e., query rates are low and/or application service time is large) then D-IRQ is preferable. But with a high interrupt load (a high query rate with short tasks) C-IRQ has significantly better performance.

The overall efficiency of a cloud server node for a given LC-Service is dependent on several dynamic factors such as IRQ-distribution, QPS rate, LC-task scheduling and LC-Service characteristics. Thus it would be impossible to achieve the optimal energy efficiency on such a system at runtime. However, the energy characteristics of C-IRQ do not suffer interrupt-unpredictability and can be used for simplistic and relatively accurate energy estimation.

4.6 Adapting to Queuing Delays

As discussed in Section 4.3, the request-response path involves queues both on the receive(Rx) and the transmit(Tx) stack. To identify the bottleneck queue, I collected the node-level timing (Section 4.4.3) for the Xapian service (a benchmark from Tailbench [106] suite) running at 1.2GHz while subjecting it to 3000 Queries per second (QPS). Xapian is a popular open-source search engine used in several websites and software frameworks for web-applications. As suggested by the authors in [106], the search index is built from a dump of Wikipedia's English version from July 2013. The query terms used in the client are chosen randomly, following a Zipfian distribution [109]. My observations and insights are described below.

4.6.1 Tail Latency Dependency on Queue Length

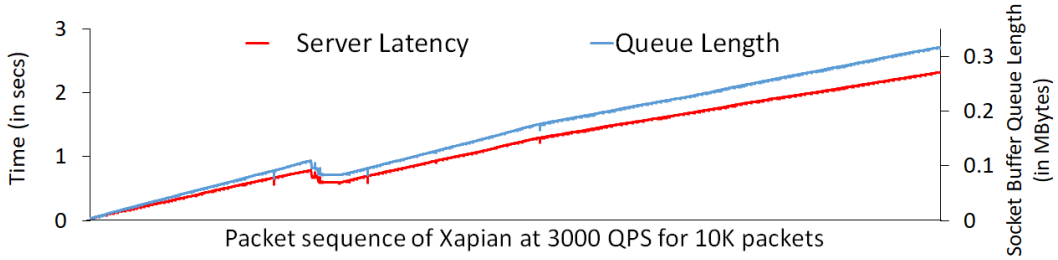


Figure 4.10: Receive socket buffer (RecvQ) length and server latency for 10000 requests to an Xapian service benchmark. The request rate is 3000 QPS and a fixed service rate with all App-cores running @1.2 GHz is used for 4 instances. The metrics collected for 10000 consecutive requests are plotted along the x-axis. The left y-axis represents the server latency for requests (plotted as the red line). The right y-axis represents the RecvQ length for requests (plotted as the blue line). The figure shows that since Rx-socket queue length is directly proportional to server latency, the instantaneous RecvQ length can be used as the sole indicator of expected response latency for an incoming request.

When I subjected 4 Xapian service threads to a request rate of 3000 QPS while maintaining the 4 App-cores at 1.2 GHz, I observed that the latency for each packet started increasing linearly. This is expected behavior for any server whose request rate exceeds the service rate. During a bursty high load period, a similar situation is caused wherein the request rate exceeds the service rate for a small time window. To identify where the increasing bottlenecks are, I measured the Rx and Tx path time along with application processing time for 10000 consecutive packets. I observed no

marked increase for application and transmit path time within the server node. The Rx path time seemed to be the sole contributor to increasing server latency.

Suspecting the socket buffer to be the source of latency I collected the per-request statistics for RecvQ i.e. I collected the instantaneous length of RecvQ at the time a request was being pushed into it after stack processing. I also collected the Server-IP-Latency (see Figure 4.4.3.2) timings for each request. The queue length information collected for 10000 consecutive packets (the blue line) is completely in proportion with the server latency measure for the corresponding requests (the red line) as shown in Figure 4.10. Thus the RecvQ queue length can be utilized as a sole indicator of expected queuing latency for a request. I use this important observation in my runtime environment to predict the growing latency during high load periods and accordingly adapt service rates through scaling techniques to avoid SLO violations.

An interesting observation in Figure 4.10 is the abrupt dip in latency and queue length during the request sequence. Upon collecting the network statistics for the period, I observed that the OS stack had to collapse and prune several Rx-socket buffered packets to prevent a buffer overflow in wake of the rapidly filling buffer. This causes a temporary push back on the request rate from the client, leading to queue build up on the client transmit side as well. The slower request rate could improve the server-side latency for some subsequent packets, but the multiplicative increase in the socket buffer window allowed the packet rate to resume and the latency to grow linearly again. Such instances of traffic backlog can be observed due to a long burst arrival or when load rates exceed the service rates of LC-Service, thereby threatening SLO violations. Thus service rates need to be increased in a timely fashion to avoid such situations.

4.6.2 Energy Efficient Service Rate Adaptation

To adapt to the high request rate for Xapian, I gradually increased the service rate by scaling the frequency of the cores executing the LC-tasks. The 4 task threads were affinitized to 4 cores. I started with the base configuration of all cores running at 1.2 GHz and scaled one affinitized core first to 2.4 GHz and then to 3.5 GHz as shown in Figure 4.11. The increased service rate

reduced the queue length and latency increase for the 10K packet sequence. I further increased the frequency for a second core up to 3.5 GHz to achieve a steady-state service rate that prevented RecvQ build-up.

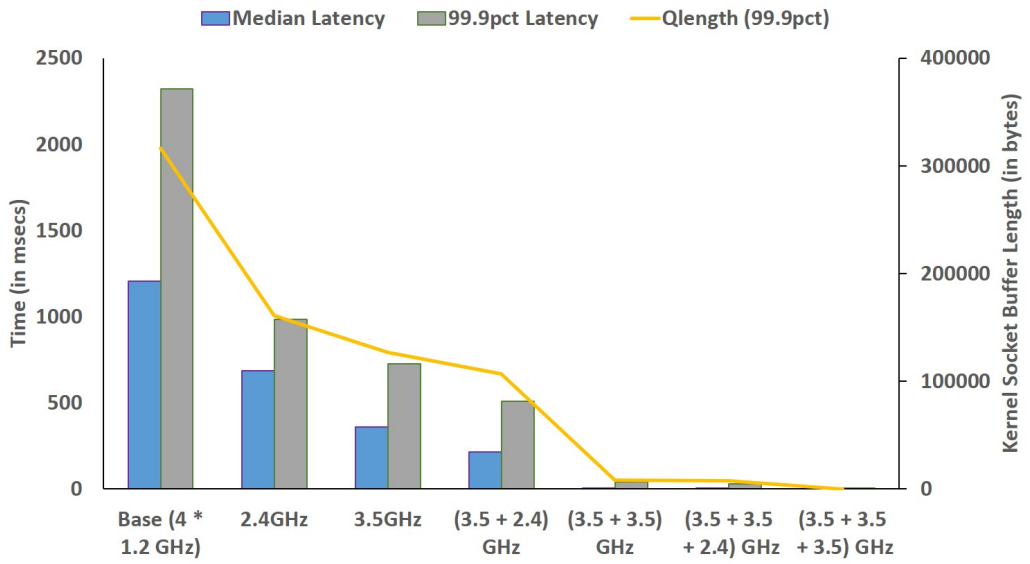


Figure 4.11: Rx Socket Buffer length (Qlength) and latency statistics for Xapian after a 10K request sequence of 3000 QPS for various server configurations. The left y-axis represents the response latency in milliseconds. The median and the P99.9 latency for the 10K requests are shown as bars. The right y-axis represents the P99.9 value of kernel socket buffer (Qlength) in bytes, for the 10K request sequence. The x-axis represents the server configurations. i.e., the core frequencies for the 4 task threads, running by default at a base service rate of 1.2 GHz. Only the frequencies of core(s) higher than the base frequency are shown for each server configuration. The median and the tail latency as well as the tail Qlength improve significantly upon gradually increasing the aggregate service rate. The configuration with minimum aggregate service rate which prevents queue buildup is the most energy efficient. From among the tested configurations, the server configuration with 2 cores @3.5 GHz and 2 cores @1.2 GHz achieves a service rate higher than the 3000 QPS arrival rate, thereby avoiding RecvQ build up.

For an energy efficient execution, service rates must be maintained at frequency levels that is just enough to prevent large queue build-up. Traditionally, CPU utilization has been used as an indicator of the service rate. This strategy tries to consume the queue quickly by running cores at high frequencies, ignoring the instantaneous queue length and/or burst phases. However, it would be more energy-efficient to run cores with service rates proportional to the Rx Socket Buffer queue length. This approach increases the average latency for the requests due to larger queue build-up. But if the service rates are updated in a timely fashion, the approach does not risk SLO violation. A conservative runtime frequency update strategy, as described in Section 4.7, can prevent SLO

violations as well as achieve better energy efficiency.

4.7 Runtime Manager

Using insights gained from experiments described in Sections 4.5 and 4.6, I set up a server-level control-plane in the form of system-level Runtime Manager. The Manager scales resources dynamically to reduce the tail latency problems, as shown in Figure 4.12. It employs Instance scaling and exploits Core-frequency heterogeneity. In this section, I describe its configurations and the various components.

4.7.1 Pre-configuration

The Runtime Manager executes at Linux real-time priority on a secondary IRQ core. For jobs with long service times (greater than 200 ms), I use D-IRQ as the effect of interrupts are negligible when the QPS is on order of 1000. I set the number of LC-Service instances to 4, which offers enough service capacity (with cores running at 3.5GHz) to meet the demands of long jobs. For short jobs, I employ C-IRQ. Since I use a single Rx-ring queue for C-IRQ, I set it to the highest buffer size to reduce the probability of buffer overflow during periods of high request rates and/or bursty arrivals. Cores that execute the LC-Service instance are configured to use the Userspace governor, while the IRQ-cores use an on-Demand governor, which automatically adjusts the IRQ-core frequency based on the CPU utilization allowing it to adapt to varying traffic rates and preventing packet drops in the Rx-ring buffer during periods of high request rates and/or bursts of arrivals.

The IRQ-cores share the Last Level Cache (LLC) with affinitized LC-Service instances, thereby improving data locality. A heuristically obtained *Low_Threshold* and *Critical_Threshold* level (represented in bytes) for queue length is set for every application through the Policy Manager (Figure 4.12). The *Low_Threshold* level represents the queue length at which I start performing service rate control via frequency scaling of application cores. This delayed queue handling, as shown in later subsections, helps increase energy efficiency. The *Critical_Threshold* level represents a crit-

ical limit that threatens SLO-violations, requiring all application cores to be switched to highest possible frequency immediately. Since the request packet sizes for different applications vary, the threshold values represent the number of requests that are queued.

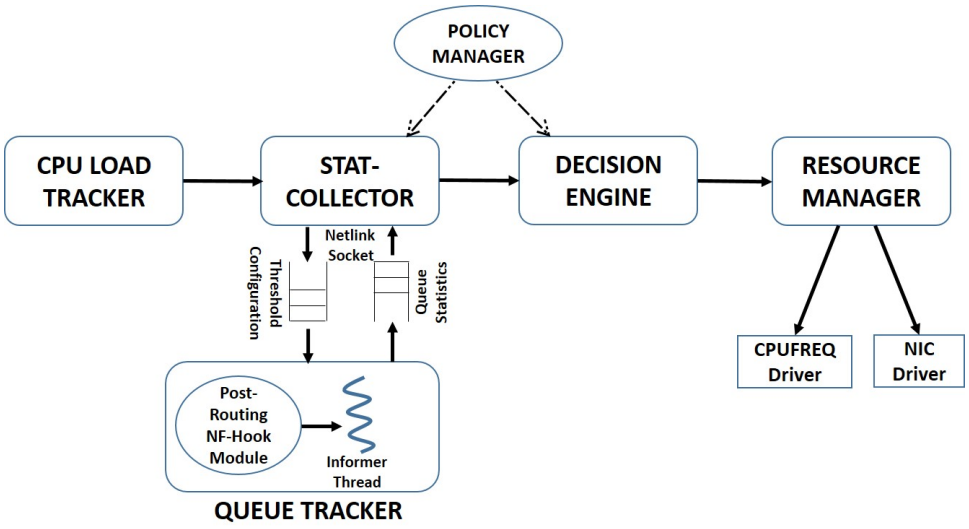


Figure 4.12: Server-level Runtime Manager for LC workloads to address interrupts and OS queuing related tail latency problems.

4.7.2 Collecting Runtime Statistics

The Stat-collector daemon, as showed in Figure 4.12, collects the system load information during runtime. The daemon receives statistics from two components: the CPU Load Tracker and the Queue Tracker. The CPU Load Tracker is a lightweight user-level routine which collects the CPU-utilization information from the /proc/stat interface for App-cores and IRQ-cores every 100 ms, and sends the utilization reports to the Stat-collector through a local socket. To receive messages in a timely fashion, and avoid scheduling delays, the Stat-collector and the Load-tracker are executed with Real-time priority on the secondary IRQ-core.

The Queue Tracker is a kernel module consisting of a netfilter hook routine (NF-routine) and an Informer thread. The routine tracks all incoming packets directed towards the LC-Service at the IP-TCP interface in the Post-routing stage. It checks the queue length by looking up the socket queue length for the packet's destination socket. When the LowThreshold mark for the queue is reached, the Informer thread is woken up, which then sends the queue length status to the Stat-

Collector daemon over a Netlink socket. The NF-routine is triggered upon every request packet arrival, but the queue-length check is performed over a Sensing-window (defined by the number of IP packets) to reduce processing overheads. A wide Sensing-window is used when the most recently measured buffer length is under the LowThreshold mark. This Queue Tracker phase, in which the queue length is low enough to not threaten SLO violations, is called the Lazy-Phase. The default Sensing-window width for the Lazy-Phase can be configured dynamically for different applications using the Policy Manager interface.

The selection of threshold values for different LC-Services requires careful consideration based on application characteristics. The queue length information obtained from the kernel represents the byte count, not the exact number of requests queued up. Depending on the LC-Service, the request size can vary from a few bytes to several kilobytes. Moreover, LC-Services may not always receive similar-sized requests. Depending on the nature of the request, each request can have different payload lengths too. Multiple requests could be merged by the OS stack into a single IP packet, causing further troubles with identifying the number of queued requests. Thus the various threshold values (in bytes) used by the Runtime Manager are set using an average request size unique to the specific LC-Service.

Upon exceeding LowThreshold, the NF-routine starts performing queue length checks more aggressively over a narrower Sensing-window (1/4th of the Lazy-Phase Sensing-window width). This Queue Tracker monitoring phase is the Active-phase, and is essential since I observed that the client-side network stack merges multiple requests during heavy traffic to reduce protocol-processing and buffer space overheads. Thus a single IP packet may contain multiple requests and can quickly increase the buffer queue length, even within a single Sensing-window width.

In Active-phase, all request packet payload lengths are summed over each Sensing-Window by the NF-routine. The Queue Tracker calculates the request rate for the Sensing-window based on average request size and infers the service rate for the window based on the queue length change. If the average request rate for 2 consecutive windows exceeds the service rate, the Informer thread sends the rates and the latest queue length to Stat-Collector. A request rate to service rate ratio called *growthRate* is calculated for the past 2 windows, and subsequently, whenever the calculated

growthRate exceeds the previously reported *growthRate* by 5%, the statistics are again sent to the Stat-collector. The statistics are also sent out when the queue length crosses over one of the thresholds: *Low_Threshold*, *Avg_Threshold* (*Low_Threshold**2), *High_Threshold* (*Critical_Threshold*/2) or *Critical_Threshold*.. The default values I used can be changed via Policy Manager updates.

4.7.3 The Decision Engine

Algorithm 1 Decision Engine Logic

procedure DECISIONLOGIC
Initialization:
    $softirqUtil \leftarrow \sum$(softIRQ utilization of IRQ-cores)
    $maxSpeed \leftarrow \sum$(maximum frequency of App-cores)
    $minSpeed \leftarrow \sum$(minimum frequency of App-cores)
    $currSpeed \leftarrow \sum$(current frequency of App-cores)
    $qLen \leftarrow$ current socket buffer queue length
    $growthRate \leftarrow \frac{Requests\_Received}{Requests\_Serviced}$, for current *Sensing-window*

Perform IRQ-core Management:
    if ($softirqUtil = 100\%$) **then**
        activate secondary IRQ-core.
    else if ($softirqUtil < 100\%$) **then**
        de-activate secondary IRQ-core.
Perform Queue Management:
    if ($qLen > Critical\_Threshold$) **then**
        Initiate Overload Response.

    else if ($qLen > High\_Threshold$) **then**
        $targetSpeed \leftarrow maxSpeed$
        Spawn new App-threads, if possible.
        Prioritize App-threads to Real-time, if possible.

    else if ($qLen > Avg\_Threshold$) **then**
        $boostFactor \leftarrow 1 + \frac{qLen - Avg\_Threshold}{High\_Threshold - Avg\_Threshold}$
        $gCoeff \leftarrow boostFactor * growthRate$
        $targetSpeed \leftarrow \min(currSpeed * gCoeff\ ,\ maxSpeed)$

    else if ($qLen > Low\_Threshold$) **then**
        $slackFactor \leftarrow 1 - \frac{Avg\_Threshold - qLen}{Avg\_Threshold - Low\_Threshold}$
        $gCoeff \leftarrow slackFactor * growthRate$
        $targetSpeed \leftarrow \max(currSpeed * gCoeff\ ,\ minSpeed)$

    else
        $targetSpeed \leftarrow minSpeed$

The Decision Engine generates the control plane decisions for the server. It receives IRQ-core and queuing statistics from Stat-Collector and executes its decision logic as described in the Algorithm 1 (above) to perform a runtime resource allocation decision. The decision logic performs an adaptive queue-length control based on a Proportional Derivative controller with specific adaptations for a light-weight kernel level implementation. The logic strives to maintain the queue levels at an application-specific average threshold (*Avg_Threshold*) queue length so that a newly queued request

does not violate the queuing delay. However it also attempts to reduce kernel overheads caused due to frequent user-kernel communications and unnecessary CPU frequency changes. After every *Sensing-window*, the Proportional component called as *slackFactor* (line 30 in Algorithm 1) or *boostFactor* (line 25 in Algorithm 1) is calculated depending on the current queue length. A *growthRate* value (ratio of bytes consumed to bytes received in the queue) is also calculated (line 8 in Algorithm 1) which serves as the Derivative component for correcting queue length deviation from the *Avg_Threshold*. Greater the Proportional component (boostFactor or slackFactor) higher the queue length deviation and thus greater the required change in *targetSpeed*. Greater the Derivative component (growthRate), higher the rate at which the queue length is deviating and thus higher the required change in *targetSpeed*. Together, both these components, serve as insight to calculating the *targetSpeed* (line 27 and 32 in Algorithm 1) required to converge the queue length to *Avg_Threshold*. The weightage (co-efficients) values for the two components dictate how fast the queue values are expected to converge back to *Avg_Threshold*. In my implementation, I heuristically chose both co-efficients to be 1. The *targetSpeed* is the cumulative App-cores frequency as suggested by the Decision Engine and translated to individual core frequencies (as described in Section 4.7.4) by the Resource Manager.

I divide the queue length range into 4 logical bands since different load ranges require different response behavior in the decision logic. At very low loads, i.e., when queue length is below *Low_Threshold*, the App-cores are run at lowest frequency (line 35 in Algorithm 1). The *Low_Threshold* value is chosen such that queuing delay incurred by incoming requests, upon being serviced at the lowest core frequency, does not violate the SLO target. Similarly, the *Avg_Threshold* value is chosen such that at some intermediate CPU frequency level, incoming requests meet the SLO target. As mentioned earlier, the *Avg_Threshold* is chosen to be twice of *Low_Threshold* for my study. *High_Threshold* denotes the queue length such that when serviced by cores running on highest frequency, SLO is still comfortably met. Note that the queue length band representing *Low_Threshold* to *Avg_Threshold* would have a different range than *Avg_Threshold* to *High_Threshold* band. Thus the Proportional factor in the controller need to differently calculated for both bands. The *Critical_Threshold* is the queue length at which an incoming request is serviced just within the SLO target latency, despite all App-cores running on highest frequency. As

mentioned earlier, the *High_Threshold* is chosen to be half of *Critical_Threshold* for my study.

For the *High_Threshold* to *Critical_Threshold* band (line 19), App-cores continue to run on highest frequency. At this point the logic assumes to be receiving a burst arrival of packets so that in future Sensing-windows, the growth rate would fall below 1 and the queue length would fall below *High_Threshold*. Note that increasing *High_Threshold* closer to *Critical_Threshold* level would enable the requests to be serviced at highest frequency only when queue gets longer. Though this reduces power consumption, it also involved higher risk of SLO violation during phases of burst arrivals. Though I chose a wide range for this band, a narrower band could further improve the energy saving (shown in Figure 4.7.5) of applications at the risk of SLO violation. However, I did not perform such sensitivity studies for the setup. To harness the queue length in this band, I proposed two possible actions. First, prioritizing the App-threads to real-time priority, thereby avoiding them being switched out of their execution context by other applications. And second, spawning new App-cores to consume the queue faster. However the implementation of these approaches are beyond the scope of this work. If the queue length exceeds *Critical_Threshold* (line 16), despite the prior mentioned actions, the LC-Service must brace for an SLO violation and respond accordingly.

4.7.4 Resource Management

The Resource Manager is the component that implements the control plane decisions by performing the resource allocation based on Decision Engine's request. To add or remove IRQ-cores, it changes the SMP-affinity of the rx-ring to include or exclude a secondary IRQ-core. An alternative implementation for request distribution would be to dynamically reconfigure the number of NIC rx-rings, but that can lead to packet drops during the reconfiguration window and tail problems for dropped requests. For controlling the service rate of LC-Services, the frequency is scaled for one or more App-cores using the userspace governor for Linux Cpufreq driver. The CPU frequency transition takes from 150 usec to 400 usec (depending on the source and target frequency [63]). This latency does not affect the longer, low request rate Tailbench application jobs. However, for short high request rate jobs (like memcached), a faster transition time [110] could improve implementation results.

Translating Target Speed to Frequency: The suggested scaling proportion from the Decision Engine is used to calculate the frequency to which App-cores need to be boosted. For instance, 4 service instances running at 1.2 GHz aggregate to 4.8 GHz. Thus a scaling factor of 1.25 implies a service rate aggregate of 6 GHz, i.e., 4 cores running at 1.5 GHz. However, only a few discrete frequency levels are supported on actual cores. For example, my testbed supports 1.4GHz and 1.6GHz, but not 1.5GHz. To achieve 6 GHz, two cores are each boosted to 1.4 GHz and two to 1.6 GHz. Since power consumption grows super-linearly with CPU frequency, cores running at lower frequencies are more energy-efficient. Thus instead of running a few cores on very high frequency (for e.g., 3 cores on 1.2 GHz and 1 core on 2.4 GHz), the frequency update is distributed across App-cores. Similar calculation are performed to scale down the frequency of CPUs when the queue length decreases to a lower threshold.

4.7.5 Energy Savings

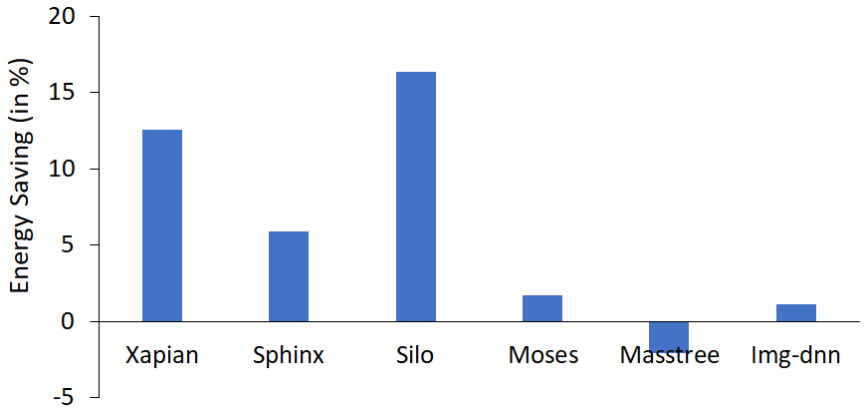


Figure 4.13: Energy saving achieved by the Runtime Manager for six different Tailbench applications over the standard on-Demand CPU frequency governor.

I executed 6 Tailbench application services at different request rates that were all high enough to exceed the aggregate service rate at the lowest App-core frequency (1.2 GHz). I empirically set the *Critical_Threshold* to be between 10x (for longer jobs like Sphinx) and 1000x (for shorter jobs like Masstree) times the minimum service time for a request on a core running at the base frequency. I compared the energy consumed by the server node CPUs using an on-demand frequency governor with the Runtime Manager using a user-space frequency governor (Figure 4.13). The results show

that with my Runtime Manager, the server can process the same number of requests with lower energy consumption. This energy gain is achieved at a cost of higher median and tail latency - however, the queue length seldom exceeded the *High_Threshold* for the tested request values, so the SLO was never threatened.

While Xapian, Sphinx and Silo clearly showed energy savings, Masstree showed the opposite. An in-depth analysis of Masstree revealed that there was no socket buffer queue build-up even for a very high query rate. A closer analysis of the code revealed that the task threads simply received a large batch of 256 requests in a single packet and immediately responded back with an acknowledgment without performing any processing on the requests. Since this is not representative of a typical LC-Service, the negative results, can be ignored. On other hand, Moses suffered from cache contention among its 4 LC-Service threads [106], causing the effective service rates to vary erratically leading to higher CPU execution frequencies and hence lower energy savings. A more fine-grained threshold management and scaling might further reduce the energy consumption.

4.8 Related Work

Network stack bypassing approaches, such as DPDK [111], are used in servers to eliminate interrupt overhead and perform request queue management at application layer. But such customized approaches are vendor-specific, platform-dependent and support only a handful of NICs. This study targets generic interrupt-driven network architectures used in high-speed cloud end-systems. The proposed Centralized-Interrupt handling approach would not be suitable for systems using polling based DPDK. With DPDK, the NIC driver runs continuously in a polling mode and hence the buffer queue maintained in kernel sockets would then be maintained in DPDK application layer buffer. Thus the proposed queue-length based frequency and core scaling is also applicable in systems using DPDK and should result in similar gains in energy efficiency.

Tail latency problems were encountered in data centers [6] while implementing infrastructure for low latency user query on distributed data. Li et al. [30] studied the various sources of tail latency within a data center node. Several approaches to reducing tail latency have been proposed, including using

replicas [112] and using hedged requests [6]. Adrenaline [17] prioritizes and increases the execution speed of queries with higher service rates to reduce violation of tail latency SLO. However, such approaches do not address the issue of high energy usage in data centers.

A number of studies have addressed the trade-off between the latency and energy usage for LC applications. Pegasus [14] uses DVFS techniques to power down all nodes during periods of low loads. Timetrader [24] specifically targets nodes with faster response times to balance both latency and energy problems. Energy efficiency improvement through workload consolidation without affecting tail latency violation have been suggested in many other papers. In [26], the authors use fine-grained DVFS to adapt the frequency for collocated latency-critical and batch jobs. Rubik [25] uses fine-grained frequency scaling along with cache partitioning to isolate the interference between the latency-critical Tailbench [106] applications and batch jobs. Wong et. al. [15] propose an efficiency-aware intra-node level scheduling approach to improve cloud energy efficiency.

Addressing tail variability, as pointed out in [6], is usually achieved by reducing the tail latency and hence is a lesser studied aspect. Chronos [113] achieved predictable low latency for short tasks by eliminating network stack overheads and using user level stack implementation. This approach is similar to DPDK [114]. Latency predictability has also been addressed in different data centers platforms [115, 116, 107]. However, these studies do not provide information regarding the sources of tail latency variability or methods to mitigate them.

Although previous studies have exposed the various sources of tail latency, they do not speculate on their impact for future workloads. In this work, I analyzed the impact of interrupts and queuing on tail latency, energy efficiency and tail latency variability for high request rates. A recent study [117] has suggested pinning IRQ servicing on dedicated cores. I show that centralized IRQ needs to be correctly configured and scaled to avoid becoming a bottleneck. Additionally, I demonstrate the advantage of a centralized IRQ approach to reducing tail latency variability. Earlier works have used application level buffer length [25] or response time [14] driven energy scaling approaches which rely on instantaneous load information to make a scaling decision. But the proposed queue-length based approach allows us to predict future delays in advance and adapt to it.

4.9 Conclusion

In this chapter, I proposed methods to mitigate a refined goal for Latency-Critical Services at the Server-level: a) meeting tail latency target, b) improving energy efficiency and c) minimizing tail variability. I showed that kernel level softIRq can introduce significant tail variability and reduce energy efficiency for short microsecond order tasks (e.g., Memcached-queries). I proposed a centralized-IRQ method that significantly reduces variability and improves energy efficiency. For relatively longer, millisecond order Tailbench tasks, I demonstrated that socket buffer queuing delays contribute significantly to tail latency elongation. I proposed a Runtime Manager that uses kernel level data, exploits Core-frequency Heterogeneity and performs instance scaling to dynamically adapt the service rate to meet the tail latency target while also saving energy (up to 16% energy saving).

Chapter 5

End-to-end Service Level Objective: A Novel User-centric Paradigm for LC-Services

In previous chapters I proposed various control plane strategies to exploit heterogeneity for LC-Services. But along my research, I realized that the traditional execution paradigm is cloud-centric and guarantees only intra-cloud latency. It does not guarantee end-to-end user latency or user's quality of experience (QoE). To guarantee user QoE, a novel user-centric paradigm must be defined and novel control plane strategies must be developed. In this Chapter, I first demonstrate the need for a novel paradigm that guarantees End-to-end Service Level Objectives (ESLO) for LC-Services. Then I describe the control plane enhancements required to implement this paradigm for a microservice-based deployment. As a first step for this new paradigm, this chapter aims to enhance the current control plane to meet the SLO for user QoE (or end-to-end latency) while maximizing server utilization. This implicitly addresses the tail-energy problem for LC-Services, albeit in the new paradigm. This new paradigm supports opportunities to exploit various forms of heterogeneity. In the next chapter, I describe a method to exploit cluster heterogeneity for further energy saving in the proposed ESLO-enforcing framework.

5.1 Background

Human perceptual abilities [1, 2, 3] drive the response time requirements for Latency-Critical Services (LC-Services). For example, response latency bounds of 10ms for AR/VR applications and 100ms for live search applications are required to guarantee an immersive experience for users. Requests violating these latency bounds cause significant deterioration in a user's QoE [5]. Since user QoE impacts application ratings and revenues, LC-application owners want strict latency guarantees. However, current cloud frameworks do not offer such guarantees.

The cloud eco-system for an LC-Service is comprised of three stakeholders: the users, the Cloud Service Provider (CSP), and the CSP-Client (or LC-application owner). The CSP-Client rents the cloud resources from the CSP to host LC-Services which service user requests. To increase their revenue, CSP-clients expect to achieve a high QoE for users, typically above some QoE threshold. User feedback-based studies suggest that QoE for LC-Services decreases as end-to-end delay (E2E delay) increases. Note that E2E delay is the response time measured at the user end. Typically, the QoE-E2E delay relationship follows a sigmoid function [46] wherein the user QoE degrades continuously with increase in E2E delay (see Section 5.2.1). To achieve a QoE target, CSP-clients require a corresponding E2E delay to be met. However, CSPs only guarantee Service Level Objectives (SLO) for the intra-cloud delay (i.e., response time measured at the cloud gateway) - not the E2E delay. A typical SLO is specified as the 99th percentile (or P99) of this intra-cloud delay (IC delay) not exceeding some pre-specified delay target (e.g., 100 ms). To emphasize, the cloud-centric SLO ignores the external network delay faced by requests which can significantly impact user QoE. The impact of ignoring the external network delay is exacerbated when it is a significant part of the E2E delay.

As LC-Service users typically expect sub-second E2E delays, standard external network delays in the order of 10s-100s of milliseconds can significantly impact user QoE. Furthermore, since external network delays vary across users and over time, the intra-cloud delay guarantees do not translate to E2E delay guarantees. Thus there is a disconnect between user QoE and CSP's SLO. To meet the E2E delays required by the CSP-client, CSPs must adopt a user-centric SLO that accounts for

the external network delay.

In this work, I propose the use of an End-to-End Service Level Objective (ESLO) that extends the traditional SLO-based cloud frameworks to guarantee and E2E delay target and hence user QoE target. I design and implement ESLO-aware scheduling and scaling strategies that account for external network delay variability. These strategies are model driven and are implemented as ESLO-aware functions in the Kubernetes [31] framework. They use real-time information about external network delays of requests to meet ESLO and maximize server utilization. Maximizing server utilization minimizes the number of instances and cores required to meet ESLO. The main contributions of this work are the following:

1. I propose a novel ESLO definition to meet a desired User QoE target (Section 5.3). I implement an ESLO-enforcing control plane framework in Kubernetes on a real cloud testbed (Section 5.6).

2. I present multiple ESLO-aware deadline-based server-level scheduling strategies to improve tail QoE. For example, the MinTardy [118] and the proposed Value-based EDF strategy (Section. 5.4) achieved more than 20% higher utilization compared to external network delay unaware FCFS scheduling policy.

3. I develop an ESLO-aware instance scaling strategy that estimates and scales the number of LC-Service instances required to meet the ESLO target at a given load (Section 5.5).

4. Finally, I perform a case study on enforcing ESLO for an Edge-cloud deployment of an LC-Service benchmark with low E2E delay requirements (Section 5.6).

5.2 End-to-end Service Level Objective (ESLO)

The End-to-end Data Path Figure 5.1 shows the end-to-end request-response path of a cloud-hosted LC-Service. User HTTP requests are received by a frontend server (LB-Pod[1]) which serves

[1]Pod is a Kubernetes equivalent of an application instance. LB-Pod represents an instance of HTTP Load-Balancer hosted by frontend servers.

as reverse-proxy and performs load balancing in assigning requests to the backend servers. A new TCP (frontend) connection is first set up for every non-connected client. In practice, a user may have multiple clients. However, in this study I will consider one client per user and hence use the terms client and user interchangeably. Based on the HTTP header fields, the LB-Pod identifies the target service for the requests. For every new frontend connection, a corresponding TCP (backend) connection is set up to a target Service Pod. All subsequent user requests received by the LB-Pod on the new frontend connection are forwarded over the corresponding backend connection to the same target Service Pod. After the request has been processed, the response from the Service Pod is sent back to the LB-Pod. The response may then be sent back to the client or forwarded to another service for subsequent processing. Upon a frontend connection closure by the user, the corresponding backend connection is also closed by the LB-Pod.

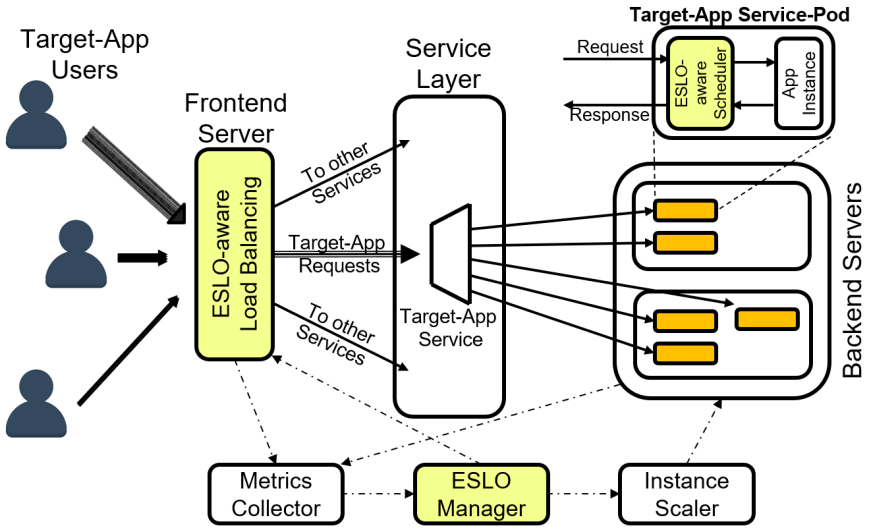


Figure 5.1: This figure shows the end-to-end path of a user request. The lengths of the arrows from the users to the front-end server reflects the variability in the external network delay across users. Variability of the external network delay from the same user is represented by thickness of the lines. The yellow colored boxes correspond to the Kubernetes functions that have been implemented/modified in this work.

Based on the above description, the end-to-end delay of a user request can be decomposed into 5 parts - the forward delay from the user to the LB-Pod, the forward intra-cloud delay, the waiting time and the service time of the request at the Service-Pod, the reverse intra-cloud delay, and finally, the return delay of the response from the LB-Pod to the user. I will use the term **external network delay** to refer to the sum of the forward and backward delay of the request between the

user and the LB-Pod. The sum of the queuing time and the service time at the Service-Pod and the intra-cloud network delay will be referred to as the **IC delay**. I assume that the intra-cloud network delay is relatively small and predictable (achieved using methods discussed in [119, 120]). Thus, the IC delay consists primarily of the queuing delay and the service time at the Service-Pod. The CSP guarantees an SLO on the IC delay. This will be referred to as the **cloud SLO** (or simply **SLO**) and typically, this is a delay bound on the tail of the IC delay distribution. For example, a SLO that guarantees the 95% of IC delays are less than 1 second (this will be referred to P95 IC delay of 1 second). The sum of the IC delay and the external network delay is the **end-to-end delay (E2E delay)**.

5.2.1 Need for ESLO

The user QoE[2] for an online service is a function of the E2E delay; the larger the delay the lower the QoE. The function relating the QoE to the E2E delay is derived using user studies and depends on the application. A typical *QoE-E2E delay* relationship follows a sigmoid function, depicted by the envelope black line in Figure 5.2. This is a representative model for the general class of LC-Services [46, 121, 122] and shows rapid QoE degradation when the E2E delay exceeds 1 second. In practice, the shape can vary for different LC-Services. For example, in AR/VR services, QoE degradation would be sooner and steeper due to shorter E2E delay requirement (i.e. under 100 msecs).

To study the relationship between the QoE, E2E delay, and the IC delay, I performed a simulation analysis. I considered a single Service-Pod to which request arrival follows a Poisson process and the request service time is drawn from a negative exponential distribution with a mean of 100 msec. The arrival rate was set such that the Service-Pod utilization was 75%. I assumed that the Service-Pod had a large buffer and hence no service requests were lost. The external network delay follows a uniform distribution in the range of 1-400 msec. The requests are served using a First Come First Serve (FCFS) policy. Figure 5.2 shows the QoE vs. the IC delay of 10000 consecutive requests (shown as blue dots). In the following discussion I will consider a SLO with a P95 IC

[2]In the rest of the thesis, I will refer to User QoE as simply QoE for short.

delay of 1 second.

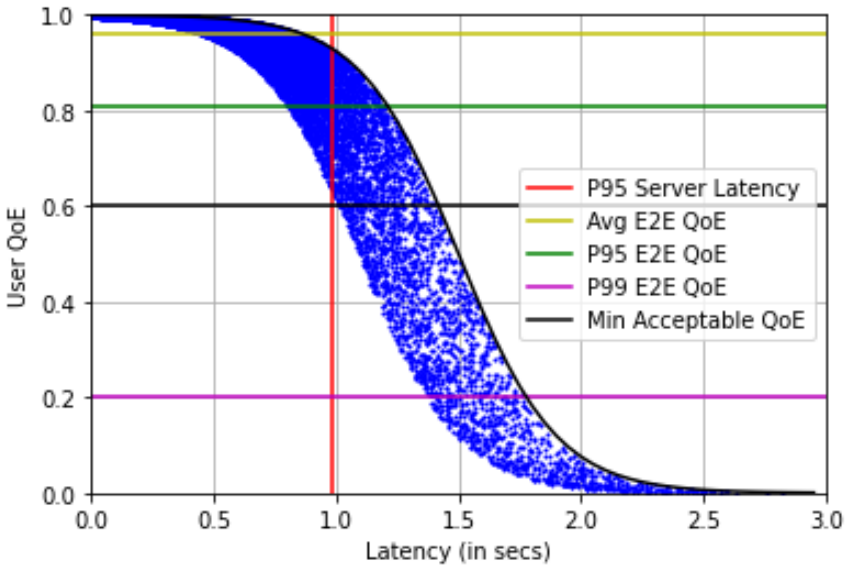


Figure 5.2: QoE as function of the IC delay (x-coordinate value of a blue dot). The black line is the sigmoid function representing QoE as a function of the E2E delay. The horizontal distance from a blue dot to black curve is the external network delay of the request corresponding to a blue dot.

In Figure 5.2, the horizontal distance from a blue dot to the corresponding black dot on the QoE curve represents the external network delay. Intuitively, all requests (blue dots) on the same horizontal line achieve the same QoE, irrespective of their IC delays. This is due to the variability in external network delay. Similarly, all requests (blue dots) on the same vertical line have the same IC delay, but different QoEs. From the figure we can make two key observations. First, QoE for requests for which the SLO is met (all the blue dots left of the vertical red line) can vary from 0.6 to 1.0. Thus, while a CSP may uphold the SLO, it can not guarantee a high QoE for those timely-served requests. Second, many requests that did not meet the SLO (blue dots right of the vertical red line) ended achieving high QoE; many achieving QoE greater than 0.8. The key point is that *due to the variability in the external network delay there is a disconnect between the cloud SLO and achieved QoE.*

5.2.2 External Network Delay Characteristics

As CSPs can not usually manage external network delay, it is typically ignored during application performance engineering and cloud resource management. However, the magnitude and the variability of external network delay can greatly impact E2E delay and hence the QoE. To estimate the magnitude of the external network delay, I performed a ping latency test from 200 globally-distributed servers [123] to an Amazon EC2 instance hosted in the US-west zone. From the results, I found the round trip latencies varying from 6 to 400 msec. Techniques such as Geo-replication can reduce propagation delays, but increases the cost for CSP-clients. In addition to routing and propagation delays due to geographical distance, users also incur last-mile delays ranging from under 10 up to 100s of milliseconds over Wifi and wireless carrier networks. Additionally, network dynamics can also introduce temporal variability in external network delay for a user. In summary, a LC-Service may deal with users with a wide range of external network delays, ranging from under 10 msec to over 500 msec.

5.2.3 Impact of External Network Delay Variability on Server Utilization

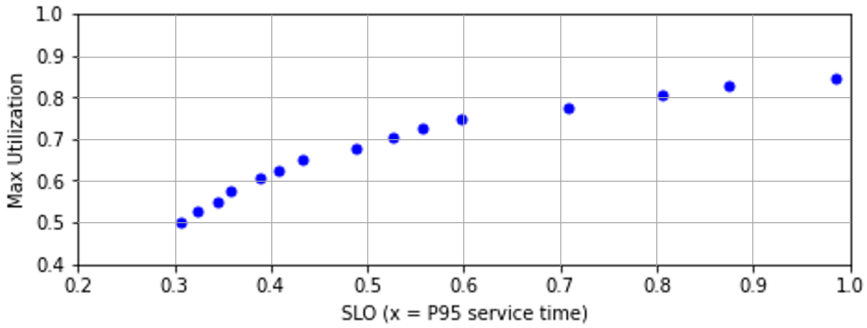


Figure 5.3: This figure quantifies the maximum server utilization that is achievable at different SLOs (95% IC delay target in seconds). The higher the delay target, the higher the maximum server utilization.

Requests with longer external network delays are left with shorter headroom to complete service at Service-Pod without QoE degradation. Thus, the QoE for such requests are more sensitive to longer queuing delays (which are the major contributor to the IC delay [30]) compared to requests with short external network delays. To accommodate varying external network delays, CSPs typically

impose shorter SLO delay target. For example, imposing a P95 IC delay target of 600 msec can accommodate external network delay up to 400 msec while easily meeting E2E delay target of 1 sec. But SLOs with shorter delay targets can sustain smaller queuing delays which requires the Service-Pod to be operating at lower utilization resulting in under utilization of resources. This is corroborated by the observation in Figure 5.3 which shows the maximum achievable server utilization steadily decreasing for shorter SLO delay targets.

5.2.4 Exploiting External Network Delay Variability

Clearly, over-compensating for external network delay in SLO delay targets leads to under-utilization of cloud resources. In this work, I argue the variability in the external network delay among users can be exploited to meet QoE expectation of more users with fewer cloud resources. For this, I leverage the prior observed characteristics of internet traffic. Firstly, despite variability in round-trip time (RTT) of user requests, very few requests (less than 1%) incur significant deviation from their estimated RTTs [124]. Secondly, RTTs for a user remain mostly unchanged over a small sub-second time window [125] and hence can be reliably estimated from the observed RTT statistics of prior TCP transactions (or TCP handshaking).

By inferring the external network delay for a user, service headroom can be determined on per-request basis. Scheduling shorter headroom requests ahead of relatively longer headroom requests can enable more requests to meet the E2E delay target, thereby achieving high QoE for more users. In Section 5.4, I compare various scheduling strategies suited for the proposed ESLO paradigm.

5.3 ESLO-based Cloud Framework

The metric to measure achieved QoE across user requests is determined by the CSP client and is different for different LC-Services. In [46], average QoE (Avg QoE) (averaged over all requests from users) is used as the metric to evaluate various design choices. However, the average QoE is a loose metric and depending on the distribution, even a high average QoE may correspond to a

large number of user having low QoE. Similar to the cloud SLO, a more stricter metric would be that 95th percentile (P95) or the 99th percentile (P99) QoE values be greater than a threshold. I compared the effectiveness of Avg QoE in enforcing strictness when compared with P95 and P99 QoE values as the utilization at the Service-Pod is increased from 50% to 90% (resulting in an increase in the E2E delay). This is shown in Figure 5.4. We observe that the degradation in Avg QoE is very slow, whereas P99 QoE (and even P95 QoE) decays quickly close to 0. Thus, for an ESLO framework, I propose that the CSP client imposes a strict QoE requirement specified by a QoE target for the tail QoE.

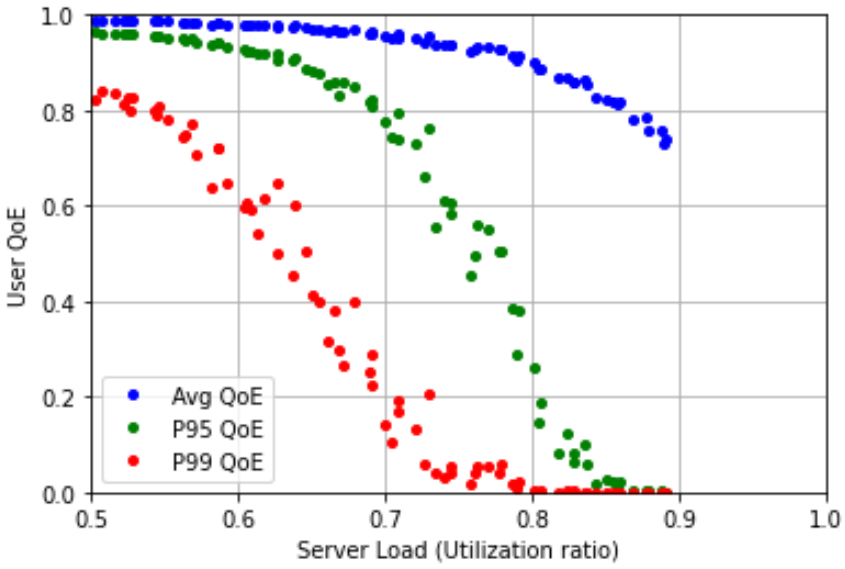


Figure 5.4: The Average QoE (Avg QoE), P95 QoE and P99 QoE with increasing load. The results quantify the expected results that the average is insensitive upto a high load and that the P99 QoE can be low even at moderate loads.

ESLO Definition Given a QoE target for the tail QoE, the relationship between QoE and E2E delay (sigmoid function given in Figure 5.2), imposes a E2E delay target on the tail E2E delay. We refer this as the **End-to-end Service Level Objective (ESLO)**. Specifically, ESLO is an E2E delay target, denoted by E2ED_T for (say) 95 percentile of the E2E delay.

It is important to note that the above ESLO definition incorporates requests that are feasible. These are request for which the E2E delay target can be met. In contrast, infeasible requests are those whose external network delay is so high that they cannot be completed within E2E delay

target, even if they experience no queuing delay. A *Feasible Request* can be identified upon arrival if it satisfies the condition

$$E + S + \alpha + \beta < E2ED_T \tag{5.1}$$

where E is the Expected external network delay and S is the predicted (or average) service time. α and β are empirically determined parameters. α accounts for intra-cloud networking delay, scheduling overhead and service time variations. β accounts for external network delay variance.

ESLO-aware Framework Implementing an optimal ESLO-conforming framework requires support at both cluster and server-level. Figure 5.1 shows my proposed Kubernetes-based ESLO-aware cloud framework. It consists of the following three control-plane components which are highlighted in yellow in the figure.

- **ESLO-aware Scheduler** This is the scheduler associated with each Service-Pod that manages the queue of waiting requests and schedules them for service based on a ESLO-aware scheduling strategy (Section 5.4).
- **ESLO Manager** This component assists the Instance Scaler by monitoring the system metrics from the Service-Pods and user traffic characteristics from the LB-Pod. It then performs ESLO-aware scaling decisions and triggers them through Instance Scaler routines of Kubernetes (Section 5.5).
- **ESLO-aware Load Balancer** This is an enhanced load balancer that collects external network delay information of requests. These statistics are used to perform ESLO-aware scheduling and scaling (Sections 5.4 and 5.5).

In my implementation, when a request is received by the enhanced HAProxy LB-Pod, it adds the external delay information into the requests and forwards it to the Kubernetes Service Layer. At the Service-Pod, this information is extracted by the scheduler to perform ESLO-aware scheduling. The LB-Pod also exports the delay information to the Prometheus metrics monitoring framework. The statistics regarding the requests and the Service-Pods obtained from Prometheus is used by the ESLO manager to orchestrate ESLO-aware scaling.

Inferring the External Network Delay: Effectiveness of ESLO framework relies upon accurately estimating the external network delay. Linux kernel TCP implementation estimates the round trip time (RTT) for every new connection during the 3-way handshake phase. This RTT variable is dynamically updated for every send-Ack sequence. In my implementation, I have used the estimated RTT from TCP-layer of LB-Pod as the external network delay for requests. These RTT estimations are adjusted to account for RTT variation for a given user [124]. I also account for application-specific response packet size in my external network delay estimation. Newer techniques [126, 127] for accurate estimation of external network delay are beyond the scope of this work.

Kubernetes - A Cloud Resource Management Framework: Some of our specific design choices for my framework are influenced by the Kubernetes architecture [31], the cloud framework on which I implemented my proposed design. Hence I briefly describe the Kubernetes architecture first. Kubernetes is currently the most popular cloud resource manager which is being used in more than 78% of production clusters worldwide [128]. Hence we chose to implement the EDAF design on it. In Kubernetes, the smallest unit of deployment and management is known as a Pod, and every service instance is hosted on one. Each pod has its own IP address and is reachable from any other pod hosted on any Kubernetes-managed server node within the cluster. A service instance consists of a Docker image running inside a container. A Pod may host multiple containers. Multiple identical pods together represent a single virtual Service (popularly known as microservice) exposed by the Service layer. In Kubernetes, a Service is represented by a single virtual IP address. Connections or data sent to this virtual IP in the Service layer are load-balanced across the real IPs belonging to Service-Pods. The virtual to real IP address matching is performed in the Linux kernel, either using IPTables rules or by IPVS [129] support. Figure 5.1 also shows the data path within the Kubernetes architecture.

5.4 ESLO-aware Scheduling

The scheduler component is responsible for managing the execution time of the requests at the instance. Traditionally, requests are serviced in a FIFO manner upon arrival at an instance. Therein queuing time for requests are queue-length dependent. Such a strategy is suitable for a cloud-centric framework where every incoming request has same headroom (equal to the SLO target latency). In it, any queued request has waited longer than the requests behind it in the queue, thus having lower remaining headroom. Thus all the requests waiting in a queue at an instance are ordered by their remaining headroom. This is unlike the case in the proposed user-centric paradigm, wherein incoming requests vary in headroom and queued requests are not sorted by remaining headroom. Hence the need for an additional server-level control-plane component that can dynamically manage the scheduling sequence of requests to maximize the number of requests that can meet their headroom.

When a request arrives at a Service-Pod, the scheduler using the information about the external network delay and the E2E delay target, can determine the processing time headroom of the request. It can further classify the request as feasible or infeasible. The goal of the scheduler is to schedule feasible requests such that the number of requests that meet the E2E delay target is maximized. The scheduler runs on each instance and schedules each request as a non-preemptive task.

I consider several existing deadline-aware scheduling algorithms that use the headroom of each request as a soft deadline[3]. The scheduler maintains two queues: a priority queue for the feasible requests, and a low priority queue for the infeasible requests. If a feasible request ends up having a long queuing delay such that it can not be serviced within its E2E delay target, it is moved to the low-priority queue. Requests in the priority queue are serviced in the order determined by the scheduling algorithm; requests in the low-priority queue are served in a FCFS manner.

[3]Here the soft deadline for a request corresponds to the time by which the ESLO-aware Scheduler must complete processing it. I assume request service time can be estimated based on service distribution and request-specific characteristics. There are no hard deadlines for requests, so all incoming requests are processed eventually.

5.4.1 Scheduling Algorithms

In this study I consider the following scheduling algorithms.

MinTardy: In this algorithm the goal of the scheduler is to minimize the number of tardy tasks [118, 130]. Given a set of equal-weighted tasks with deadline and known service times, the *MinTardy* [118] algorithm generates a sequence of task execution that maximizes the number of tasks that meet their deadlines. In a real implementation, however, Service-Pods receive requests at arbitrary times. Thus, the schedule is dynamically computed whenever a new request arrives. This algorithm has time complexity of $O(nlog(n))$, where n is the number of requests. While *MinTardy* achieves schedule that is close to optimal, the scheduling overhead is high during periods of high load when the queues get longer.

Earliest Deadline First (EDF): I implemented the standard EDF algorithm where the request with the smallest headroom is scheduled next.

Value-based EDF: This algorithm is an extension to the EDF algorithm [131]. Note that in order to determine the deadline I estimate the service time of the request. I calculate a parameter $p_i = \frac{qoe_i}{d_i}$ for each queued request i, where qoe_i is the expected QoE of request i and d_i is the headroom. The expected QoE for each request is calculated assuming it is the next scheduled request, and the request with the highest p value is the one that is scheduled. Note that if all the qoe_i values are equal, then this heuristic will be the same as the EDF; the request with the earliest deadline will have the largest p value. But otherwise, this algorithm is better suited for ESLO as compared to EDF because it accounts for the QoE. Since this is an $O(n)$ algorithm, it has a low scheduling overhead.

Least Laxity First (LLF): This is the standard least-laxity algorithm that calculates the laxity in headroom left for every queued request, assuming it is the next scheduled request. The request with the least laxity is scheduled next.

First Come First Serve (FCFS): FCFS represents the default server-level scheduling algorithm used by the CSPs. This algorithm is not ESLO-aware and serves requests in the order of their

arrival at the Service-Pod, as would be in a typical cloud SLO-based execution.

5.4.2 Implementation in Kubernetes

In my Kubernetes implementation, each Service-Pod hosts an ESLO-aware Scheduler container alongside the LC-Service instance. The Scheduler container serves two purposes. Firstly, it acts as a sidecar proxy for the LC-Service instance and performs pre/post-processing on the requests/responses such as decapsulation/encapsulation of HTTP headers. To determine the remaining headroom for the request, it extracts the external network delay information and arrival timestamp inserted into the headers by our customized HAProxy. Secondly, it determines the next request to be scheduled from a list of waiting requests and dispatches it to the LC-Service instance over the pods loopback interface. When a request is serviced, the response is sent back to HAProxy LB-Pod and the next request, if any, is scheduled.

I employed a sidecar-assisted approach for scheduling requests due to its recently growing popularity [132] for application portability. With this approach, LC-Service applications or containers need not be updated. The sidecar proxy can relay the requests to LC-Service instances in their native supported format (i.e., HTTP or TCP). Since inter-process communication within a server typically incurs a latency under 10 microseconds [133], no significant impact can be expected on millisecond order request headroom. Also, since the Scheduler operates at pod level, the arrival rate and queues are significantly smaller compared to those at the LB-Pod.

5.4.3 Performance Comparison

I first studied the effect of employing an ESLO-aware scheduling strategy on QoE. For the same configuration used in Figure 5.2, I collected the IC delay and E2E delay statistics for a *MinTardy* algorithm based Scheduler with a single Service-Pod. Using the E2E delay I obtained the QoE for each request and plotted the Figure 5.5. I observed significant improvement in tail QoE compared to the SLO-based implementation (Figure 5.2). The P99 QoE improved from 0.2 to 0.75 for the same load, thereby meeting the QoE target. I also observed marginal improvement in average

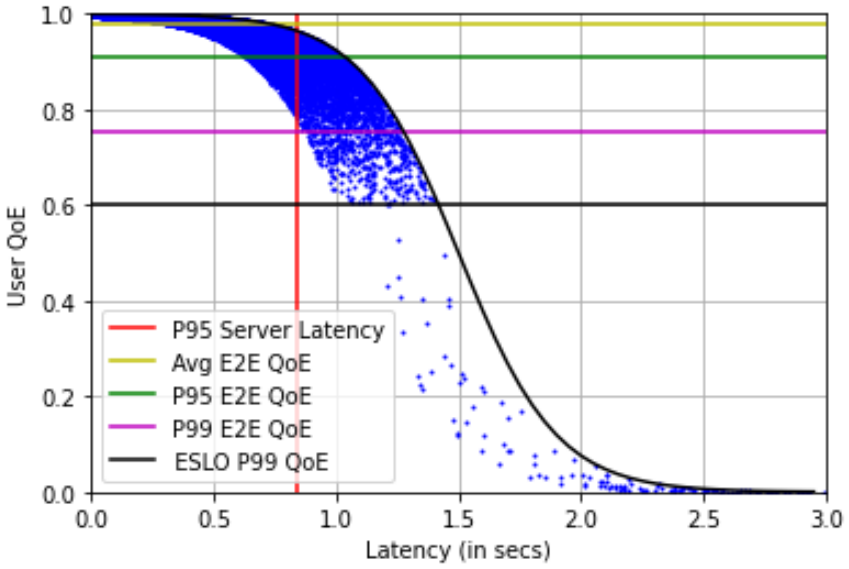


Figure 5.5: QoE as function of the IC delay (x-coordinate value of a blue dot) under MinTardy scheduling. As in Figure 5.2, the black line is the sigmoid function representing the QoE as a function of the E2E delay. Comparing this to Figure 5.2 I see that a deadline-based scheduling at the service instance can significantly improve the P95QoE and the P99QoE.

QoE. This improvement, however, comes at a cost of reduced QoE for requests not meeting the E2E delay target. As observed, the sparse region below the ESLO QoE target shows that unlike in a traditional SLO-based approach, I chose to penalize the violating requests. I did this because delaying the execution of these requests creates room for more requests to be scheduled to meet their deadlines. Due to the sigmoid function representing the relationship between QoE and E2E delay, servicing a potentially QoE-violating request not only achieves sub-optimal QoE but also delays the already queued requests. Hence I chose to implement the two-level queue for the ESLO-aware strategies described earlier.

To analyze the performance of the various ESLO-aware scheduling strategies, I compared their tail QoE with that of the ESLO-unaware FCFS strategy. For a fair comparison with FCFS, I ensure that all requests are *Feasible Requests*. I observed (as shown in Figure 5.7) that MinTardy and Value-based scheduling meet an E2E QoE target of 0.6 up to a higher load compared to other strategies. Since MinTardy minimizes the number of threshold violating requests, it achieves the highest utilization (84%) while meeting a P99 QoE target of 0.6 (Figure 5.7). Value-based scheduling also performed comparably and had high P99 QoE values. In contrast, the ESLO-

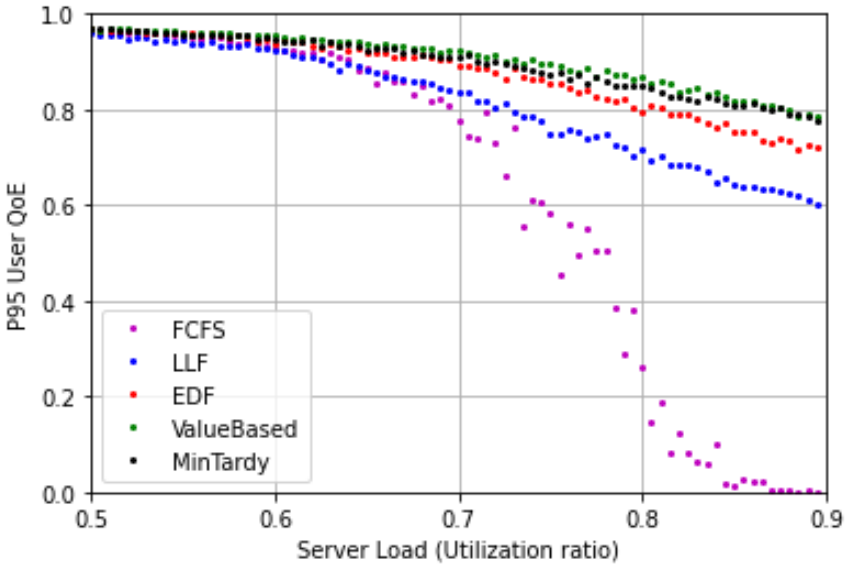


Figure 5.6: Comparison of the P95QoE for the scheduling algorithms at various load levels. The MinTardy and Value-based EDF (VB) perform the best.

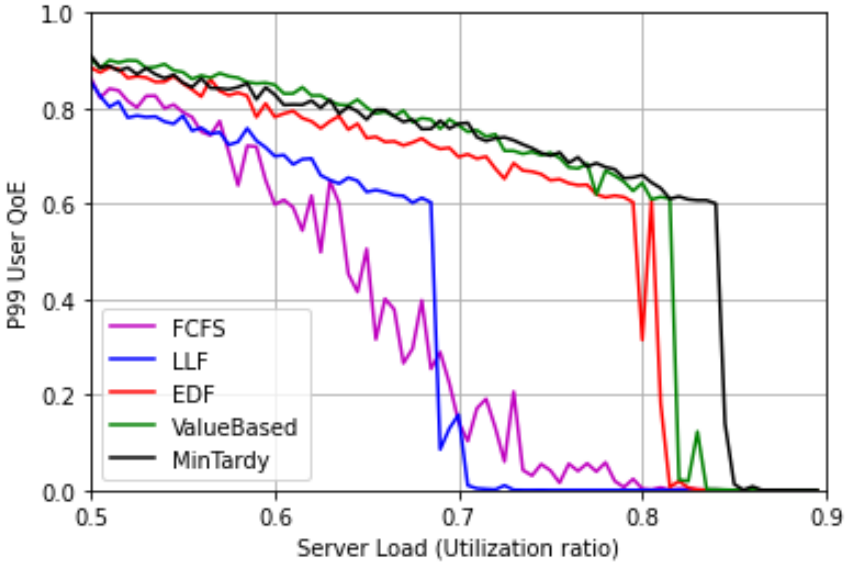


Figure 5.7: Comparison of the P99 QoE for the scheduling algorithms at various load levels. The MinTardy and Value-based EDF (VB) perform the best.

unaware FCFS strategy could achieve only about 60% maximum utilization. In addition, all the ESLO-aware strategies achieved above 90% for the same P95 QoE target, as compared to 75% maximum utilization in the case of the ESLO-unaware FCFS strategy. This implies that Service-Pods hosting ESLO-aware Schedulers can be allowed to run at much higher utilization without risking ESLO violations. On a large scale cloud deployment, this can result in significant savings,

both in terms of compute resources and energy usage.

Observations: In Figure 5.7, we observe the P99 QoE dropping sharply for the ESLO-aware strategies beyond certain load levels at which point the ESLO target is violated. This can also be seen in Figure 5.5, where requests violating the QoE target primarily end up achieving very long E2E delay and low QoE. It is also interesting to note that the proposed low-overhead Value-based heuristic performs close-to-optimal MinTardy algorithm and better than the EDF algorithm. This is because Value-based takes both QoE and request headroom into consideration, thereby also accounting for the sigmoid characteristics into its scheduling decision.

Practical Challenges: While the proposed ESLO-aware scheduling approaches are effective at meeting ESLO, they need further improvement to address a wide class of LC-Service deployments. First, I assume all requests have equal weights and service characteristics, so the proposed scheduling strategies are not directly applicable to services supporting multiple subscription tiers or request types. Second, the ESLO-aware strategies punish the tardy requests (and Infeasible Requests) harshly and provide no guarantee on their E2E delay. Third, they rely on a decent prediction of request service time to achieve maximal benefit.

5.5 ESLO-aware Scaling of Service Instances

Maximizing the utilization of instance-hosting CPUs enables larger number of requests to be serviced by each instance. But the proposed ESLO-aware scheduling strategy must be augmented with a cluster-level control-plane that manages the load assigned to each instance. As in cloud-centric paradigm, low load can cause under-utilization of compute resources, whereas high load can cause ESLO violation. In the previous section we see that the maximum utilization achieved for an ESLO-aware scheduling can be inferred. Based on this inference, an empirical scaling strategy can be implemented that maintains the minimum number of instances (lower bound) that need to be active to meet the ESLO objective. However to obtain an upper bound on the number of required instances, I went ahead to perform a model based analysis assuming the traditional FIFO

scheduling at the instances.

My model driven approach takes the external network delays into consideration. I assume that the external network delay is a random variable denoted by T. Let S be the random variable representing the pod-level delay of a request. I assume that T is some bounded distribution in the range T_{min} to T_{max} with a density function $f_T(t), t > 0$. Let E be a random variable that denotes the E2E delay. Given an external network delay $T = t$, if the E2E delay of a request must not exceed a threshold D, its IC delay must not exceed the threshold $D - t$. Since S and T are independent, the conditional probability that this request does not exceed the ESLO E2E delay target, D_{eslo}, is given by

$$\begin{aligned} P(E < D_{eslo}|T = t) &= P(S < (D_{eslo} - t)|T = t) \\ &= P(S < (D_{eslo} - t)) \end{aligned} \tag{5.2}$$

Each LC-Service instance is modeled as an $M/M/1$ queuing system with an arrival rate λ requests/sec and mean service rate of μ requests/sec. For such a system it can be easily shown [21] that

$$P(S < (D_{eslo} - t)) = 1 - e^{-(\mu-\lambda)(D_{eslo}-t)} \tag{5.3}$$

Thus, the unconditional probability that a request has an E2E delay less than D_{eslo} (unconditioning Equation 5.2) is then given by

$$\begin{aligned} P(E < D_{eslo}) &= \int_{T_{min}}^{T_{max}} P(S < (D_{eslo} - t)|T = t) f_T(t) dt \\ &= \int_{T_{min}}^{T_{max}} (1 - e^{-(\mu-\lambda)(D_{eslo}-t)}) * f_T(t) dt \end{aligned} \tag{5.4}$$

The way to determine the maximum ESLO-conforming arrival rate, λ_{eslo}, using Equation 5.3 and Equation 5.4 is given by

$$1 - \int_{T_{min}}^{T_{max}} e^{-(\mu-\lambda_{eslo})(D_{eslo}-t)} \geq x \tag{5.5}$$

where x is the desired percentile of the E2E delay. If external delays obey standard distribu-

tions (e.g. normal, exponential or uniform distributions), Equation 5.5 can be readily solved to obtain λ_{eslo}. For more realistic distributions, the external delay range can be discretized and an approximate λ_{eslo} can be inferred in the runtime from Equation 5.6.

$$\sum_{t=T_{min}}^{T_{max}} e^{(\mu-\lambda_{eslo})t} * P(T=t) = (1-x)e^{(\mu-\lambda_{eslo})D_{eslo}} \tag{5.6}$$

The above equation can be solved for λ_{eslo} using either partial Taylor series expansion or numerical methods [134].

5.5.1 Scaling the Number of Pods

The value of λ_{eslo} in Equation 5.6 gives the maximum request load than can be sustained by a single LC-Service instance (a pod) while meeting ESLO. If there are N instances hosted across multiple server nodes and the aggregate load is equally divided among them (using a round-robin or random load-balancing of requests), then the aggregate load denoted by Λ_{eslo} that can be supported while meeting the ESLO is simply $\Lambda_{eslo} = N * \lambda_{eslo}$. Note that this is based on the use of FCFS request scheduling at the pods. As we have seen in Section 5.4, by using different ESLO-aware scheduling strategies it is possible to sustain a higher load within each pod while achieving the required ESLO. As such, with regards to the scheduling discipline, Λ_{eslo} will be a lower bound on the aggregate traffic load that will meet the ESLO. In general the lower bound of maximum cluster load satisfying ESLO, Λ_{eslo}, is given by

$$\Lambda_{eslo} = N * (\lambda_{eslo} - \epsilon) \tag{5.7}$$

where ϵ is the correction factor introduced to account for pod-level scheduling approach and deviations in real service time estimation. ϵ value is obtained empirically during runtime using a single instance. It is worth noting that when at least x% of requests meet the E2E deadline at each instance, it is also met for at least x% of the requests at the aggregate cluster level. Additionally, homogeneous clusters have identical service time distributions across all instances. Hence under ideal load balancing, if the average of the xth percentile E2E delay across the instances is less than

the E2E delay target, then ESLO is met at the cluster level.

I use the above inferences to statically calculate the threshold load-levels Λ_{eslo} corresponding to each instance count. The measured cluster-level load is then used for a reverse lookup to infer the optimal number of required instances N. The number of instances are then scaled up or down to N. In production systems, some buffer instances [21] are maintained by CSPs to accommodate load spikes and to avoid expensive instance creation and tear down operations. In this work I do not explore such optimizations - I simply propose and implement a potential ESLO-aware scaling strategy.

5.5.2 Implementation in Kubernetes

In my implementation, the ESLO Manager orchestrates the Pod scaling. It runs within a dedicated Kubernetes pod and collects metrics regarding the Service-Pods and the LB-Pod from Prometheus. Every pre-configured sampling epoch, queries are performed for the number of current LC-Service instances along with their CPU utilization. The Manager also fetches three metrics reported by the HAProxy LB-Pod to Prometheus for the past epoch. These are the request rate, external delay distribution, and LC-Service response time distribution. The HAProxy code was instrumented to generate the external network delay statistics at each epoch as a histogram of a set of discrete delay bins (as discussed regarding Equation 5.6) and exported to Prometheus. Other statistics were natively supported by HAProxy and only needed to be exposed to Prometheus. The adjustment to the desired number of instances derived using Equation 5.7 was triggered by the ESLO Manager by invoking the Kubernetes Pod-Autoscaler manually. The response time metrics from HAProxy enabled the ESLO Manager to empirically tune the correction factor ϵ using a gradient descent method.

5.6 An ESLO-aware Control Plane Implementation

The ESLO framework can be also be implemented in Edge cloud infrastructure. The external network delays for Edge Services are smaller with a narrower delay spectrum. However, the E2E delay expectations are also shorter. We hosted *img-dnn* LC-Service pods on Haswell processor-based servers in the ChameleonCloud testbed [47]. I considered *img-dnn* the image recognition application from the Tailbench [106] suite that performs handwriting matching using deep neural networks. Benchmark results show that the mean requests service time was approximately 10 msec. To simulate multi-user traffic, I developed a open-loop traffic generation tool as in [106]. From a local client machine, I generated traffic following a Poisson process with mean rate up to 200 requests/second. I used Netem [135] to emulate uniformly distributed external network delays[4] in the range of 1 to 41 msecs. Since the client was on the same LAN as the Kubernetes nodes, the observed user RTT at the LB-Pod was primarily due to the delays injected by Netem. I assumed a stricter *QoE-E2E delay* function obtained by scaling down by a factor of 10. I assumed an ESLO with QoE_T of 0.6 (i.e., $E2ED_T$ = 143 msec) for P95 and P99 QoE.

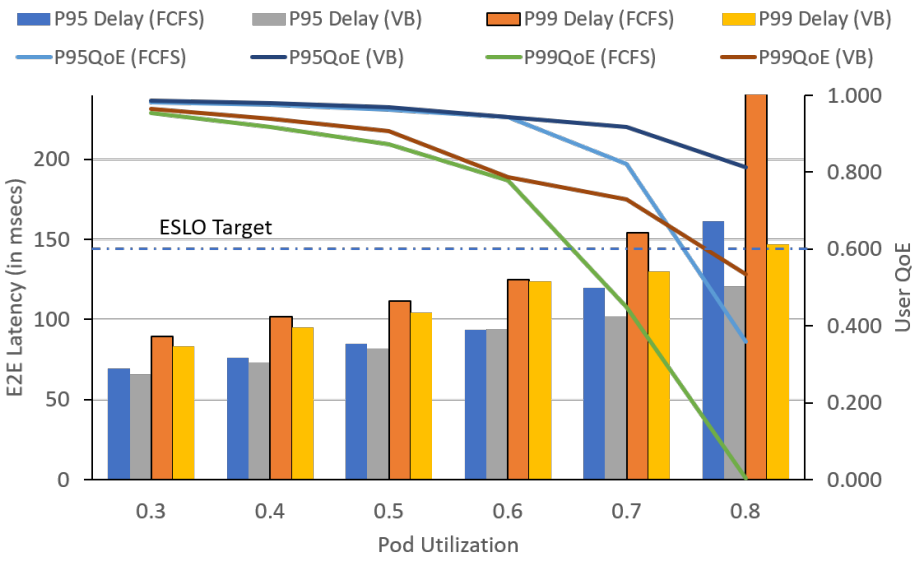


Figure 5.8: This figure shows the performance of the Value-based EDF scheduling algorithm (VB) as a function of load implemented in Kubernetes. The external network delay unaware FCFS algorithm is shown for comparison. With increasing load the P95 and P99 QoE of FCFS is much lower than of VB.

[4]Results for other external network delay distributions omitted for space.

I implemented the Value-based EDF policy and compared it to the default FCFS strategy. While MinTardy and Value-based EDF have similar performance, the former has higher computational overhead. The results in Figure 5.8 corroborate my findings from the previous experiment (Fig. 5.7): The Value-based EDF policy meets the QoE target for both P95 and P99 ESLO up to a higher load achieving more than 75% utilization - at least 15% higher than FCFS.

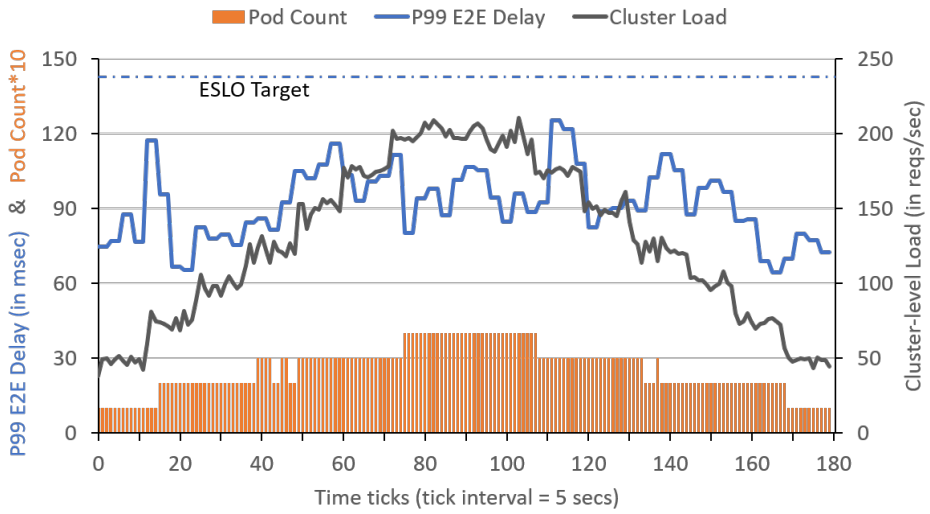


Figure 5.9: This figure shows the effectiveness of the scaling algorithm as the load is changed (black line). The bars at the bottom show the number of pods (scaled by 10). The E2E delay target (dotted blue line) is shown to be met.

I studied scaling by subjecting the HAproxy LB-pod with a variable load (Figure 5.9). The ESLO manager fetches Prometheus metrics every 5 seconds, and scales the number of pods between 1 to 4 based on the observed load and delay distribution over a rolling window of 30 seconds. The P99 E2E delay is measured at the client over discrete 15 sec windows. I observe that the P99 E2E delay maintains a comfortable margin from the E2E delay target for ESLO, despite assuming no correction factor (ϵ=0). This is because the pod-level λ_{eslo} estimation (Eq. 5.6) is based on the FCFS scheduling strategy, while using a Value-Based EDF scheduling allows pods to achieve higher λ_{eslo}. Although the ESLO manager employs more than the required number of instances, they serve as a buffer for load spikes. Since pod creation can take several seconds, pod scaling must account for a sufficient utilization buffer, especially for Edge services with lower E2E delay requirements. For large scale cloud deployments, the correction factor ϵ can be tuned empirically to calculate the desired number of pods based on autoscaling[21] techniques.

Using external network delay statistics, my proposed scaling technique could promptly scale the number of service instances to meet ESLO. Traditional ESLO-unaware scaling would either employ too many instances or violate ESLO targets. Significant amounts of energy is also saved by CSPs/Edge-Providers by increasing server utilization and hosting fewer active servers. Besides, ESLO-aware capability on Edge platforms with even stricter E2E delay targets demonstrates the genericness and versatility of my proposed framework. While the preliminary studies are encouraging, further engineering required for large-scale deployments are beyond scope of the current work.

5.7 Related Works

The vast majority of literature on cloud response time optimizations for end-system servers[27, 136, 92, 30, 112, 17] and intra-cloud network[137, 138, 139, 140, 141, 142] ignore external network delays. Very few works[46] have discussed the impact of external delays on user-perceived response time. [143], etc. present delay breakdowns along user request-response paths, revealing significant overheads introduced by external network delays. [46, 144] etc. further demonstrate the impact of external network delays on the user's QoE. But all this prior work has adhered to the CSP-centric paradigm for SLO enforcement. To the best of our knowledge this paper presents the first work that proposes and implements a paradigm shift to a user-centric End-to-end SLO (ESLO) enforcement.

Deadline-aware scheduling approaches for soft and hard deadlines have been thoroughly studied over several decades. In the cloud-context, such scheduling strategies are employed by resource managers[145] and Job schedulers[102, 32]. But such scheduling approaches are designed for either instances or data-centric jobs, not for user requests to microservices. Some works perform headroom-aware request scheduling[139, 24] on instances to pull back the tail latency of servers, thus meeting the SLO. But such works must also account for variable external network delay to enforce ESLO. There have also been many studies of cloud-based instance scaling strategies[146, 147, 148, 149, 150, 151, 152] using resource utilization and load metrics. Machine-learning based predictive scaling strategies[153] are already supported by cloud Autoscalers. My proposed scaling

is a much simpler but orthogonal approach that enforces ESLO. Existing SLO-enforcing scalers can easily adopt our strategy to support the ESLO-paradigm.

Many CSPs and LC-Services support multiple subscription tiers. LC-Services can also support mixed request types with different service distributions [17] and QoE characteristics[121]. My proposed framework needs further adaptations to evolve into a multi-tier ESLO-enforcing framework. Pricing based utility models for Users(e.g. [154]) or CSP-clients(e.g. [155]) can guide a CSP when chalking out optimal pricing for certain SLO requirements. Given a certain pre-agreed SLO, my work focuses on optimizing resource utilization. The proposed ESLO metrics may be defined through domain specific languages [156, 157].

My work and E2E[46] have some resemblance in that both consider external delay metrics to perform scheduling decisions for user requests. But these are entirely different approaches. E2E maximizes average user QoE by performing external network delay aware scheduling. But what we propose is a paradigm change, wherein CSPs can offer QoE targets to Application owners as new SLOs and guarantees can be provided on tail QoE of Users. This needs a redesigning of the entire control-plane.

5.8 Conclusion and Future Work

My previous works attempted to develop control plane strategies for exploiting heterogeneity in traditional cloud-centric paradigm. During the course of the research on exploting heterogeneity I realized that current paradigm does not guarantee the user QoE expectations. So in this chapter I proposed a user-centric paradigm for LC-Service execution and proposed a novel set of control plane strategies that are needed to implement this paradigm. In effect, this work proposes a paradigm shift from cloud-centric to user-centric SLO enforcement. By acknowledging external network delay and addressing its variability, CSPs can guarantee stricter user QoE based SLOs such as a high target QoE value for 99% QoE. I proposed and implemented scheduling and scaling strategies for this new cloud framework that support meeting a user-centric SLO which is referred to as the ESLO. These strategies exploited the variability in the external network delay. We presented a

proof-of-concept of the entire system on a Kubernetes-based edge cloud supporting an LC Service.

Additional challenges may emerge in large-scale deployments of the proposed framework by CSPs. For example, the scaling strategy may require further engineering to accurately account for the intra-cloud network delays. In chapter 7.2, I present details of several more opportunities for future work in the proposed user-centric domain. I hope that the feasibility of the proposed approach motivates future research in this direction.

Chapter 6

Control Plane Strategies to Exploit Cluster Heterogeneity for ESLO

6.1 Background

In the previous chapter I presented the User-centric paradigm and an implementation framework that could guarantee End-to-end SLO (ESLO) for end-users of microservice-based LC-Services. We saw that by adopting new instance-level scheduling and cluster-level instance scaling techniques in the control plane, CPU utilization could be maximized and resource consumption minimized. However, as discussed in earlier chapters, traditional control plane strategies need to be adapted for real-world heterogeneous clusters. Similarly, my proposed ESLO framework also needs to be adapted to make it heterogeneity-aware.

In Chapter 2 and 3, I presented two different cluster-level approaches to exploit cluster heterogeneity. However the same approaches cannot be directly used in my proposed ESLO framework since incoming request have varying execution headroom. Unlike in Chapter 2, the cluster-level load can not be correlated to a server/cluster configuration since external network delay distribution of the requests too contributes to the choice of optimal set of cores. With an additional input variable in the form of a distribution, RL-learning would be prohibitively expensive and thus I believe infeasi-

ble for this purpose. The heuristics discussed in Chapter 3 rely on identifying the MSG-Capacity for a CPU core. However, in presence of variable headroom, no such metric can be determined for a core.

Since I cannot apply both the prior proposed approaches to the proposed user-centric framework, I present a novel approach in this chapter that takes external delay variation into consideration. I demonstrate how external network delay variability can be leveraged to reduce energy overhead by exploiting cluster-level heterogeneity. I propose a novel delay-spectrum based load partitioning between slow-efficient (SE) and fast-inefficient (FI) CPUs, and show that on a small scale deployment we can achieve between 5-62% power saving (Section 6.4). Note that I do not consider Core heterogeneity or Core-frequency heterogeneity in this chapter and leave exploiting such heterogeneity to future work. Also, I do not consider a generic cluster-level 2D-heterogeneity as in Chapter 3 but rather a 1D-heterogeneity with slow-efficient (SE) and fast-inefficient (FI) CPUs as in Chapter 2.

Note that, the heterogeneity-aware load partitioning approach described in this chapter saves energy by exploiting cluster heterogeneity, but it is not necessarily an optimal approach. Further saving may be achieved by other load balancing approaches based on optimization theory. However, my proposed strategy is easy to implement with popular Load-balancer softwares (like HAProxy) while still achieving significant benefits as shown in later sections. In this chapter, I do not discuss any related work since all works are already discussed in the previous chapters.

6.2 Exploiting Cluster Heterogeneity in ESLO Paradigm

Like in previous chapters, load could be partitioned in an optimal ratio between the different types of server CPUs. But unlike prior chapters, deriving this optimal ratio would need the external delay distribution to be factored into the derivations based on Chapter 5. But the resultant optimal ratio would be highly volatile and would need to be updated every epoch by the load balancer. The corresponding scaling derivations too would be complicated, hard to implement and ineffective in meeting ESLO in real-world system.

To avoid these challenges, I adopted an alternative load partitioning approach. I partitioned the load along the delay-spectrum instead of the conventional load-spectrum, i.e. instead of determining the ratio of distribution between server CPUs, I determine an external delay threshold as a partitioning marker for the incoming requests. Prior works [35, 34] have partitioned requests along the *Load Spectrum*, i.e. when aggregate load exceeded the SLO-conforming load capacity of the SE-cores, extra load was diverted to the FI-cores. In contrast, I propose a load partitioning strategy along the *External Network Delay Spectrum* (described below). This approach is particularly suitable for the user-centric paradigm since external delay variability translates to headroom variability for incoming requests. Unlike in cloud-centric paradigm where headroom is fixed for all requests, the headroom variability in user-centric paradigm presents an opportunity for load partitioning along an alternate spectrum, i.e. the external network delay spectrum.

6.2.1 Delay-spectrum based Load Partitioning

I based my load partitioning strategy on three key observations. First, faster nodes can achieve higher utilization compared to slower nodes when servicing requests with longer external network delay requests. Second, throughput and utilization of slower nodes significantly improves for lower external network delays. Third, past studies [34] have shown that, in a heterogeneous node, maximizing utilization of the SE cores maximizes the energy efficiency of the node. Extending these insights to a heterogeneous cluster motivated me to partition the external network delay range such that faster nodes serve requests with longer external network delay (and hence lower headroom). This enables the SE nodes to achieve higher utilization, while ensuring high throughput for requests with longer external network delay. Again, note here that the objective of this load partitioning strategy is to demonstrate the feasibility of energy saving by leveraging cluster heterogeneity and exploiting external network delay variability. Determining the optimal control plane strategy is left for future work.

6.2.2 The Algorithm

I consider requests with external network delay in the range $[T_{min}, T_{max}]$. The load partitioning task first requires identifying a threshold delay, T_{th}, such that requests in the range $[T_{min}, T_{th})$ and $[T_{th}, T_{max}]$ are forwarded to SE and FI nodes, respectively. To determine T_{th}, I first make an assumption that the number of SE node instances are capped. This is a realistic assumption since, given an unconstrained number of SE nodes, CSPs would always prefer using them to minimize cluster energy overhead. I assume that SE and FI nodes have mean service rates of μ_{SE} and μ_{FI}, respectively. I further assume that $\Lambda_{SE\_T_{max}}$ is the maximum ESLO-conforming cluster load for the entire delay range with maximum possible SE node instances. Note that $\Lambda_{SE\_T_{max}}$ varies with the delay distribution.

As per my proposed control plane strategy, no partitioning is performed when the cluster load (Λ) is less than $\Lambda_{SE\_T_{max}}$. For these load levels, all LC-Service instances are hosted on SE nodes which are scaled following the strategy described in Section 5.5. When Λ exceeds $\Lambda_{SE\_T_{max}}$, I determine T_{th} and partition the load based on the external network delay. I determine T_{th} as follows. Upon partitioning, the aggregate load serviced by SE nodes, $\Lambda_{SE\_T_{th}}$, would be $\Lambda * \int_{T_{min}}^{T_{th}} P(T = t)dt$. The corresponding pod level load λ_{eslo} can be derived from Equation 5.7 and is given by

$$\lambda_{eslo} = \frac{\Lambda}{N} * \int_{T_{min}}^{T_{th}} P(T = t)dt + \epsilon \tag{6.1}$$

$$= \frac{\Lambda}{N} \sum_{t=T_{min}}^{T_{th}} P(T = t) + \epsilon \tag{6.2}$$

Replacing the λ_{eslo} value in Equation 5.5 and Equation 5.6, provides the continuous and discrete version of an equation for T_{th}. The latter is given by

$$\sum_{t=T_{min}}^{T_{th}} e^{-(\mu-\lambda_{eslo})(D_{eslo}-t)} * P(T = t) = 1 - x \tag{6.3}$$

While the resultant discrete version for T_{th} can be hard to solve, I adopt a heuristic approach to determine it in a real implementation. The delay range $[T_{min}, T_{max}]$ is partitioned into bins and the delay probabilities are evaluated for each bin. I evaluate the left-hand expression in Equation 6.3

starting with the lowest delay bin consisting of T_{min}. By incrementally expanding the range, we can eventually obtain the T_{th} for which the value of the expression is closest to but less than $1 - x$. The heuristic is only of $O(n^2)$ with the number of delay bins. Hence, it can be executed per epoch for several fine-grained delay bins without significant overhead.

Although I explored only two-level heterogeneity in this study, the proposed approach can be extended to n node types with n-1 delay thresholds, where each delay threshold can be incrementally calculated.

6.3 Adapting ESLO-aware Control Plane for Heterogeneity-awareness

To introduce heterogeneity-awareness in the ESLO-enforcing framework discussed in the Chapter 5, I did not need to add any additional component to the control plane, but instead updated the 3 proposed components as follows.

6.3.1 Adapting ESLO-aware Scheduler

The Scheduler works at the instance level on a single core. Since I did not consider Core-frequency heterogeneity in this work, not heterogeneity-awareness is required to the scheduling strategy. However, the Scheduler must be aware of the service capacity of the hosting core to perform correct service time and headroom calculations. The Task scheduler may perform a sample execution while loading to determine the service capacity of the hosting core. In my implementation I statically updated the capacity values since the values are universal for each core type. The capacity values are then used for subsequent calculations by the scheduler.

6.3.2 Adapting ESLO-aware Scaling

The scaling strategy is similar to that of E-first heuristics-based strategy described in Chapter 3 where scaling is done for the SE core instances as long as an SE core is available. The FI cores are

scaled when load is high enough that SE core instances cannot meet the ESLO. To implement this strategy in Kubernetes, it must be closely co-ordinated with the load balancing strategy.

The ESLO Manager orchestrates both the scaling and the load-balancing strategy in my implementation. Based on the aggregate load metric in the past epoch, the Manager first determines if ESLO can be met with instances hosted on SE-instances only. If so, scaling of instances is performed only on SE nodes. If not, the manager calculates the delay threshold T_{th} as discussed above. Then the manager schedules the maximum possible number of instances on SE nodes and schedules additional pod creation on FI nodes. It performs future scaling on FI nodes until the cluster load falls below Λ_{SE_eslo}. However, the LB-Pod can only perform load partitioning across multiple registered Kubernetes Services. Thus I register a separate Kubernetes Service for each type of node, running the same LC-Service container image. Having separate node-type-aware Services also helps the ESLO Manager to separately scale the instances of two node types (as in Section 5.5). The virtual IPs for both services are registered as backend LC-Service servers with LB-Pod.

6.3.3 Adapting ESLO-aware Load-balancing

The load balancer is the most crucial control plane component that performs the partitioning of the traffic in my implementation. The heterogeneity-unaware load balancer adopted a simple equi-partitioning approach. In my proposed heterogeneity-aware version, the partitioning occurs at two levels. First, the requests are partitioned based on their external network delay metric: the smaller headroom requests are forwarded to the Service-level Load balancer for FI nodes, while the ones longer than partitioning threshold are sent to SE nodes. The Service-level Load balancer then performs the second level of partitioning with an equi-distribution approach among the homogeneous cores of either FI or SE types. The threshold is updated every epoch based on measured cluster load and the subsequently scaling adaptations need to performed as mentioned above.

To implement the partitioning, the RTT information is first collected from the socket layer of inbound TCP connection for request. HAProxy allows custom rules (called Access Control Lists or

ACLs) for traffic management on a per request-basis. Using an ACL rule, the RTT is inserted as a custom HTTP header *Conn-RTT* before forwarding the request over the backend connection. I define another ACL rule which then checks if the value of RTT in the *Conn-RTT* field is less than T_{th}. If so, the request is forwarded to the virtual IP corresponding to the SE-node's Kubernetes Service. If not, it is forwarded to the FI-node's Kubernetes Service. Subsequently the Service layer determines its destination LC-instance pod based on configured IPVS load balancing policy (round-robin for our implementation). To dynamically update T_{th}, I exploit the dynamic ACL reconfiguration feature in HAProxy. The ESLO Manager performs this dynamic update over pod-to-pod IP communication, upon which subsequent requests are partitioned based on the new threshold.

6.4 A Case for ESLO-aware and Cluster Heterogeneity-aware Resource Management on an Edge Cluster

To study the efficacy of my proposed approach, I extended the ESLO framework as per the details described in the previous section and performed the experiments for the edge-based case study as discussed in Chapter 5.6.

Cluster Load	Real T_{th}	Estimated T_{th}	E2E P99 Delay (in msecs)	Num SE-Pods	Num FI-Pods	Power Overhead (in W)	%age Power Saving
25	41	41	142.028	2	0	0.572	62.101
50	27	28	140.054	2	1	2.428	16.360
75	20	21	136.308	2	1	3.381	12.429
100	17	18	142.663	2	1	4.356	16.464
125	12	15	140.116	2	1	6.313	5.010
150	10	13	135.294	2	2	8.004	8.453
175	7	12	124.773	2	2	8.380	4.945
200	6	11	112.782	2	3	8.490	12.682

Figure 6.1: This table of results shows the effectiveness of the load partitioning algorithm with increasing load in a system with two types of nodes: SE and FI. The small discrepancy between the estimated and the actual and load partitioning delay (T_n) to meet the ESLO objective suggests engineering a small correction factor.

To demonstrate the energy saving opportunity, I hosted LC-Service pods on Intel Haswell-based(FI) and Intel Atom-based(SE) servers. Pods on both server types were reachable via separate Kubernetes services and were separately scaled by the ESLO manager. For this study, I assumed the maximum number of SE-hosted pods was 2. Power measurement were done using RAPL counters. I explored the optimal load partitioning that meets the ESLO for various cluster loads. The estimated delay threshold T_{th} was obtained from Equation 6.3. A delay bin size of 1 msec was used to calculate T_{th}. The optimal T_{th} that meets ESLO was experimentally obtained. Results of my experimental study is summarized in the table from Figure 6.1. Without a correction factor, I observed higher deviation from the estimated T_{th} for higher loads. Since the mean service times of SE-pods were at least twice that of FI-pods, they are more more sensitive to an ESLO violation due to traffic bursts or service time variations. The sensitivity would be even greater for LC-Services with shorter delay requirements, and would need a correction factor. However for lower load levels, the discovered threshold of T_{th} closely matched the estimation.

I compared the dynamic power consumption of a heterogeneous deployment with a homogeneous deployment of FI-pods only. Both deployments were dynamically scaled by the ESLO Manager to meet the ESLO. I observed that at low loads, SE-pods alone can be employed, thereby saving a lot of power otherwise consumed by FI-pods in a homogeneous deployment. As cluster-level load is increased, the threshold delay for load partitioning decreases, resulting in an increased section of delay spectrum being serviced by FI-pods. While the SE-pods remain maximally utilized, more FI-pods are employed at higher loads and cluster-level power consumption is mainly driven by the number of FI-pods used and their utilization-level. Since CPU power consumption is known to increase non-linearly with utilization, I observe non-linear energy saving with load increase. Power saving is lower when FI-pods in a heterogeneous cluster had higher utilization.

While I presented a small-scale study to demonstrate the power-saving opportunity, the findings are also applicable to large-scale cloud deployments where significant amounts of energy can be saved by exploiting a few energy-efficient servers during low traffic periods. Besides ESLO-enforcing capability on Edge platforms with even stricter E2E delay targets demonstrates the genericness and versatility of my proposed framework.

6.5 Conclusion and Future Work

In this chapter, I demonstrated the feasibility for energy saving on heterogeneous clusters by designing a novel load partitioning strategy along the external network delay spectrum. I presented a proof-of-concept of the entire control plane by extending the ESLO-enforcing framework described in the previous chapter.

The proposed energy saving approach suggested in this chapter may not be optimal. I neither performed optimization analysis nor derived an optimal theoretic load partitioning strategy. I have also not considered exploiting other forms of heterogeneity (i.e. CPU heterogeneity and Core-frequency heterogeneity) in the user-centric approach. My proposed strategy assumes a 1D-heterogeneity in cluster where the slower core is also the more efficient core. Future studies should address these shortcoming and assumptions to performing optimal energy saving in a heterogeneous cluster.

Chapter 7

Conclusion and Future Works

7.1 Summarizing Research Conclusions

Cloud computing for microservices-based Latency Critical Services (LC-Services) has been heavily researched in the past decade. A major subset of this research has been aimed at solving the tail-energy conundrum [91, 12, 92, 10, 27, 93, 59, 94, 24, 17, 14, 11, 95] as explained in Chapter 1. The continuous evolution of the underlying server hardware, software architectures, frameworks and use cases for cloud-hosted applications requires new approaches to cloud resource management. In addition, as cloud warehouses grew over the past decade, heterogeneity has evolved as a unique challenge for the CSPs. The variability in processing speeds and power profiles among various compute units within the cloud leads to heterogeneity, both in service time and energy footprint of identical request tasks. In the context of this continuously evolving cloud research domain, my research work tries to explore three unique research frontiers: exploiting heterogeneity, improving efficiency in supporting LC-Services, and achieving higher user QoE.

7.1.1 Exploiting Heterogeneity

In this thesis, I propose and analyze control plane strategies that exploit various forms of heterogeneity that exists in current generation cloud clusters to improve the efficiency of the LC-Services. Some of the proposed approaches have great potential for future software and hardware infrastructures. The various forms of heterogeneity addressed and/or exploited in my research are discussed in the following subsections.

7.1.1.1 CPU Heterogeneity

In Chapter 2, I demonstrated that using Heterogeneous Multi-Processors (HMPs) in a cluster not only presents energy saving opportunity at the server level but also at the cluster level. By maximally utilizing the slower but efficient cores of HMPs across the clusters, use of faster and inefficient cores can be minimized across the cluster. I developed Greeniac, a reinforcement learning agent, to learn the optimal service configurations for a given load. I exploited problem specific optimizations to not only speed up its learning phase but also learn the most efficient cluster-level service configurations for different cluster loads. Though HMPs have not gained popularity with cloud warehouse owners, they are particularly promising for smaller cluster deployments such as edge and fog cloud platforms.

Due to lack of availability of commercial HMP-clusters, I developed and implemented an event-based simulator, HMP-ClusterSim, that simulates a cluster based on a queuing model. In each server, I implemented an HMP with multiple small and big cores. The throughput and power characteristics used to simulate the HMP were obtained from real system measurements. I verified the correctness of the simulator by comparing with the theoretical results of M/M/1 server implementation. Subsequently I implemented Greeniac on HMP-ClusterSim to verify its efficacy and perform various sensitivity studies.

I found that Greeniac saved more energy on larger clusters and on HMPs with greater disparity in energy efficiency between its cores. I also found that an increased variability in service times, increased compute load and increased SLO strictness result in faster saturation of small cores and

greater use of big cores. However, Greeniac managed to learn the efficient service configurations in all cases while meeting the latency SLO of LC-Services.

7.1.1.2 Cluster Heterogeneity

In Chapter 3, I discussed experiments performed to demonstrate how cluster heterogeneity, both along the dimensions of capacity and energy efficiency, can result in non-optimal throughput and energy footprint. To reduce this inefficiency introduced by co-existence of multiple types of server processors within cloud clusters, I proposed heuristics-based control plane strategies in the chapter. First, I identified the optimal strategy for each of the two dimensions of heterogeneity, separately. I proposed a novel Maximum-SLO-Guaranteed-Capacity (MSG-Capacity) proportional request distribution technique to address the capacity heterogeneity among CPUs. This approach relies on the identifying the maximum attainable throughput of a CPU that still guarantees the latency SLO for an LC-Service. To address efficiency heterogeneity, I proposed an Efficient-first (E-first) heuristic for request distribution. As per this approach, CPUs with higher efficiency must be always preferred by instance scaling routine to schedule new service instances over less efficient CPUs. Then by superimposing both approaches I proposed an Energy-efficiency and MSG-Throughput (E-MT) request distribution strategy. The E-MT strategy maximizes utilization while minimizing the energy footprint of the target LC-Service on any typical heterogeneous cluster with capacity and energy efficiency heterogeneity.

To perform a generic study on heterogeneous clusters, I extended the HMP-ClusterSim, to simulate a cluster with non-identical servers, each with identical cores. Real capacity and power characteristics of existing processors were used in its simulation of servers.

7.1.1.3 Core-frequency Heterogeneity

In Chapter 2, I described how I trained Greeniac to also consider energy savings gained by exploiting core-frequency scaling or DVFS for the big and small cores. Due to the super-linear relation between frequency and power, CPUs can be relatively energy-efficient at lower frequencies. Power profiles

of CPUs at different frequency levels are measured and used to train Greeniac as possible service configurations for a CPU core.

I also put Core-frequency heterogeneity into use, as discussed in Chapter 4, when exploring OS-level optimization approaches to reduce inefficiency caused by interrupt and kernel-level queuing delays. By adapting scaling application servicing cores, kernel socket buffer queue lengths and hence the queuing delays could be managed using a Runtime Manager. Besides saving energy, this helped in both meeting SLO targets and improving latency predictability for the LC-Services.

7.1.1.4 External Network Delay Heterogeneity

In Chapter 5, I presented a different type of heterogeneity, occurring not in compute elements, but rather in external network delay of user requests. Current cloud-centric SLO-guaranteeing frameworks being agnostic of the external delay of requests, can result in bad user QoE. By introducing awareness about this inherent heterogeneity into the cloud's control plane, statistical guarantees can be instead placed on the user perceived response time or the user QoE. By performing external network delay aware scheduling on servers, both the tail QoE and server utilization could be improved.

In Chapter 6, I presented a unique way to improve the efficiency of cloud clusters wherein cluster heterogeneity could complement external network delay heterogeneity. By serving requests with longer delays on fast but energy-hungry CPUs and non-delayed requests on slow efficient servers, greater energy saving could be achieved while meeting QoE requirements.

7.1.2 Improving LC-Service Efficiency

In the attempt to optimize LC-Service execution, I identified and addressed 3 different efficiency goals. An important objective throughout my research has been to continue to address the Tail-Energy conundrum while exploiting various forms of heterogeneity. In addition, in Chapter 4, I also aimed to improve predictability of server response time.

7.1.2.1 Energy Saving

An important objective of my research was to maximize energy saving for LC-Services. In Chapter 2 and Chapter 3, I showed that opportunity of significant energy saving through cluster level load management by the control plane. In Chapter 2, energy saving were achieved in HMP cluster by always preferring smaller-efficient cores for execution over big inefficient cores. In Chapter 3, this approach was generalized i.e. task scheduling and load distribution in the control plane was biased in favor of the more efficient CPUs in a heterogeneous cluster. Similarly in Chapter 6, energy saving was gained by exploiting cluster heterogeneity for specifically servicing delayed requests on less efficient faster servers. The energy saving shown in all three chapters are significant. In Chapter 4, energy saving was gained within a server by frugally boosting the core frequency of servicing cores, when SLO was threatened by kernel-level queuing delays.

7.1.2.2 Throughput

While meeting the SLO latency target was a prerequisite in my problem statements, I also achieved increased server utilization or cluster throughput using my techniques. In Chapter 3, I identified a unique CPU-specific heuristic called Maximum SLO-Guaranteed Capacity (MSG-Capacity). Cores running at higher than this utilization level start violating SLO. Thus using this heuristic for proportional load distribution between the cores maximizes the throughput in a heterogeneous cluster. In Chapter 5, I showed that using an external network delay aware scheduling can also enable significantly increased server utilization while meeting QoE targets.

7.1.2.3 Latency Predictability

In Chapter 4, I showed that high interrupt rate can cause wide variations in the server response times. This is due to frequent switch-out and migration of application context by the interrupt-handler at high request loads. By using a centralized IRQ handling core and frequency scaling it to meet high IRQ-processing demand, application cores are no longer context switched and achieve a very predictable latency.

7.1.3 Meeting SLOs

SLO enforcement is a rarely explored research topic in this domain. In my work, I studied the impact of tuning SLOs and explored alternate SLO paradigms suitable for future use cases.

7.1.3.1 Cloud-centric SLO

In part of my research (in Chapter 2, 3 and 4), like in most other research in this domain, I have considered the server response latency as the metric for service level objective for LC-Services. This includes the service time and the queuing delay within the server. Specifically, the objective was to provide statistical bounds on server response latency. In Chapter 2, I studied how varying SLO strictness impacts optimal cluster configuration learnt by Greeniac. Increased strictness leads to lower utilization, more use of big cores and thus higher energy consumption. In Chapter 3 too, I showed that in case of capacity heterogeneity within a cluster, capacity-based load distribution suffers lower cluster throughput with increase in SLO strictness, while the proposed MSG-Capacity based distribution is immune to its variations.

7.1.3.2 User-centric SLO

I found the traditional server latency bounds to be cloud-centric, and thus proposed a user-centric end-to-end SLO (ESLO) in Chapter 5. By guaranteeing statistical bounds on end-to-end delay as observed by the users, user QoE could be guaranteed. Guaranteeing ESLO required accounting for external network delay in the cloud control plane. I developed external delay-aware scheduling, scaling and load balancing components for a Kubernetes-managed cluster to implement the ESLO in a real cloud testbed. The proposed ESLO is a paradigm shift from the traditional cloud-centric approach.

7.2 Future Works

My research has identified a number of important research problems that can be explored as a part of future work. In this section I discuss some of these problems. These studies would build upon my work and can further improve resource management in production cloud and edge cloud platforms.

7.2.1 Hardware Infrastructure Related Research

My research was partly built on a simulation-based study on the efficiency impact and opportunities presented by various forms of compute heterogeneity. More extensive study is required to further understand the impact under real production cloud platforms and workloads, especially in wake of evolving hardware infrastructure in public clouds. Following is a list of future research yet to be explored in this area.

- In my research I have not considered the effect of interference experienced by applications from co-located server workloads. However, cloud servers employ many-core processors which might run several service instances and batch jobs from a variety of workloads concurrently. As these concurrent jobs share non-compute resources such as memory hierarchy and network bandwidth, these co-executing processes can introduce significant contention for these resources. The resulting performance and tail latency degradation can result in both SLO and ESLO target violations. Research is required to develop methods to minimize such interference. Additionally, the control plane load balancing and scaling strategies need to adapt dynamically to avoid SLO and ESLO violations due to these contentions.

- Performance of memory-intensive LC-Services could depend on the performance of memory hierarchy in individual servers. Due to the evolution of memory technology, newer memory options such as high-speed DDR RAMs, high bandwidth memory, 3D NAND memory, PCIe-based SSD, etc., would co-exist in future server hardware. This would increase variability in data access time as it will depend on where the application data is located. The resulting

variability in service time needs to be minimized by intelligent data placement algorithms or accounted for when employing control plane strategies proposed in my work. Similar studies are also required for Non-Uniform Memory Access (NUMA) server platforms which have different memory access times for local and remote memory modules. Novel memory management strategies could be developed that can reduce the variability in service times and throughput.

- Future cloud platforms are expected to be sustainable. Switching from brown to green energy brings in challenges like power budget management, limitation on power spikes due to bursty task arrivals, efficient cooling technology, etc.. Control plane strategies need to adapt accordingly when exploiting heterogeneity in such energy-constrained platforms. CPU power management techniques are also evolving in modern CPUs due to the use of faster and efficient voltage regulators, power gating and power-limiting technology. These techniques affect the throughput, tail latency, and efficiency of LC-Services dynamically and thus require to be managed in coordination with control plane strategies to maximize the performance of a heterogeneous system.

- A significant source of response delay could arise from the intra-cloud network delay. In my research, I ignored this delay. However, when designing the strategies for longer millisecond-order tasks, intra-cloud network delays must be accounted for when they are comparable to service latencies of LC-Services. Making the intra-cloud delay more predictable would make it easier to guarantee ESLO. Advancements in cloud network topology and infrastructure will allow proposed control plane strategies to be implemented in the network hardware. Software Defined Networking (SDN) can also be leveraged to implement novel techniques to exploit heterogeneity efficiently.

- Current generation of high-bandwidth network interface cards (SmartNICs) also posses compute capabilities. Service processing may be offloaded to SmartNICs for a faster and energy-efficient processing without OS overhead. However, research is required to investigate for possible bottlenecks in SmartNICs specifically for non-application-level execution path at different load levels. Hardware-specific enhancements in custom NICs could also enable efficient and effective request scheduling within the servers to implement ESLO-aware resource

management.

- Several prior research works have studied ways to exploit heterogeneous architectures like GPUs, FPGAs, and TPUs, for efficient cloud service execution. While accelerators are highly energy-efficient, they are not generic enough to solely execute LC-Services. However, with future transition from discrete accelerators to CPU-integrated accelerators, cluster level heterogeneity can result in much wider disparities in capacity and efficiency of CPUs for certain workloads. Control plane strategies for scaling and task scheduling will need to be revisited for enforcing ESLO guarantees.

7.2.2 Software Framework Related Research

Cloud platform host a highly complex software infrastructure involving several control plane components that perform many tasks including resource management, monitoring, load balancing, power management, dynamic network configuration, and data management, both at the cluster level and at the server level. My research described in this dissertation will require additional studies and adaptation before being adopted on a production-ready public cloud platform. Following are some of the areas of future work in this domain.

- Public cloud providers often use custom network protocols or customized TCP/IP configurations for communications between the servers. With millions of flows being simultaneously handled by the networking software stack, intra-cloud network delay depends largely on the congestion control algorithms, buffering capacities, type of flows and the current state of network traffic, Since I have not explored networking aspect of LC-Service deployment, further studies are required to take them into considerations while performing efficient scaling and load balancing decisions across networked clusters.

- In this thesis, I have demonstrated the benefits of exploiting cluster level heterogeneity, but on the scale of a small cluster. Typical public/private cloud platforms perform resource management on thousands of servers. Scaling multiple LC-Services, load balancing requests across numerous machines and locality-aware resource allocation are not straightforward on

large clusters. Adapting my proposed approaches for such large-scale deployment requires further studies.

- Over the past decade, deployment units for application services have changed from VMs to containers to micro-VMs and more recently Unikernels. Each of these approaches incurs different server-level overheads and requires its own set of optimal calibrations for the virtualized server resources. For a multi-core server hosting multiple workloads to maximize its utilization, the network stack must be analyzed for interrupt and queuing-related bottleneck, in the base OS platforms, the hypervisor platform and the deployment units to identify sources of inefficiency.

- The cloud software infrastructure has become extremely heterogeneous over the past decade. To cater to a variety of application owners and to compete with other cloud owners, CSPs continuously work on introducing novel application execution paradigm and frameworks. The deployment models may be public/private or hybrid. The service models could be IaaS, PaaS, SaaS, CaaS or FaaS based. Similarly, applications might rely on bigdata frameworks like Hadoop and Spark or require streaming data processing using platforms such as Amazon Kinesis, Azure Stream Analytics, and Apache Kafka. For resource management, CSPs rely on container orchestration frameworks such as Kubernetes (or its variants), Azure Container Instances, and Amazon ECS. Recently popular serverless compute platforms such as AWS Fargate, AWS Lambda, Google Cloud Run or Azure Functions also present cost-effective and low overhead deployment alternatives for application owners. My research is primarily based on Kubernetes-hosted microservices. However, a comprehensive study is required to exploit heterogeneity and implement user-centric SLOs in other platforms as well.

- While my research is applicable to generic cloud infrastructure, application deployment could be partitioned across multiple cloud domains in a multi-cloud setting. The service infrastructure might also spread across multiple tiers, from public clouds to fog cluster to edge platforms. Exploiting heterogeneity and guaranteeing ESLOs on such multi-tier deployments can be even more challenging and needs further study.

7.2.3 Differences in Application Characteristics

- Cloud-based applications are typically implemented a collection of several microservices communicating with each other. An incoming request traverses through a Directed Acyclic Graph (DAG) of services before producing the final response for the users. In this this I the techniques used were applicable for a single service. To guaranteed SLO for a chain of LC-Services requires a more holistic approach. Accordingly, heterogeneity exploiting strategies need to adapt too. Efficiently enforcing ESLO on such a DAG implementation would need further studies.

- In my studies I assumed the LC-Service service time to follow a standard distribution. However real microservice deployments can witness wide variations in service times. This variation could be either due to different request types or due to variation in persistent data access time. Such variations must be accounted for when performing control plane level management of heterogeneous clusters.

- Application owners sometime offers multiple subscription tiers for their application usage. Different tiers might have different SLO requirements. The same might apply to ESLO as well. Implementing control plane strategies for such applications requires prioritization of request based on their tiers. Thus the proposed load balancing and scaling techniques for heterogeneous clusters and ESLO-enforcing framework would have to take such additional user context into account.

7.2.4 User Characteristics

Requests received by the LC-Services come from different users. Current cloud-centric frameworks do not consider user context when performing control plane decisions. But to meet user-centric SLOs, user context need to be taken into account while making control plane decisions. For an LC-Service, type of requests, their traffic patterns and request inter-arrival times depend significantly on its user characteristics. To make timely decisions that save energy and meet SLOs, users characteristic could be learnt through machine learning approaches. Then this can be exploited in

developing pro-active control plane strategies for load partitioning on a heterogeneous cluster.

7.2.5 Business and Pricing Models

From the perspective of CSPs and application owners, SLOs and customer pricing models significantly impact each other. CSPs would like to offset the total cost of ownership of cloud warehouses by charging the tenants and application owners in order to meet their revenue targets. Application owners in turn would choose the cloud providers that offers the best price for their expected quality of service. Accordingly, they develop their custom pricing models to meet their revenue goals. Developing pricing models that are revenue generating for both stakeholders is complex and requires balancing the LC-Service targets, energy objectives, infrastructure logistics and service costs. For a new execution paradigm, newer models need to be built. Similarly, a thorough study on revenue benefit gained by exploiting heterogeneity on large scale can trigger a shift in cloud industry towards increasing cluster heterogeneity to reduce future energy footprint.

Bibliography

[1] Robert B. Miller. "Response Time in Man-Computer Conversational Transactions". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). San Francisco, California: Association for Computing Machinery, 1968, pp. 267–277. ISBN: 9781450378994. DOI: 10.1145/1476589.1476628. URL: https://doi.org/10.1145/1476589.1476628.

[2] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. "Impact of Response Latency on User Behavior in Web Search". In: *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '14. Gold Coast, Queensland, Australia: Association for Computing Machinery, 2014, pp. 103–112. ISBN: 9781450322577. DOI: 10.1145/2600428.2609627. URL: https://doi.org/10.1145/2600428.2609627.

[3] Dennis F Galletta et al. "Web site delays: How tolerant are users?" In: *Journal of the Association for Information Systems* 5.1 (2004), p. 1.

[4] Matthias Dick, Oliver Wellnitz, and Lars Wolf. "Analysis of factors affecting players' performance and perception in multiplayer games". In: *ACM SIGCOMM workshop on Network and system support for games*. 2005.

[5] Sebastian Egger et al. "Waiting times in quality of experience for web based services". In: *2012 Fourth International Workshop on Quality of Multimedia Experience*. IEEE. 2012, pp. 86–96.

[6] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[7] Esmail Asyabi et al. “CTS: An operating system CPU scheduler to mitigate tail latency for latency-sensitive multi-threaded applications”. In: *Journal of Parallel and Distributed Computing* (2018).

[8] Ashkan Paya and Dan C Marinescu. “Energy-aware load balancing and application scaling for the cloud ecosystem”. In: *IEEE Transactions on Cloud Computing* 5 (2017), pp. 15–27.

[9] Adam Belay et al. “The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane”. In: *ACM Transactions on Computer Systems (TOCS)* 34.4 (2017), p. 11.

[10] Yanpei Liu, Stark C Draper, and Nam Sung Kim. “SleepScale: runtime joint speed scaling and sleep states management for power efficient data centers”. In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 3. IEEE Press. 2014, pp. 313–324.

[11] David Lo et al. “Heracles: Improving resource efficiency at scale”. In: *ISCA*. 2015.

[12] Luiz André Barroso and Urs Hölzle. “The case for energy-proportional computing”. In: (2007).

[13] Arman Shehabi et al. “United states data center energy usage report”. In: (2016).

[14] David Lo et al. “Towards energy proportionality for large-scale latency-critical workloads”. In: *ISCA*. 2014.

[15] Daniel Wong. “Peak efficiency aware scheduling for highly energy proportional servers”. In: *ISCA*. 2016.

[16] *Enable up to 40% better price-performance with AWS Graviton2 based Amazon EC2 instances*. `https://pages.awscloud.com/rs/112-TZM-766/images/2020_0501-CMP_Slide-Deck.pdf`.

[17] Chang-Hong Hsu et al. “Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting”. In: *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE. 2015, pp. 271–282.

[18] Marek Rychly, Pavel Smrz, et al. “Scheduling decisions in stream processing on heterogeneous clusters”. In: *2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE. 2014, pp. 614–619.

[19] Jorge Manuel Gomes Barbosa and Belmiro Daniel Rodrigues Moreira. "Dynamic job scheduling on heterogeneous clusters". In: *2009 Eighth International Symposium on Parallel and Distributed Computing*. IEEE. 2009, pp. 3–10.

[20] *Cloud Load Balancing in Google*. `https://cloud.google.com/load-balancing`.

[21] Anshul Gandhi et al. "Autoscale: Dynamic, robust capacity management for multi-tier data centers". In: *ACM Transactions on Computer Systems (TOCS)* (2012).

[22] Anshul Gandhi et al. "Adaptive, Model-driven Autoscaling for Cloud Applications." In: *ICAC*. Vol. 14. 2014.

[23] *Amazon elastic container service*. `https://aws.amazon.com/ecs/`.

[24] Balajee Vamanan et al. "Timetrader: Exploiting latency tail to save datacenter energy for online search". In: *MICRO*. 2015.

[25] Harshad Kasture et al. "Rubik: Fast analytical power management for latency-critical systems". In: *MICRO*. 2015.

[26] George Prekas et al. "Energy proportionality and workload consolidation for latency-critical applications". In: *ACM Symposium on Cloud Computing*. 2015.

[27] Christian Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *ASPLOS*. 2014.

[28] Nagabhushan Chitlur et al. "QuickIA: Exploring heterogeneous architectures on real prototypes". In: *HPCA*. 2012.

[29] Brian Jeff. "Big. LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration". In: *Design Automation Conference (DAC)*. 2012.

[30] Jialin Li et al. "Tales of the tail: Hardware, os, and application-level sources of tail latency". In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–14.

[31] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: up and running: dive into the future of infrastructure.* " O'Reilly Media, Inc.", 2017.

[32] Brendan Burns et al. "Borg, omega, and kubernetes". In: (2016).

[33] Rajiv Nishtala et al. “Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads”. In: *HPCA*. 2017.

[34] Vinicius Petrucci et al. “Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers”. In: *HPCA*. 2015.

[35] Md E Haque et al. “Exploiting heterogeneity for tail latency and energy efficiency”. In: *MICRO*. 2017.

[36] *Odroid heterogeneous multi-core cluster and home cloud*. `https://magazine.odroid.com/article/odroid-hc1-and-odroid-mc1/`.

[37] Jason Mars and Lingjia Tang. “Whare-map: heterogeneity in homogeneous warehouse-scale computers”. In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 619–630.

[38] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *ACM European Conference on Computer Systems*. 2015.

[39] Christopher A Bohn and Gary B Lamont. “Load balancing for heterogeneous clusters of PCs”. In: *Future Generation Computer Systems* 18.3 (2002), pp. 389–400.

[40] Thomas Decker, Reinhard Lüling, and Stefan Tschöke. “A distributed load balancing algorithm for heterogeneous parallel computing systems”. In: (1998).

[41] Faraz Ahmad et al. “Tarazu: optimizing mapreduce on heterogeneous clusters”. In: *ACM SIGARCH Computer Architecture News* 40.1 (2012), pp. 61–74.

[42] Matei Zaharia et al. “Improving MapReduce performance in heterogeneous environments.” In: *Osdi*. Vol. 8. 4. 2008, p. 7.

[43] Zacharia Fadika et al. “MARLA: MapReduce for heterogeneous clusters”. In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE. 2012, pp. 49–56.

[44] Alexey Tumanov et al. “TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 35.

[45] Richard S Sutton, Andrew G Barto, Francis Bach, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[46] Xu Zhang et al. "E2E: embracing user heterogeneity to improve quality of experience on the web". In: *SIGCOMM'19: Proceedings of the ACM Special Interest Group on Data Communication* (2019).

[47] Joe Mambretti, Jim Chen, and Fei Yeh. "Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn)". In: *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*. IEEE. 2015, pp. 73–79.

[48] Kate Keahey et al. "Lessons Learned from the Chameleon Testbed". In: *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, 2020.

[49] Adam Kerin. *Introducing the Snapdragon 810 and 808 Processors: The Ultimate Connected Computing Experience*. 2014.

[50] Eitan Zemel. "The linear multiple choice knapsack problem". In: *Operations Research* 28.6 (1980), pp. 1412–1423.

[51] Fei Li et al. "Efficient and scalable IoT service delivery on cloud". In: *2013 IEEE sixth international conference on cloud computing*. IEEE. 2013, pp. 740–747.

[52] Jakub Dolezal, Zdenek Becvar, and Tomas Zeman. "Performance evaluation of computation offloading from mobile device to the edge of mobile network". In: *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE. 2016, pp. 1–7.

[53] Vytautas Valancius et al. "Greening the internet with nano data centers". In: *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM. 2009, pp. 37–48.

[54] Malte Schwarzkopf et al. "Omega: flexible, scalable schedulers for large compute clusters". In: *ACM European Conference on Computer Systems*. 2013.

[55] Hailong Yang et al. "PowerChief: Intelligent Power Allocation for Multi-Stage Applications to Improve Responsiveness on Power Constrained CMP". In: *ISCA*. 2017.

[56] Ali M Alakeel. "A guide to dynamic load balancing in distributed computer systems". In: *International Journal of Computer Science and Information Security* (2010).

[57] Martin Randles, David Lamb, and Azzelarabe Taleb-Bendiab. "A comparative study into distributed load balancing algorithms for cloud computing". In: *IEEE Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2010.

[58] Vinod Kumar Vavilapalli et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–16.

[59] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[60] *Apache Lucene*. `http://lucene.apache.org/`.

[61] *Lucene Nightly Benchmarks*. `https://home.apache.org/~mikemccand/lucenebench/`.

[62] Robert Schöne, Daniel Molka, and Michael Werner. "Wake-up latencies for processor idle states on current x86 processors". In: *Computer Science-Research and Development* (2015).

[63] Abdelhafid Mazouz et al. "Evaluation of CPU frequency transition latency". In: *Computer Science-Research and Development* (2014).

[64] Etienne Le Sueur and Gernot Heiser. "Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns". In: *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*. HotPower'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–8. URL: `http://dl.acm.org/citation.cfm?id=1924920.1924921`.

[65] Chia-Ming Wu, Ruay-Shiung Chang, and Hsin-Yu Chan. "A green energy-efficient scheduling algorithm using the DVFS technique for cloud datacenters". In: *Future Generation Computer Systems* 37 (2014), pp. 141–147.

[66] Nikzad Babaii Rizvandi, Javid Taheri, and Albert Y Zomaya. "Some observations on optimal frequency selection in DVFS-based energy consumption minimization". In: *Journal of Parallel and Distributed Computing* 71.8 (2011), pp. 1154–1164.

[67] Zhuo Tang et al. "An energy-efficient task scheduling algorithm in DVFS-enabled cloud environment". In: *Journal of Grid Computing* 14.1 (2016), pp. 55–74.

[68] Young Choon Lee and Albert Y Zomaya. "Energy efficient utilization of resources in cloud computing systems". In: *The Journal of Supercomputing* 60.2 (2012), pp. 268–280.

[69] Sukhpal Singh and Inderveer Chana. "Resource provisioning and scheduling in clouds: QoS perspective". In: *The Journal of Supercomputing* 72.3 (2016), pp. 926–960.

[70] Longxin Zhang et al. "Maximizing reliability with energy conservation for parallel task scheduling in a heterogeneous cluster". In: *Information Sciences* 319 (2015), pp. 113–131.

[71] Christina Delimitrou and Christos Kozyrakis. "Hcloud: Resource-efficient provisioning in shared cloud systems". In: *ASPLOS*. 2016.

[72] Rodrigo N Calheiros et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and experience* 41.1 (2011), pp. 23–50.

[73] M Aater Suleman et al. "Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp". In: *MICRO*. 2012.

[74] Francisco Romero and Christina Delimitrou. "Mage: Online Interference-Aware Scheduling in Multi-Scale Heterogeneous Systems". In: *arXiv preprint arXiv:1804.06462* (2018).

[75] Sukhpal Singh and Inderveer Chana. "A survey on resource scheduling in cloud computing: Issues and challenges". In: *Journal of grid computing* (2016).

[76] E Del Sozzo et al. "Workload-aware power optimization strategy for asymmetric multiprocessors". In: *DATE*. EDA Consortium. 2016.

[77] Mehmet Demirci. "A survey of machine learning applications for energy-efficient resource management in cloud computing environments". In: *International Conference on Machine Learning and Applications (ICMLA)*. 2015.

[78] Jason Mars, Lingjia Tang, and Robert Hundt. "Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity". In: *IEEE Computer Architecture Letters* 10.2 (2011), pp. 29–32.

[79] Qi Zhang et al. "Dynamic heterogeneity-aware resource provisioning in the cloud". In: *IEEE transactions on cloud computing* 2.1 (2014), pp. 14–28.

[80] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. "Exploiting cloud heterogeneity to optimize performance and cost of MapReduce processing". In: *ACM SIGMETRICS Performance Evaluation Review* 42.4 (2015), pp. 38–50.

[81] Wei Chen, Jia Rao, and Xiaobo Zhou. "Addressing performance heterogeneity in mapreduce clusters with elastic tasks". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 1078–1087.

[82] Gunho Lee and Randy H Katz. "{Heterogeneity-Aware} Resource Allocation and Scheduling in the Cloud". In: *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11)*. 2011.

[83] Zhonghong Ou et al. "Exploiting Hardware Heterogeneity within the Same Instance Type of Amazon {EC2}". In: *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*. 2012.

[84] *CPU Benchmark*. `https://www.cpubenchmark.net/`.

[85] *CPU Benchmarks and Hierarchy*. `https://www.tomshardware.com/reviews/cpu-hierarchy,4312.html`.

[86] *CPU Energy Efficiency Trend*. `https://www.electronicdesign.com/technologies/microprocessors/article/21802037/energy-efficiency-of-computing-whats-next`.

[87] Rathijit Sen and David A Wood. "Pareto Governors for Energy-Optimal Computing". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.1 (2017), p. 6.

[88] Chih-Hsun Chou, Laxmi N Bhuyan, and Daniel Wong. "μDPM: Dynamic Power Management for the Microsecond Era". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 120–132.

[89] EC Amazon. *Auto Scale. Amazon Web Services, Inc.* 2013.

[90] *Amazon Elastic Load Balancer*. `https://aws.amazon.com/elasticloadbalancing/`.

[91] Luiz André Barroso. "The price of performance". In: *Queue* 3.7 (2005), pp. 48–53.

[92] David Meisner et al. "Power management of online data-intensive services". In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 3. ACM. 2011, pp. 319–330.

[93] Osman Sarood et al. “Maximizing throughput of overprovisioned hpc data centers under a strict power budget”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2014, pp. 807–818.

[94] Harshad Kasture and Daniel Sanchez. “Ubik: efficient cache sharing with strict qos for latency-critical workloads”. In: *ACM SIGPLAN Notices*. Vol. 49. 4. ACM. 2014, pp. 729–742.

[95] Haishan Zhu and Mattan Erez. “Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems”. In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 33–47.

[96] Mateusz Guzek, Dzmitry Kliazovich, and Pascal Bouvry. “HEROS: Energy-efficient load balancing for heterogeneous data centers”. In: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE. 2015, pp. 742–749.

[97] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2017, pp. 1–12.

[98] Andrew Putnam et al. “A reconfigurable fabric for accelerating large-scale datacenter services”. In: *MICRO* (2015).

[99] Quan Chen et al. “Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers”. In: *ACM SIGPLAN Notices*. Vol. 51. 4. ACM. 2016, pp. 681–696.

[100] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware scheduling for heterogeneous datacenters”. In: *ACM SIGPLAN Notices*. Vol. 48. 4. ACM. 2013, pp. 77–88.

[101] Konstantinos Karanasos et al. “Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters.” In: *USENIX Annual Technical Conference*. 2015, pp. 485–497.

[102] Kay Ousterhout et al. “Sparrow: distributed, low latency scheduling”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 69–84.

[103] Robert Grandl et al. “Graphene: Packing and dependency-aware scheduling for data-parallel clusters”. In: *Proc. of the USENIX OSDI*. 2016.

[104] Mosharaf Chowdhury et al. "HUG: Multi-resource fairness for correlated and elastic demands". In: *USENIX NSDI*. 2016.

[105] Huazhe Zhang and Henry Hoffmann. "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 545–559. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872375. URL: http://doi.acm.org/10.1145/2872362.2872375.

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

[107] Yunqi Zhang et al. "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference". In: *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE. 2016, pp. 456–468.

[108] Etienne Le Sueur and Gernot Heiser. "Dynamic voltage and frequency scaling: The laws of diminishing returns". In: *Proceedings of the 2010 international conference on Power aware computing and systems*. 2010, pp. 1–8.

[109] Dror G Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[110] Nathaniel Pinckney et al. "Shortstop: An on-chip fast supply boosting technique". In: *VLSI Circuits (VLSIC), 2013 Symposium on*. IEEE. 2013, pp. C290–C291.

[111] DPDK Intel. *Data plane development kit*. 2014.

[112] P Lalith Suresh et al. "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection." In: *NSDI*. 2015, pp. 513–527.

[113] Rishi Kapoor et al. "Chronos: predictable low latency for data center applications". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012, p. 9.

[114] Ivano Cerrato, Mauro Annarumma, and Fulvio Risso. "Supporting Fine-Grained Network Functions through Intel DPDK." In: *EWSDN*. 2014, pp. 1–6.

[115] Keon Jang et al. “Silo: Predictable message latency in the cloud”. In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 435–448.

[116] Cheng Li et al. “Quantifying and improving i/o predictability in virtualized systems”. In: *2013 IEEE/ACM 21st International Symposium on Quality of Service*. IEEE. 2013, pp. 1–6.

[117] Geoffrey Blake and Ali G Saidi. “Where does the time go? characterizing tail latency in memcached”. In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE. 2015, pp. 21–31.

[118] J Michael Moore. “An n job, one machine sequencing algorithm for minimizing the number of late jobs”. In: *Management science* 15.1 (1968), pp. 102–109.

[119] Jeffrey C Mogul and Ramana Rao Kompella. “Inferring the network latency requirements of cloud tenants”. In: *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*. 2015.

[120] Yanfang Le et al. “PL2: Towards Predictable Low Latency in Rack-ScaleNetworks”. In: *arXiv preprint arXiv:2101.06537* (2021).

[121] David E Irwin, Laura E Grit, and Jeffrey S Chase. “Balancing risk and reward in a market-based task service”. In: *Proceedings. 13th IEEE International Symposium on High performance Distributed Computing, 2004.* IEEE. 2004, pp. 160–169.

[122] Brent N Chun and David E Culler. “User-centric performance analysis of market-based cluster batch schedulers”. In: *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID’02)*. IEEE. 2002, pp. 30–30.

[123] *CDN Latency Test*. URL: `https://www.cdnperf.com/tools/cdn-latency-benchmark`.

[124] Toke Høiland-Jørgensen et al. “Measuring latency variation in the internet”. In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 2016, pp. 473–480.

[125] Brandon Schlinker et al. “Internet performance from facebook’s edge”. In: *IMC*. 2019, pp. 179–194.

[126] Radhika Mittal et al. “TIMELY: RTT-based congestion control for the datacenter”. In: *ACM SIGCOMM* 45.4 (2015), pp. 537–550.

[127] Xiaoqi Chen et al. "Measuring tcp round-trip time in the data plane". In: *Proceedings of the Workshop on Secure Programmable Network Infrastructure*. 2020, pp. 35–41.

[128] Cloud Native Computing Foundation (CNCF). *CNCF SURVEY 2019*. 2020. URL: `https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf`.

[129] *Linux IP Virtual Server*. `https://en.wikipedia.org/wiki/IP_Virtual_Server`.

[130] Michael Pinedo. *Scheduling*. Vol. 29. Springer, 2012.

[131] Gaurav R Ghosal et al. "A Deep Deterministic Policy Gradient Based Network Scheduler For Deadline-Driven Data Transfers". In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 253–261.

[132] Yujing Wang and Darrel Ma. "Developing a Process in Architecting Microservice Infrastructure with Docker, Kubernetes, and Istio". In: *arXiv preprint arXiv:1911.02275* (2019).

[133] Aditya Venkataraman and Kishore Kumar Jagadeesha. "Evaluation of inter-process communication mechanisms". In: *Architecture* 86 (), p. 64.

[134] C Martin, J Miller, and KENT Pearce. "Numerical solution of positive sum exponential equations". In: *Applied mathematics and computation* 34.2 (1989), pp. 89–93.

[135] Stephen Hemminger et al. "Network emulation with NetEm". In: *Linux conf au*. 2005, pp. 18–23.

[136] Jing Li et al. "Work stealing for interactive services to meet target latency". In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. 2016, p. 14.

[137] Keqiang He et al. "Presto: Edge-based load balancing for fast datacenter networks". In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 465–478.

[138] Abdul Kabbani et al. "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM. 2014, pp. 149–160.

[139] David Zats et al. "DeTail: reducing the flow completion time tail in datacenter networks". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 139–150.

[140] Jonathan Perry et al. “Fastpass: A centralized zero-queue datacenter network”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 307–318.

[141] Christo Wilson et al. “Better never than late: Meeting deadlines in datacenter networks”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 50–61.

[142] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. “Deadline-aware datacenter tcp (d2tcp)”. In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 115–126.

[143] Qi Alfred Chen et al. “Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis”. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. 2014, pp. 151–164.

[144] Junchen Jiang et al. “Eona: Experience-oriented network architecture”. In: *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. 2014, pp. 1–7.

[145] P. Thinakaran et al. “Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 977–987.

[146] Jared Eugene Wray et al. *Predictive two-dimensional autoscaling*. US Patent 9,329,904. 2016.

[147] Ming Mao, Jie Li, and Marty Humphrey. “Cloud auto-scaling with deadline and budget constraints”. In: *2010 11th IEEE/ACM International Conference on Grid Computing*. IEEE. 2010, pp. 41–48.

[148] Jing Jiang et al. “Optimal cloud resource auto-scaling for web applications”. In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE. 2013, pp. 58–65.

[149] Hector Fernandez, Guillaume Pierre, and Thilo Kielmann. “Autoscaling web applications in heterogeneous cloud infrastructures”. In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, pp. 195–204.

[150] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. “Auto-scaling web applications in clouds: A taxonomy and survey”. In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–33.

[151] Rui Han et al. “Lightweight resource scaling for cloud applications”. In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE. 2012, pp. 644–651.

[152] Rajesh Nishtala et al. “Scaling Memcache at Facebook.” In: *nsdi*. Vol. 13. 2013, pp. 385–398.

[153] *AWS Predictive Autoscaling*. `https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-predictive-scaling-for-amazon-EC2-in-aws-auto-scaling/`.

[154] Junliang Chen et al. “Tradeoffs between profit and customer satisfaction for service provisioning in the cloud”. In: *Proceedings of the 20th international symposium on High performance distributed computing*. 2011, pp. 229–238.

[155] Caesar Wu, Rajkumar Buyya, and Kotagiri Ramamohanarao. “Modeling cloud business customers’ utility functions”. In: *Future Generation Computer Systems* 105 (2020), pp. 737–753.

[156] Salman A Baset. “Cloud SLAs: present and future”. In: *ACM SIGOPS Operating Systems Review* 46.2 (2012), pp. 57–66.

[157] Damián Serrano et al. “SLA guarantees for cloud services”. In: *Future Generation Computer Systems* 54 (2016), pp. 233–246.