Undergraduate Thesis

PERFORMANCE EVALUATION OF DIFFERENT MAPPINGS FOR CLOUD-BASED MICROSERVICES

Christodoulos Constantinou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2024

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Performance evaluation of different mappings for cloud-based microservices

Christodoulos Constantinou

Supervising Professor

Dr. Yanos Sazeides

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor's in Computer Science from the University of Cyprus

May 2024

Acknowledgments

I would like to express my sincere appreciation to all the people that played a crucial role in the procedure of completing this thesis.

Firstly, I would like to thank my supervisor Dr. Yanos Sazeides for his continuous constructive feedback and important research knowledge insights, which have played a major role in shaping the trajectory of my research throughout two semesters.

Additionally, I would like to thank Georgia Antoniou, a PhD student at the University of Cyprus, for her constant help and contribution in the technical aspects and implications encountered in the procedure of completing this thesis.

Lastly, I would like to express my heartfelt gratitude towards my family and friends for supporting me throughout my whole academic journey at the University of Cyprus.

Abstract

Over the past decade, cloud computing powered by the usage of microservices software architecture, has transformed application development from the traditional monolithic approach. Many major corporations have transitioned to the usage of microservices, aiming to leverage their flexibility, scalability and efficiency to improve their application development process and overall Quality of Service. However, despite their benefits, microservices also face several implications.

In this thesis I explore how the cluster size and the mapping of microservices to nodes can affect the performance and the power consumption of a microservice-based latency-critical application. The benchmark application chosen for the experiments of this thesis is one of the five end-to-end services developed as part of the DeathStarBench benchmark by the SAIL group at Cornell University [1]. My research reveals that deployment strategies significantly impact both application performance and power efficiency. For instance, while increasing cluster size does not always guarantee better QoS, strategic node mapping can improve latency and manage power consumption effectively. These findings highlight the necessity of balancing trade-offs in deployment strategies to enhance both power efficiency and application performance.

Contents

Chapter 1

Introduction

1.1 Problem

As more technology companies transition from a monolithic approach to building their applications to a microservices-based one, it has become crucial to study and evaluate this approach. Consequently, there is a need to explore the capabilities and limitations of a system that utilizes microservices, and to determine how to achieve maximum efficiency. The primary objective of this study is to examine how different mappings of microservices to nodes, can impact the performance and the power consumption of a microservice-based application.

To achieve this objective, we first investigate the functionalities and characteristics of the microservice-based benchmark used in this study. This is crucial in understanding later the results of the experimental evaluation. Once we gain a general understanding of the application, the microservice software paradigms and the hardware configuration of the evaluated system we move onto experimental evaluation. In the experimental evaluation we investigate the effects of the cluster size and the mapping of microservices to nodes on the performance of the benchmark and the power consumption of the system. Besides performance metrics such as average response time and 99th tail latency, we also collect C-state related metrics and the machine power consumption in order to gain a better understanding of node utilization.

1.2 Contribution

This thesis contributes to the understanding of microservices architecture in the context of latency-critical systems. Other papers such as ‘An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems’ [1] have already examined the performance of a microservices application and the machine utilization when deploying benchmark microservices application on a cluster of nodes. The experiments conducted in my work, further contribute to this, by also exploring all the different ways of distributing the various components and services of an application on multiple machines. Notably, my research found that while separating critical path services can introduce additional latency, strategic distribution of these services can optimize both application performance and power consumption.

At a more advanced advance research level, papers like Cost-efficient Management of Cloud Resources for Big Data Applications [2], decide their mapping strategy that aims to achieve better performance and efficiency, through the use of mathematical models, dynamic scheduling algorithms and reinforcement learning (RL) models. In my work the grouping and separation of services of the microservices benchmark application is based upon the functionality of each one, and the creation of different mappings is done through an exhaustive search which involves evaluating all possible solutions to find the optimal one.

Additionally, this work also aims to simplify and automate the process of deploying the benchmark application on cloud platform nodes, as well as of collecting, and presenting different metrics in plot form, so as to evaluate the performance of the benchmark and the utilization of the nodes. The deployment and collection of metrics is achieved through the usage of a repository consisting of a stable version of the benchmark microservices application, profiling tools, several deployment bash scripts, as well as data processing and plotting python scripts.

1.3 Outline

The contents of this paper consist of 8 chapters. Chapter 2 presents latency critical systems and metrics used to evaluate the performance of such systems. Chapter 3 analyses the architecture of a microservices application, as well as its key components and technologies used. It also highlights the advantages and implications of microservices and compares them to the

monolithic approach. Chapter 4 includes an overview of the DeathStarBench suite and explores the benchmark I used for my thesis called Social Network, in detail. Chapter 5 presents the cloud platform used to conduct my experiments, the setup and configuration of the machines used, and the metrics captured and presented in the result plots. Chapter 6 describes the mappings created and used for my experiments in detail, and their potential effect on key performance parameters. Chapter 7 presents the results from my experiments in plot form with the addition of a comparison between different mappings and observations, and conclusions derived from them. Chapter 8 includes related work done by other researchers and undergraduate students. Lastly, chapter 9 concludes the knowledge and findings found from my experiments and overall procedure of writing this paper and describes future work that can be done extending on the work done in this thesis.

Chapter 2

QoS and latency-critical system evaluation metrics

2.1 Latency-critical systems

Latency-critical systems are systems that have strict response time requirements, meaning that they must react quickly within a set time frame upon receiving input or a request. Some examples of such systems are web search engines, financial trading platforms, online gaming, and real time analytics.

Various metrics are used to evaluate the Quality of Service (QoS) and performance for these systems. The following 3 sections explain each metric in detail.

2.2 Average latency

Average latency, also known as the mean response time, is the arithmetic average of all response times for a given set of requests. This metric is important and helpful when we want to form a general idea of the typical response time for a system or application. However, the average latency values can be skewed by outliers or by a long tail in the distribution of response times. In order to evaluate a system in detail, and form a complete picture of the latency distribution, we often use percentiles such as P90 and P99. This metric is called tail latency.

2.3 Tail latency

Tail latency refers to the cases where the time of completion for a request or action is among the highest values within a given set of requests. These underperforming requests generally fall within the range of the 90th to the 99th percentile. Tail latency calculation involves measuring all request latencies and determining the latency of the slowest request at the 99th percentile or at any other percentile, depending upon what the system aims to achieve. This value is then checked to see if it falls within an acceptable range or if measures need to be taken to reduce it. Figure 1 [3], shows an example plot regarding percentile ranges for latency values. In this figure we can see, for example, that HAProxy Multi-Threaded has a value of 1000 milliseconds for 99th percentile tail latency. This means that 99% of requests will be served with response times within this time frame (< 1000ms), while the remaining 1% of requests will experience larger response times.

Maintaining low tail latency in real-life latency-critical systems is crucial, because it ensures that 99% of clients (or the majority depending on the percentile value) will be served within acceptable times. When we only consider average latency, we ensure that 50% of the clients experience low response times, while the remaining 50% possibly experience higher response time making their experience much worse. This makes tail latency the preferred metric used by software companies, when aiming to provide good QoS to the majority of their clients.

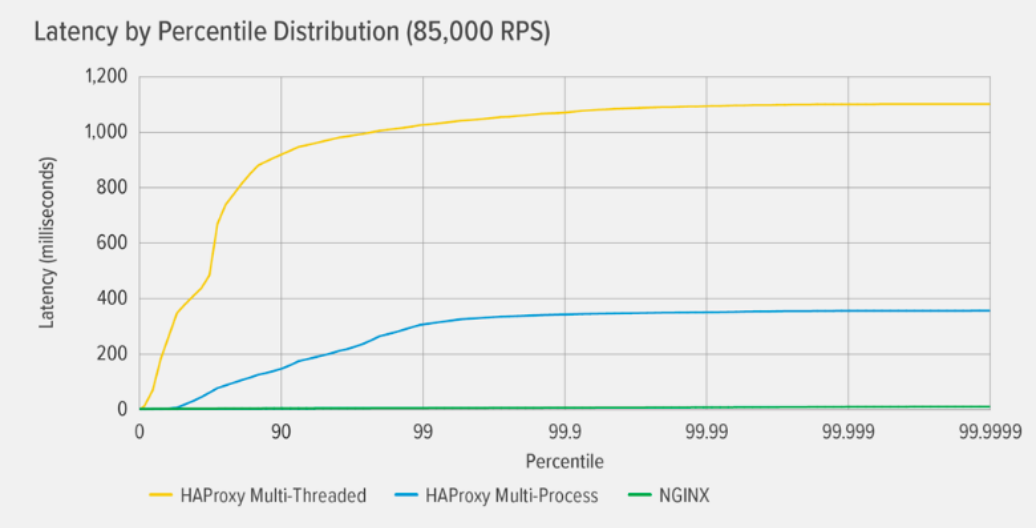

Figure 1 Percentile ranges for latency values [3]

Another important test for detecting underperforming requests in latency-critical systems is a strain test, which involves gradually increasing the system's workload until it reaches its capacity and can no longer respond. The workload in such systems is typically measured in queries per second (QPS), which represents the number of incoming requests arriving every second.

2.4 Executed and dropped requests

Limiting the underperforming requests to 1% or even below 1% can become very challenging especially in latency-critical systems that experience large amounts of request traffic. Delayed requests typically occur due to limited resources, network delays or software issues.

Except from longer response times for a percentage of requests, subsequent failure can occur in the form of requests being dropped, meaning some requests will not be executed at all. When a system experiences higher than expected request traffic and requests are dropped, QoS and the reputation of a system is degraded even more than higher latency values. In our experiment analysis, we present metrics correlated with a range of QPS values, including a small range of higher QPS values where the system begins to drop requests. These instances are shown solely to determine the exact QPS threshold at which the system starts dropping requests. Consequently, these cases are not used for evaluation purposes, as the behaviour in such scenarios is unstable and unpredictable, preventing us from drawing accurate conclusions from the results.

Chapter 3

Microservices advantages and implications

3.1 Microservices software architecture

In modern software architecture, microservices represent an innovative approach aimed at decomposing an application into a series of small services. Each microservice is specifically implemented to fulfill a specific functionality of the system. With this approach each microservice can be individually developed, deployed, and scaled. Figure 2 [4] is a typical microservices application structure, illustrating its components, including clients, API gateway, containers, microservices and the data storage.

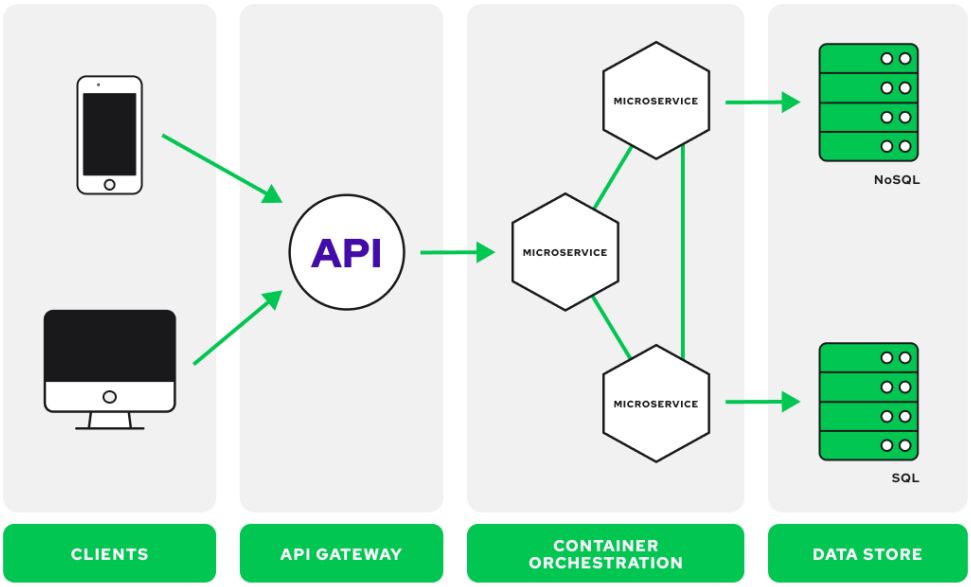

Figure 2 Microservices application structure [4]

Using containerization technologies, we can provide each microservice with its own runtime, libraries, and dependencies by isolating the microservice in its own container. Given that each microservice is isolated from the others, the microservices cannot directly communicate with each other. Communication is achieved through the use of an API (Application Programming Interface) and RPCs (Remote Call Procedures).

3.1.1 REST API

REST APIs, or Representational State Transfer Application Programming Interfaces, are a type of software interface that enables web applications to communicate with each other over the internet. They use HTTP requests such as GET, POST, PUT, DELETE and typically JSON (JavaScript Object Notation) as a format for data transfer. REST APIs are stateless, meaning they do not store any data regarding the client session on the server. All the information needed to execute a request is contained within the request itself from the client to the server. APIs are widely used due to their simplicity and scalability.

In the context of microservices software architecture, REST APIs play a crucial role in facilitating communication between different microservices. More specifically, each microservice shows its functionalities through a REST API, allowing other services to access them and use them. This allows each microservice to remain independent and loosely coupled.

3.1.2 Remote Procedure Call

Remote Procedure Call (RPC) is a communication protocol used in distributed systems with the purpose of allowing a program to cause a subroutine or procedure to execute in another address space as if it were a normal local procedure call. This can be done without the programmer having to manually code the details for the remote interaction. RPCs are based on a client-server model, where a client makes a request, and the server acts based on the requests. With this abstraction, developers can design their application in a more modular way and maintain communication between different components. There are two types of RPCs: asynchronous and synchronous. In synchronous RPC, the client waits for the server to respond before continuing its execution, while in asynchronous RPC, the client continues its execution without waiting for a response from the server.

RPCs play a crucial role in the microservices software architecture in terms of facilitating communication between the various microservices. As explained before the microservices don't have direct communication with each other. Using RPCs, communication is made possible through the exchange of structured messages over the network. The difference between RPCs and REST APIs is that RPCs are more commonly used for communication of the microservices within the same system, whereas REST APIs are mostly used for external communication.

3.1.3 Docker

Docker is an open source, widely used platform that enables developers to develop, deploy and run applications with containers without having to manage infrastructure. Docker can be used via the command line interface (CLI) in a Linux terminal or through a GUI desktop application. Containers are lightweight, standalone, executable packages that include the necessary components to run an application, which are the code, runtime, system tools and system libraries. Unlike traditional virtual machines, containers share the host system’s kernel and do not require a separate operating system making them more efficient and lightweight.

This makes docker suitable for use in the deployment and scaling of microservices. By encapsulating each microservice in an individual container it provides it with its own runtime, libraries, and dependencies thus increasing portability and scalability of the microservices. Additionally, docker eliminates the ‘it works on my machine’ problem often encountered in

software development and deployment, as it can easily be deployed across different environments.

3.1.4 Docker Swarm

Docker Swarm is a tool within docker. The added features compared to Docker as described in the previous section, is that it allows developers to manage a cluster of hosts or nodes, while treating them as a single virtual Docker host. A swarm is a collection of nodes that contain one manager node and several worker nodes. The manager node assigns a different workload or group of services to each worker node and in turn the worker nodes execute their assigned workload. This makes it ideal for usage in the deployment of services on multiple machines.

In the context of this thesis, Docker Swarm was used to create different configurations for the deployment of services from the microservices benchmark application on multiple machines.

3.2 Microservices advantages

Microservices have emerged as a preferred architectural approach due to the several advantages they offer. They achieve good scalability, due to the architecture allowing individual components to be scaled independently, making it easier to manage resources efficiently and handle different levels of workloads. Additionally, due to their flexibility and agility, microservices allow faster development and deployment cycles making updates and modifications easier. Another significant advantage is fault isolation. Because the services are designed to operate independently, identifying, and isolating faults in a given system is much easier. This means that failure in one microservice does not necessarily impact the system, ensuring higher availability and reliability. Furthermore, microservices allow for the use of different technologies and programming languages or frameworks, enabling development teams to choose the most suitable technology for creating each microservice and its functionalities. Lastly, DevOps methods are promoted through the use of microservices by facilitating continuous delivery and integration, reducing time to market and improving overall software quality.

3.3 Microservices implications

Even though microservices offer many benefits, they also face certain challenges and implications. As the number of services is very large, microservices introduce a higher level of complexity, especially in terms of deployment, monitoring, and debugging. Managing all the microservices requires additional effort and resources. Furthermore, communication between all the microservices can introduce overhead latency to the system. Another challenge microservices face is the difficulty of ensuring data consistency across multiple services, due to each service having its own storage and caching mechanisms. Overall, developing and maintaining a microservices-based application requires additional effort, as developers need to coordinate the interaction between different services and ensure consistency and compatibility.

3.4 Comparison of monolithic and microservices

Although microservices is the preferred architecture chosen over the traditional monolithic, due to the several advantages they offer, as presented in section 3.2, the monolithic software architecture still offers some significant benefits. Because all parts of the application are contained within a single codebase containing the user interface, business logic and data access, the monolithic approach offers simplicity, ease of deployment and ease of testing. Lastly, Quality of Service restrictions are not as strict in monolithic architecture compared to microservices, where communication between the services is the dominant factor, introducing additional latency. For example, if we consider a monolithic application that has QoS constraints for the 99^{th} percentile tail latency being under 10 milliseconds. Implementing an equivalent microservices application consisting of 10 different microservices, will require each of the 10 services to have tail latency under 1 millisecond to maintain the overall tail latency of the application under 10 milliseconds. However, the value of 1 millisecond tail latency for each microservice, does not account for the additional overhead latency introduced by the communication between these 10 microservices, which in most cases are deployed on different nodes. Overall, when developing a monolithic application, overhead latency introduced by communication is not as significant as in the microservices architecture, and meeting the same QoS restrictions becomes more difficult and complex when using a microservice architecture to implement an equivalent application.

Chapter 4

DeathStarBench Benchmark suite

4.1 DeathStarBench Overview

The DeathStarBench suite, introduced in the associated paper [1], is an open-source benchmark suite designed for microservices, developed by the SAIL group at Cornell University. For its implementation open-source applications and technologies are used, including NGINX, Memcached, MongoDB and MySQL. Most of the source code is built upon these technologies and the connection between the microservices is done using Apache Thrift, gRPC, or HTTP requests. Utilizing the flexibility of microservices, the DeathStarBench applications and their services use different programming languages for their implementation, including C/C++, Java, JavaScript, Python, Ruby, Go, Lua, and Scala. The benchmark applications offered in the suite cover the full functionality of a real world used system of the same nature and have the necessary components such as Frontend and load balancing, microservices logic, and databases and caching. These benchmarks extend beyond the typical 2-tier benchmarks of direct communication between server and client, meaning that they enter more complex call flows.

4.2 Social Network architecture and technologies

The Social Network includes core functionalities of a real-world social network such as Facebook and Twitter. The functionalities include user creation, following other users, creating posts, and viewing posts created by other users. In total, the Social Network has 36 different services, with each service having its own container. The load balancing of HTTP requests from the users is handled by NGINX and the storage and caching by MongoDB, Memcached, and Redis. More specifically, Memcached acts as specific caching for each microservice, and MongoDB stores all the relevant data for the functionalities that each microservice implements. The microservices use php-fpm for communication, while for inter-service communication Apache Thrift RPCs are used. Additionally, the benchmark uses Cassandra and Jaeger to capture data that represents traces for the requests of the users. Figure 3 below [1], illustrates the architecture of the Social Network and the dependencies between the microservices.

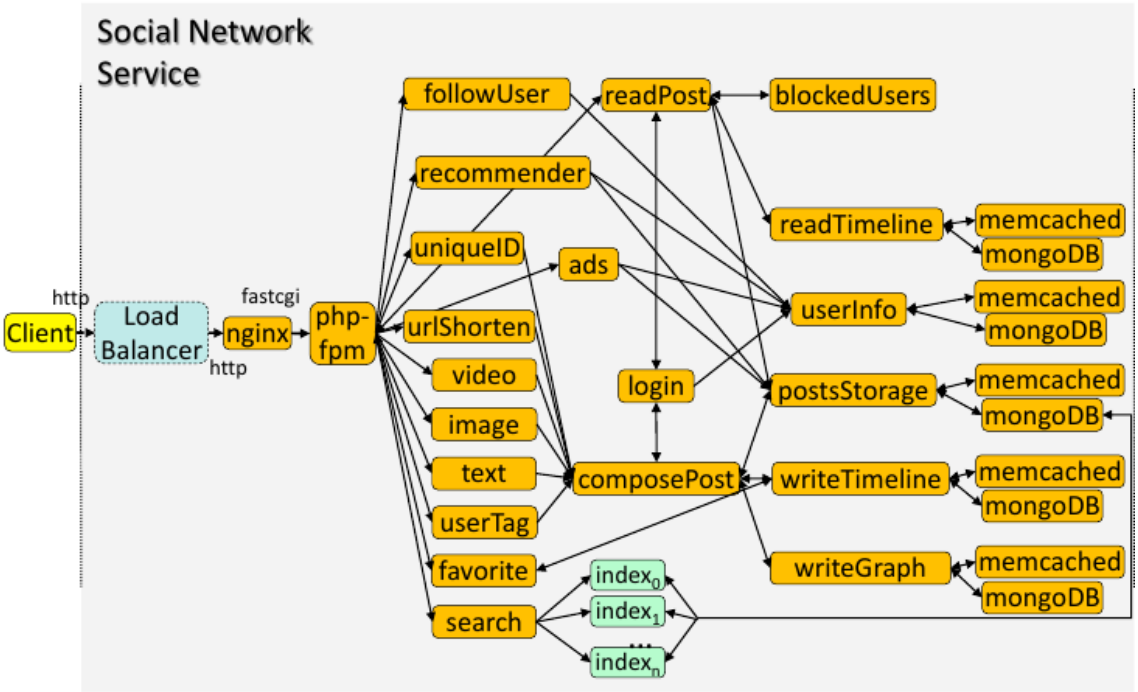

Figure 3 Social Network Architecture and Dependency Graph [1]

The number of users for the Social Network is determined by the choice from the three available user datasets available in the benchmark. The three datasets are distinguished by their sizes which are small, medium, and large. These varying dataset sizes allow users and researchers of the benchmark to examine the behaviour and the performance of the application based on the number of users created. The small dataset, which is the one used for the

experiments conducted in this thesis, includes 962 nodes which represent the number of users, and 18 800 edges which represent the connections between each user. The medium dataset contains 81,306 nodes and 1,768,149 edges while the large user dataset consists of 456,000 nodes and 835 400 edges.

4.3 Social Network workloads

Besides allowing the user of the benchmark to select from varying sizes of user datasets, the Social Network also offers several workloads with major differences between them. The workloads are Compose-Post, Read Home-Timeline, Read User-Timeline and Mixed-Workload. The workloads for each functionality can be generated by running Lua scripts implemented in the source code of the benchmark. The inputs to the scripts, provided by the user of the benchmark, define the duration and the QPS for running a specific workload. The following subsections present each of these scripts in detail.

4.3.1 Compose-Post

This script generates random content and metadata, designed to simulate the process of creating a post on a social media platform. More specifically, to achieve this the script first sets up random seed values and defines a character set used later for generating random strings that represent the text contents of the post. Then it selects a user index that represents the account of the user creating the post; based on the selected index it generates a username and a user ID. The post can also include user mentions, URLs, and other forms of media content. The number of appearances for each post element is determined randomly by the script, with the upper limit being 5 for mentions and URLs and 4 for the media content. Finally, to simulate the action of the user submitting a post on the platform, an HTTP POST request is created, which consists of the username, user ID, text content, media IDs, media types and post type. The number of times this procedure is repeated is based upon the input given to the script which, as described above, is the QPS and the duration which the workload will run for.

4.3.2 Read Home-Timeline

This script aims to emulate the client making requests to read the home timeline of a user. Initially, it selects the user ID that represents the user whose home timeline will be accessed. The ID is selected randomly in the range of 1 to 962. Once the user is determined and selected, the script then selects a post from the first 100 posts of the selected user and sets that post as the start of the range of posts that will be retrieved. The range for the retrieved posts stops after 10 consecutive posts starting from the selected post. Then to finalize, an HTTP GET request is constructed containing the information selected previously by the script to retrieve the posts from the user that has been selected.

4.3.3 Read User-Timeline

Read User-Timeline works in the same way as the Read Home-Timeline script with the key difference being that the posts being read are retrieved from the own user's timeline and not from other users' posts appearing in the selected user's home timeline. All the experiments conducted in this thesis use the Read User-Timeline script to create the workload.

4.3.4 Mixed Workload

With the Mixed Workload script, the workload generated from it consists of a combination of the actions described in the previous three scripts. To decide the type of each request, ratios representing the probability for each of the three scripts described above have been assigned. The probability for Compose Post is set to 0.1 (10%), for Read Home-Timeline to 0.6 (60%) and for Read User-Timeline to 0.3 (30%). These probability weights are not randomly set; they are statically assigned and remain the same with each run. However, the randomized nature of selecting posts and users within the procedures of each of the three scripts remains the same as described in the individual sections for each of the scripts.

Chapter 5

Experimental methodology

5.1 Establishing a cloud experimentation environment

In order to run my experiments, I had to gain access to a cloud platform. The platform I used is called Cloud Lab [4] which enables users to conduct experiments and research by connecting to the various nodes offered, through a SSH connection. The physical nodes are divided into six different clusters, Utah, Clemson, Wisconsin, Apt, Massachusetts and Emulab. The user has the option to choose from any of the physical nodes offered in each cluster. Cloud Lab also allows the user to create experiments using pre-defined profiles created by the user or profiles offered from the project the user is a member of. The profiles contain the desired experiment parameters, such as the type of nodes, the topology, OS version, desired pre-installed libraries and packages, and any other procedures the user wants to always execute when instantiating an experiment. Figure 4 [5] showcases the setup of a Cloud Lab experiment.

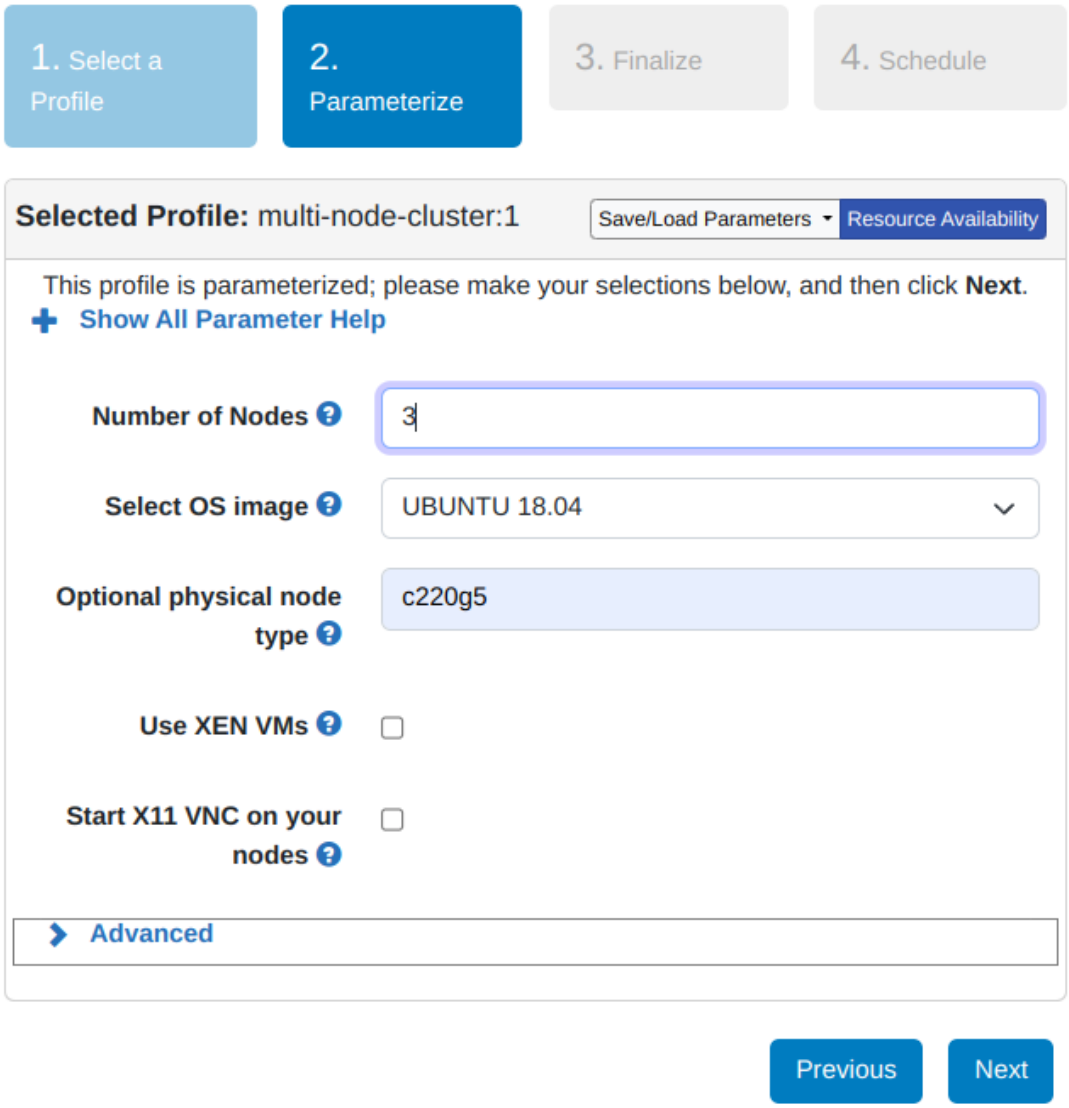

Figure 4 Cloud Lab experiment setup [5]

After completing the setup and creation of the experiments the user is provided with a SSH command for each of the nodes that have been reserved, to connect to the machines through a UNIX command line.

The platform has a strict policy on experiment duration. It limits the duration to 16 hours and if users wish to run an experiment for longer than this, they have to a request for an extension that must include an explanation for the request to be approved. Due to the limited experiment duration, I had to utilize the scp UNIX command to transfer the experiment results to my local Linux machine.

The nodes chosen to run my experiments are called c220g5 and are part of the Wisconsin cluster. The c220g5 machines have the following hardware specifications [5]: two Intel Xeon Silver 4114 10-core CPUs running at 2.20 GHz, 192GB ECC DDR4-2666 Memory, one 1 TB 7200 RPM 6G SAS HD, and one Intel DC S3500 480 GB 6G SATA SSD. Networking capabilities include a dual-port Intel X520-DA2 10Gb NIC and an onboard Intel i350 1Gb NIC.

5.2 Benchmark setup

To setup the Social Network benchmark application on the Cloud Lab nodes I had to clone the repository I maintained that includes a stable version of the application, setup and deployment bash scripts, python plotting scripts and profiling tools. The public repository can be found on GitHub: https://github.com/cconst10/Ptix-Fork , and it contains a README file called SetupAndRunREADME.md that explains the specific technical details of the steps needed to setup the benchmark and run the experiments. In the setup of the benchmark application one of the reserved nodes always acts as a client. By running the script called baseline.sh on the client node the following actions are executed on all the other nodes: which are cloning the repository, installing required language versions, libraries, packages, and configuring machine characteristics. Additionally, through this script other deployment scripts are executed. Specifically, for each experiment the baseline.sh script was modified to call a different deployment script based on the nature of the experiment, for example deployment of the Social Network using Docker Compose or deployment using Docker Swarm. The input parameter of baseline.sh is the number of nodes minus 1 to exclude the node acting as the client. Once the script is finished executing, the application is setup, and the machine is configured to the desired baseline characteristics that will be presented in detail in the following section.

5.3 Baseline configuration

For all the experiments in this thesis the machine configuration remained the same. The baseline configuration is as follows:

5.3.1 C-states

The first power saving technology altered in my baseline configuration are the C-states. C-states serve as power-saving modes integrated into modern processors to reduce power usage during periods of inactivity. These modes utilize techniques such as Dynamic Voltage Frequency Scaling (DVFS) and power gating to optimize energy efficiency. DVFS dynamically adjusts voltage and frequency, while power gating selectively disables power to inactive cores or components. For instance, Intel's Skylake architecture, which is the architecture that the processors of the c220g5 nodes I used for my experiments are built on, [6] incorporates various C-states, including C0, C1, C1E, and C6, each offering different levels of power-saving capabilities. The C0 state represents full operational mode, where cores are actively processing tasks. In contrast, C1 and C1E signify low-power idle states, with C1E facilitating quicker transitions to active mode. Meanwhile, the C6 state represents a deeper sleep mode, shutting off power to cores to achieve substantial energy savings. However, transitioning in and out of C6 can introduce latency, impacting system responsiveness, which is crucial especially in applications with strict processing requirements such as the benchmark used in this thesis. Nevertheless, the power gating mechanism used in C-states plays a pivotal role in reducing power consumption, thereby enhancing overall energy efficiency in computing systems.

For my experiments, only C0 and C1 where enabled, in order for the processor to be able to be fully operational when the system experienced more demanding benchmark workloads, but also to enter a less operational and more power saving state under less demanding workloads.

5.3.2 Un-core frequency and DVFS

The base frequency of the Intel Xeon Silver stayed the same at 2.2 GHz, but un-core frequency of the processor was set to 2GHz and Intel’s turbo boost frequency altering technology was disabled. Turbo boost is a specific implementation of DVFS for Intel processors. DVFS dynamically adjusts the voltage and frequency of a processor to optimize power consumption and performance. It aims at striking a balance between power efficiency and computational performance by scaling voltage and frequency based on workload requirements and system conditions.

5.3.3 Scaling governor

The Scaling governor is a component within the Linux kernel responsible for determining the CPU frequency based on system load and power management policies. It works together with DVFS mechanisms to adjust the CPU frequency dynamically. It offers 4 modes each with different trade-offs between performance and power consumption, allowing the users to choose one based upon their specific workload and power consumption demands. The 4 modes are Performance, Powersave, Ondemand, and Schedutil mode. For the baseline of my experiments the Scaling governor was set to Performance mode, which keeps CPU frequency to the maximum allowed value regardless of any changes in load.

The above described DVFS and Scaling governor configuration decisions used in my baseline, ensured that during my experiments the frequency stayed consistent without turbo boost or the scaling governor intervening and increasing the frequency.

5.3.4 SMT

Furthermore, Simultaneous Multi-Threading (SMT), which is a performance-enhancing technique in modern processors that allows one physical core to function as multiple logical cores, was disabled in my machine configuration. SMT aims to improve the processor's performance, more specifically in tasks that include multitasking or parallel execution. However, SMT was disabled in my baseline configuration, to ensure that the two virtual instances of the 10-core Intel Xeon Silver 4114 processors do not interfere with each other by competing for the same physical core resources.

5.4 Metrics captured

All the metrics captured through my experiments in this thesis, are presented in relation to the value of Queries Per Second (QPS). The metrics are the following:

First are tail and average latency values. Tail latency is calculated for the 99^{th} percentile, in order to evaluate the Social Network under the strict requirements of a real-world latency-critical system.

Furthermore, the experiments capture the number of total requests executed and the total number of dropped requests. These metrics allows us to determine when the system becomes overwhelmed and reaches its threshold leading to requests being dropped.

Additionally, using a profiling tool called profiler, which can be found on GitHub [7] https://github.com/hvolos/profiler , I captured the C-state residency and C-state transitions. C-state residency measures the percentage of time a processor spends in each power-saving C-state, indicating how long it remains in various idle states versus being active. Whereas C-state transitions refer to the number of times the processor switches between different C-states, capturing how frequently it moves in and out of low-power modes. Metrics regarding C-states aid in the understanding of the machine's behaviour under specific workloads and how the efficiency is impacted when the processor enters a different C-state.

The last metric shown in my experiment results is machine power consumption measured in Watts for the two sockets of the c220g5 nodes and for the DRAM memory. Power consumption metrics were collected through RAPL (Running Average Power Limit) counters which are hardware registers in Intel processors that measure energy consumption for machine hardware components, like the CPU package, cores, and DRAM. They provide precise joule-based metrics, giving us a detailed analysis of system power usage. By examining the power consumption, we can understand and determine how the different methods of mapping the services to multiple machines impact the energy efficiency under the same workload.

Chapter 6

Mapping strategies and performance considerations

6.1 Deployment mappings analysis

The experiments in this thesis consist of 8 different deployments mappings. The first mapping deploys the Social Network on a single node using Docker Compose. For this experiment 2 nodes in total were used with 1 node acting as a client, and the other node hosting all the services and application components.

In order to determine the effects of cluster size and different mapping configurations for the deployment of the benchmark, I split the application's components into 4 teams based upon their purpose and functionality. The 4 teams are Frontend, Logic and Tracing, Caching, and Databases/Storage. Using these 4 teams, I calculated all the possible distinct mappings that could be created for a cluster of 2 nodes, which are the remaining 7 of the 8 total experiments I will be presenting. Figure 5 indicates the services each of the 4 teams I created consist of, while Figure 6 shows the single node deployment and the 7 different 2 node mappings created with their associated shorter aliases used to identify each mapping on the result plots in Chapter 7. For the experiments of these 7 2 node mappings Docker Swarm was used for deployment; 3 nodes were used in total, with one of the nodes acting as a client and the other 2 nodes being used to deploy and run the application's services and components.

Team	Services
Team 1: Frontend	media-frontend, nginx-web-server
Team 2: Logic and Tracing	compose-post-service, home-timeline-service, media-service, post-storage-service, social-graph-service, text-service, unique-id-service, url-shorten-service, user-mention-service, user-service, user-timeline-service, jaeger-agent, jaeger-query
Team 3: Caching	home-timeline-redis, media-memcached, post-storage-memcached, social-graph-redis, url-shorten-memcached, user-memcached
Team 4: Databases/Storage	media-mongodb, post-storage-mongodb, social-graph-mongodb, user-mongodb, user-timeline-mongodb, jaeger-collector, cassandra-schema, cassandra

Figure 5 Teams and services

Mapping Name	Alias
Single Node	FLCD
Node0: Frontend and Databases Node1: Logic and Caching	F,D – L,C
Node0: Frontend and Caching Node1: Logic and Databases	F,C – L,D
Node0: Frontend and Logic Node1: Databases and Caching	F,L – D,C
Node0: Frontend, Logic, and Databases Node1: Caching	F,L,D - C
Node0: Frontend, Logic, and Caching Node1: Databases	F,L,C - D
Node0: Frontend, Databases, and Caching, Node1: Logic	F,D,C - L
Node0: Logic, Databases, and Caching Node1: Frontend	L,D,C - F

Figure 6 Mappings and aliases

6.2 Mapping choice impact

The choice of deployment mappings can significantly influence the performance and efficiency of the benchmark application. This following subsection, examines the anticipated impact of

different mapping configurations on key performance metrics, considering factors such as resource allocation, workload distribution across nodes and communication overhead.

6.2.1 Mapping Strategies and their effect on key performance parameters

The specific deployment mappings used in this study may create diverse performance outcomes for the benchmark application. Configurations that distribute two teams per node aim to balance workload and service distribution, potentially optimizing resource utilization and minimizing communication overhead. Conversely, mappings with three teams on one node and one team on the other might introduce resource contention, potentially impacting overall performance metrics such as latency, but also can serve as indicators for identifying which teams of services should not be separated on different machines due to the needs of the other three teams of services. Additionally, observing similarities in improvement or degradation of performance across mappings with common patterns, such as isolating specific teams of services from others, can provide insights into the optimal deployment strategy for the benchmark application.

Chapter 7

Experiment results and evaluation

7.1 Experiment details

The metrics described in section 5.4 are presented in relation to a QPS range of 100-1000 QPS with an increase of 100 QPS each time to determine when the system becomes overwhelmed with handling too many requests. Additionally, average and tail latency are also shown in relation to a QPS range of 100-600 QPS, in order to examine them before dropped requests occur at higher QPS. The runtime of the workload for each value of 100, 200, 300 ... 1000 QPS is 30 seconds. For the single node deployment in section 7.2, to increase the accuracy and consistency of the results, each 30 second workload is repeated 5 times with each bar representing 1 repetition in the plots. For the comparison of different mappings in section 7.3, the average of 5 repetitions is shown in each bar in the result plots.

7.2 Single Node deployment results

Before splitting the application's components and services on 2 nodes using different mappings, I had to establish a solid way of deploying the application on 1 Node and get familiar with collecting and presenting the metrics presented in all the experiment plots. However, the 1 node

deployment results still provide some insightful results and allow us to examine latency and machine behaviour while ensuring that no additional latency is introduced due to the communication of components and services between multiple machines.

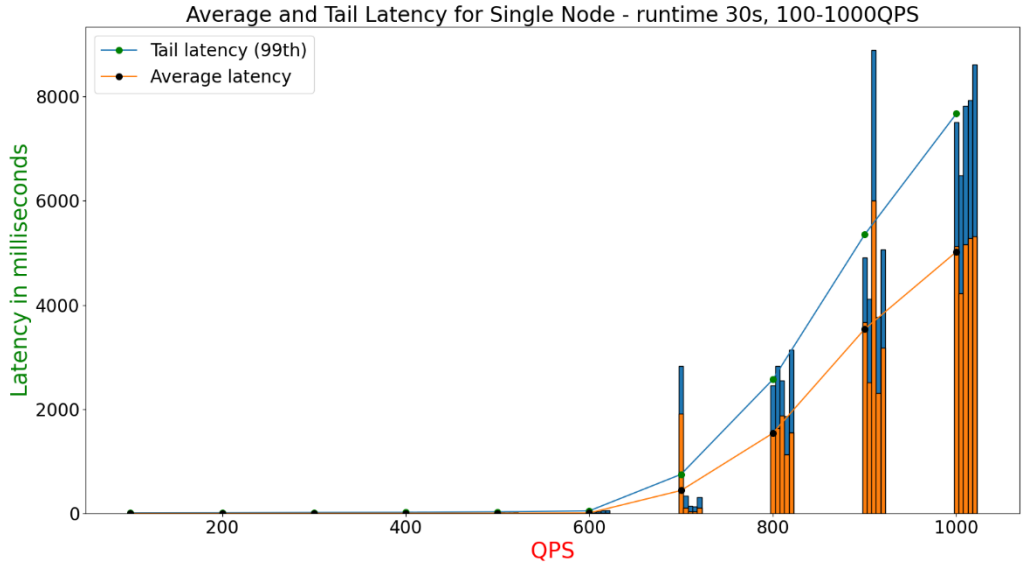

Figure 7 Average and tail latency relative to 100-1000QPS, Single Node deployment

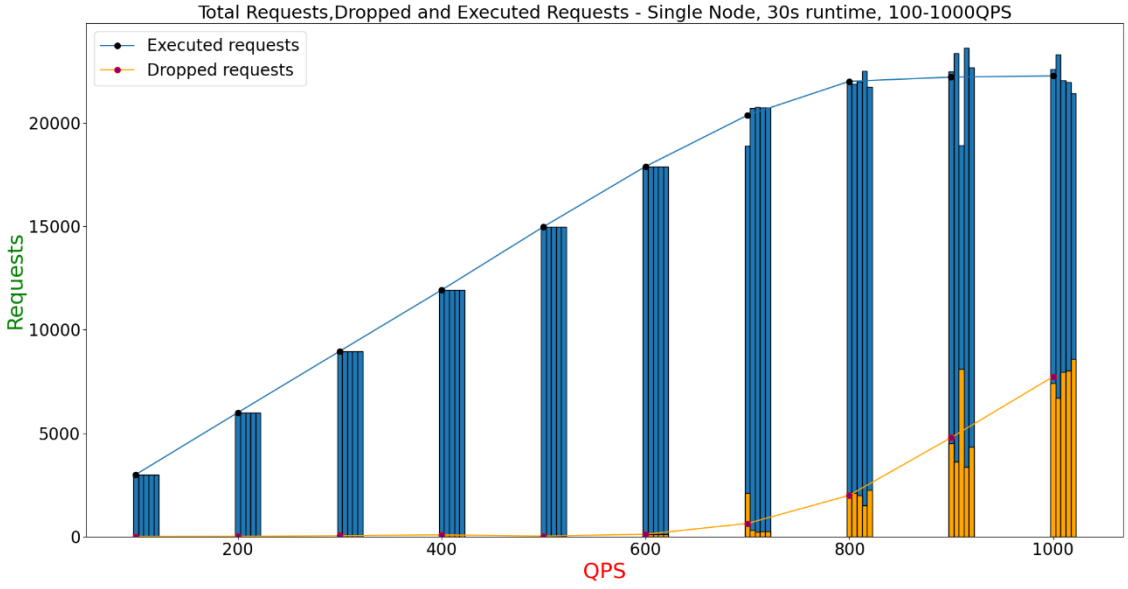

Figure 8 Total, dropped and executed requests relative to 100-1000QPS, Single Node deployment

As shown in Figure 7 and 8, the system handles all incoming requests efficiently up to 600 QPS. From 700 QPS and onwards, requests start being dropped, and the latency for the requests that are executed experiences a significantly larger increase which only becomes greater as the number of requests increases until it reaches 1000 QPS. This indicates that the QoS the system can offer is acceptable and stable within the QPS range of 100 – 600 QPS.

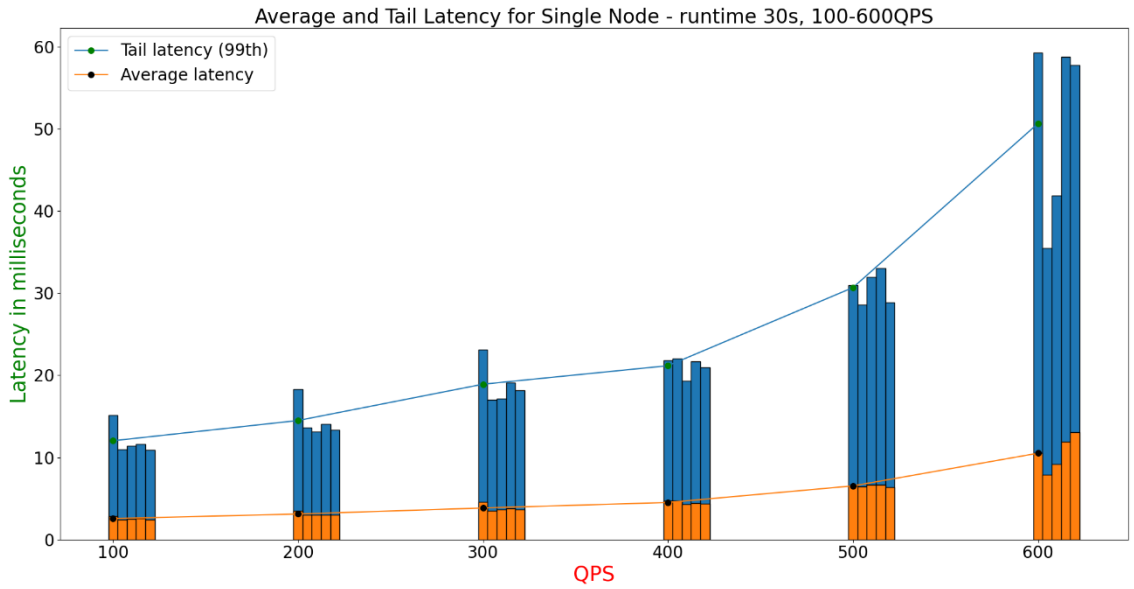

Figure 9 Average and tail latency relative to 100-600 QPS, Single Node deployment

Figure 9 illustrates the tail and average latency while the system is still stable and does not drop any requests. Average latency for the range of 100-600 QPS remains under 10 milliseconds, and tail latency calculated for the 99th percentile has a maximum value of 60 milliseconds at 600 QPS. It is noteworthy that tail latency is almost 6 times greater than the average latency for the same workload.

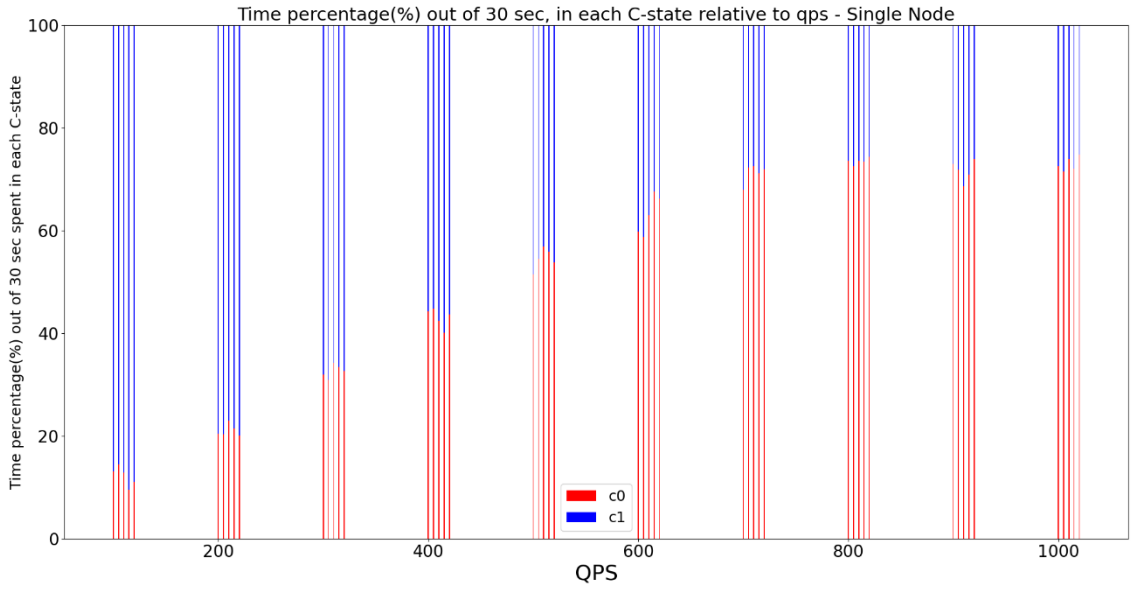

Figure 10 C-state residency relative to 100-1000 QPS, Single Node deployment

The plot in Figure 10, presents C-state residency for the QPS range of 100-1000QPS. The percentage value shown in this y-axis of this plot represents the percentage of time out of the total runtime of 30 seconds for each run. From 100 – 600 QPS we observe that the C0 residency increases while the number of incoming requests increases. This is expected since the processor must handle more requests and in turn stay in a more active C-state (C0) for larger periods of time. For the range of 700 – 1000 QPS, where the system is unstable and requests start being dropped, the time percentage spent in C0 stays the same at 75%. This is due to resource saturation caused by the demanding workload at higher QPS. Still, it is interesting to note that the system reaches 70% utilization before requests start being dropped. Most data center operators, when hosting latency-critical applications, aim at keeping the utilization between 5-25% to maintain tail latency under control.

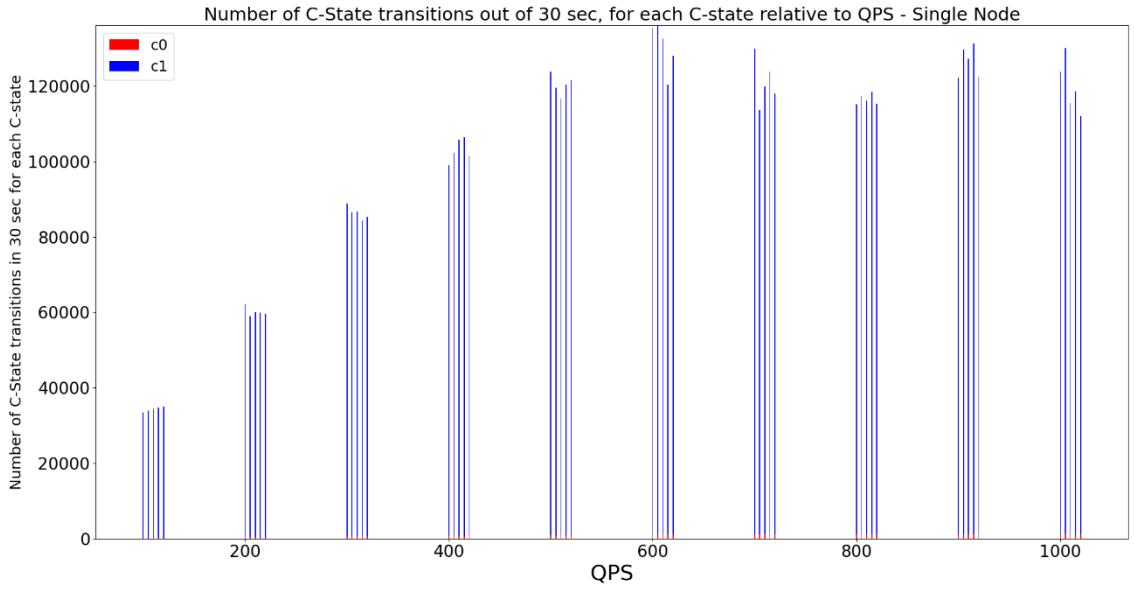

Figure 11 Number of C-state transitions relative to 100-1000 QPS Single Node deployment

Figure 11 shows the number of C-state transitions, which are the number of times the processor switches between the 2 enabled C-states (C0, C1). As noted in the previous paragraph, evaluating the C-state residency plot, as the QPS increase the time spent in C1 decreases and time spent in the more active C0 increases.

In the C-state transitions plot (Figure 11) we observe that the number of C1 transitions increases while the QPS increase. This suggests that even though the processors transition to C1 more frequently as QPS increases, they do not stay there for long time periods, which is evident from the increasing C0 residency observed in Figure 10. More specifically, when the system experiences higher workloads the processors try to enter a more idle state but are interrupted by the high workload demand, forcing them to spend more time in a more active state (C0), which leads to greater C0 residency but also a larger number of C1 transitions. Additionally, the frequent transitions to C1 might be influenced by power management policies that aim to save energy by quickly moving the processors to an idle state whenever there is a brief pause in activity. However, due to the high demand, these idle periods are very brief, resulting in numerous transitions.

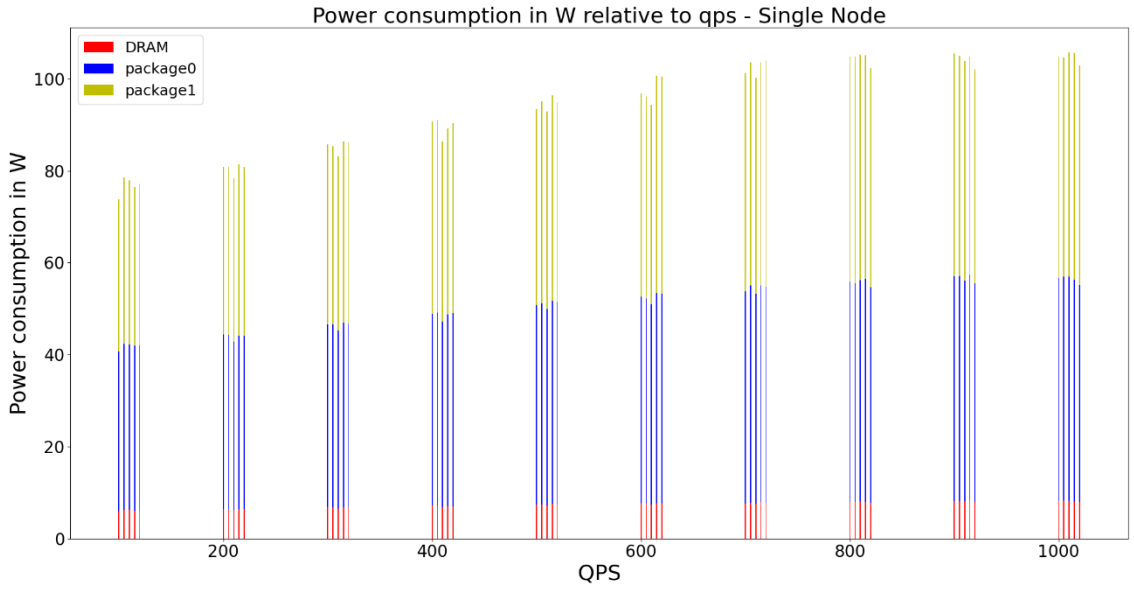

Figure 13 Power consumption in Watts relative to 100-1000 QPS Single Node deployment

In the power consumption plot for the Single Node deployment, we can see that the main contributors to the overall power consumption are the two processors, with both packages contributing almost the same number of Watts to the overall power consumption. As the QPS increase, power consumption for DRAM does not showcase any significant increase, whereas the power consumed by the 2 packages increases linearly in relation with the QPS. The increase of power consumption in relation to the increase of QPS aligns with the observation from Figure 10, which is that at higher QPS the two sockets could not enter a more power saving C-state (C1) for long periods of time, which in turn lead to higher power consumption.

7.3 Two Nodes results and comparison of all mappings

In this section the plots compare the single node deployment with the 7 different 2 node mappings. Each of the 8 bars appearing in the plots represent the average of the metric for each mapping, except for power consumption where for the 7 2-node mappings the sum of the power consumption of the 2 machines used is displayed in each bar.

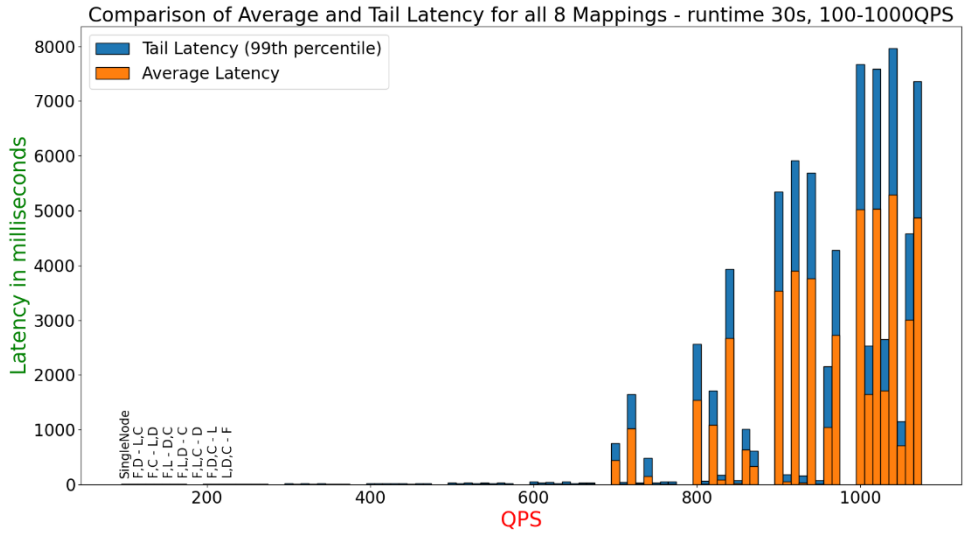

Figure 14 Average and tail latency relative to 100-1000 QPS, Comparison for all 8 mappings

Figure 15 Dropped and executed requests relative to 100-1000 QPS, Comparison of all 8 mappings

As observed in the single node deployment the system becomes overwhelmed and starts dropping requests after 600 QPS. In figures 14 and 15 we can see that this is also the case for all the 7 2 node mappings, because even though each machine has less load to handle, the communication overhead between the 2 machines plays a major role in the increase in latency. This means that the addition of a second machine for deployment has not extended the range

of QPS in which the system does not drop any requests so, to determine which 2 node mappings offer the best and worst QoS and performance we will evaluate the results presented in the following figures for up to 600 QPS, while the system is still stable and handles the requests efficiently.

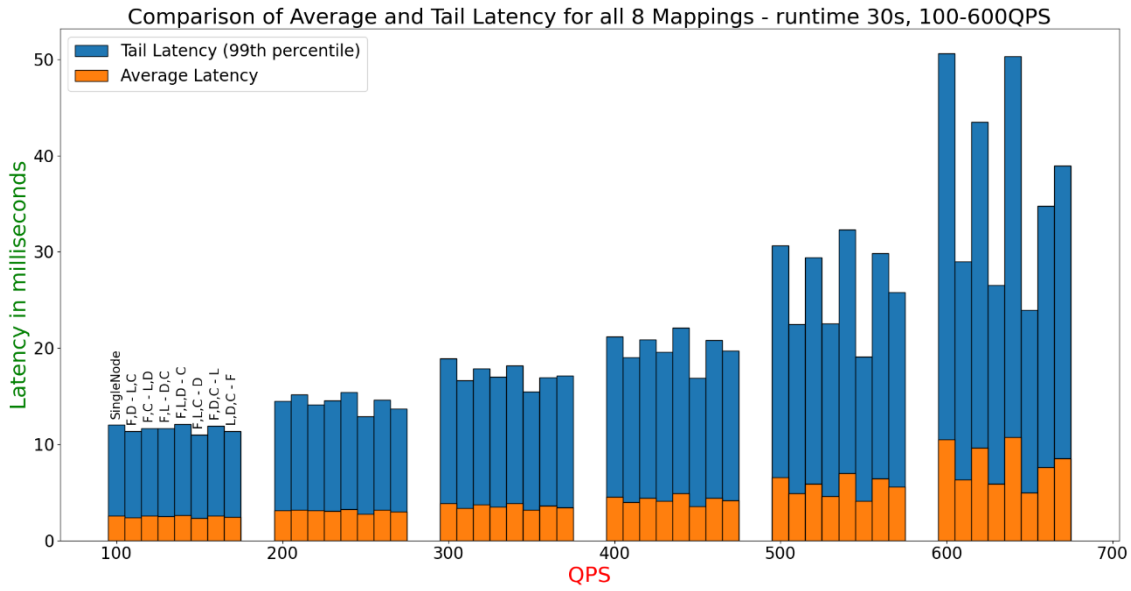

Figure 16 Average and tail latency relative to 100-600 QPS, Comparison of all 8 mappings

For the range of 100 – 300 QPS there isn't a significant difference in average and tail latency between the 8 mappings. In the range of 300 – 600 QPS it becomes evident that the best performing mapping in terms of average and tail latency is the mapping called F,L,C – D, which is the mapping that isolates databases on 1 of the nodes and deploys the other 3 teams of services (Frontend, Logic and Caching) on the other node. Moreover, at 600 QPS we can also observe that the 2-node mapping that displays the highest average and tail latency is F,L,D – C which isolates Caching on a single node and deploys the other 3 teams on the other node.

Even though Caching does not store as much data as the Databases, the data transfer from databases to the second machine that hosts the other 3 teams of services, is not as costly to latency because the mapping that isolates databases on 1 machine has proven to be the best performing in terms of latency. This is due to the other services needing to access cached data very frequently. This frequent need of cached data being transferred across machines due to the benchmark exhibiting good data locality, trumps the latency introduced from the transfer of larger data from the databases between the 2 machines and their services (best performing

mapping – Databases isolation), making the mapping that isolates Caching on one of the nodes the worst performing amongst all the mappings.

Furthermore, analysing the latency values of the mid-range deployment mappings, meaning the ones that don't display the lowest or the largest latency values and comparing them with the best and worst performing ones reveals distinct patterns and similarities. More specifically, mappings that distribute workload evenly across nodes, such as F,D - L,C and F,C - L,D, demonstrate relatively balanced performance, with tail latency values ranging from 30-35 milliseconds at 600 QPS. Whereas configurations deploying three teams on one node and one on the other, like F,D,C - L and L,D,C - F, tend to exhibit higher latency, with tail latency values reaching 35-50 milliseconds at 600 QPS due to increased resource contention. An exception is observed with mappings isolating the Databases team on one node (F,L,C – D), which as noted in the two previous paragraphs achieves the lowest latency values of 25 milliseconds at 600 QPS by optimizing resource utilization, minimizing communication overhead, and leveraging data locality. Notably, mappings co-locating teams with frequent interactions, like F,L - D,C, show improved latency outcomes, with tail latency values around 28-30 milliseconds at 600 QPS, highlighting the benefits of minimizing inter-node communication.

Regarding the overall comparison between Single Node deployment and 2-node deployment in terms of latency we can observe that none of the 7 2-node mappings offer worse QoS compared to the single node deployment. Still though, the worst performing 2 node mapping offers almost as high latency as the single node deployment and the best performing 2-node mapping offers significant QoS improvements, displaying lower average and tail latency.

At 600 QPS it is interesting to note that the single node deployment and the worst performing 2-node mapping have a tail latency of 50 milliseconds, whereas the best performing 2-node mapping offers a 50% improvement, with the tail latency at 600 QPS being 25 milliseconds. This indicates that the careful choice of mapping when using two machines plays a more crucial role in latency improvements, compared to opting for the use of two machines instead of one, in the hopes that the added machine will improve QoS. This is evident from several 2-node mappings having the same or not significantly better latency values, when compared to the 1-node deployment.

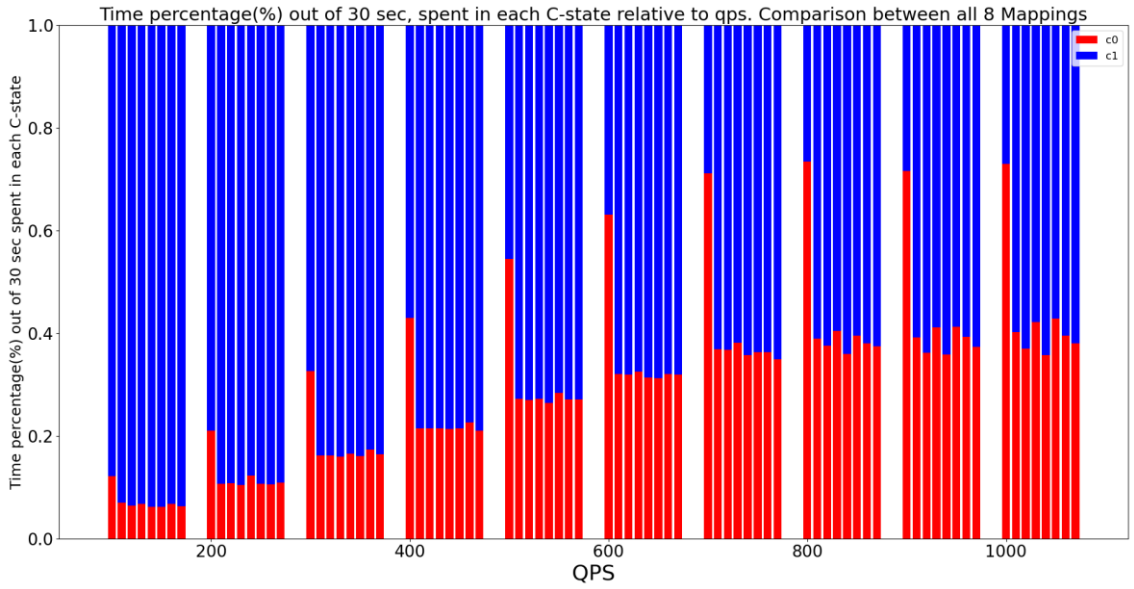

Figure 17 C-state residency relative to 100-1000 QPS, Comparison of all 8 mappings

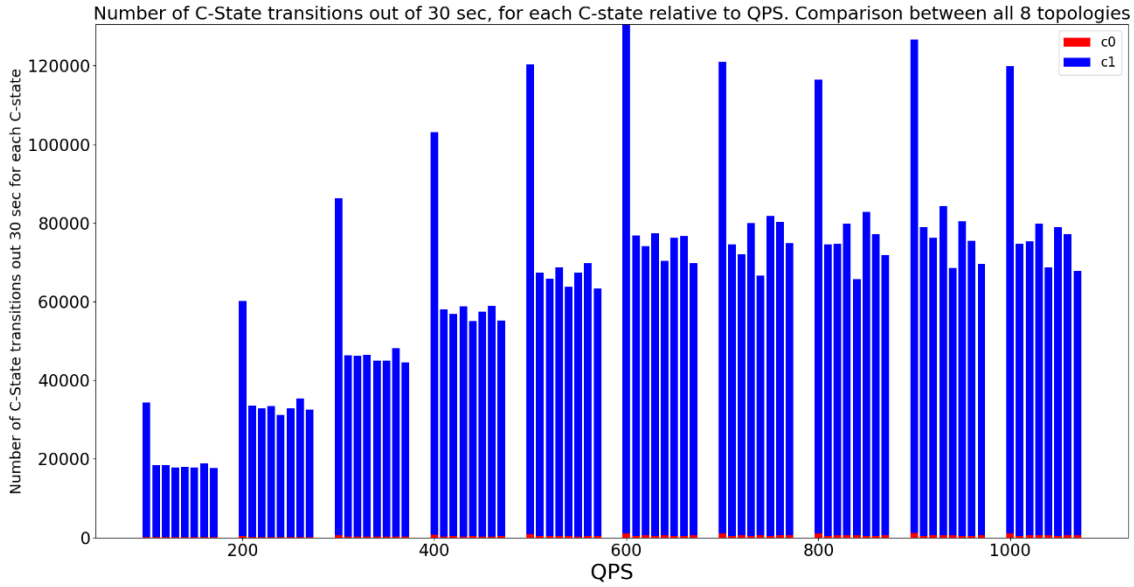

Figure 18 Number of C-state transitions relative to 100-1000 QPS, Comparison of all 8 mappings

Figures 17,18 display the average of C-state residency and C-state transitions between the 2 machines used for the 7 2-node mappings. Each of the 8 bars seen in Figures 17 and 18 represent the values for each mapping in the order they appear in the text labels of Figures 14,15,16.

Due to some of the 2-node mappings deploying 2 teams on both nodes, and some mappings deploying 3 teams on 1 node and isolating 1 team on the other node, calculating and presenting the average C-state residency between the 2 nodes used, is better for a comparison between the 2-node mappings, as well as for a comparison to the single node deployment. More specifically, this allows us to evaluate machine behaviour considering both the machines used and not only the node dealing with more demanding tasks.

The trend of C0 residency increasing linearly in corelation with the QPS within the range of 100-700 QPS, observed and discussed in section 6.2 remains true for all the 7 2-node mapping experiments, including the fact that after 700 QPS the C0 residency remains relatively the same until 1000 QPS (due to dropped queries).

Having to handle the same workload using twice the number of machines allows the nodes used in the 2-node mappings to enter a more idle state (C1) for longer time periods compared to the single node deployment. This is evident from the plot seen in Figure 17, where none of the 7 2-node mappings exceed 40% of the runtime spent in C0, whereas as seen in section 6.2 and here in Figure 17 the single node deployment shows a maximum percentage of time spent in C0 of 75%.

When comparing the 7 2-node mappings with each other in terms of C-state residency, we can see that the average residency between the 2 machines used is almost identical for all the mappings.

Regarding the number of C-state transitions, shown in the plot of Figure 18, all 7 2-node mappings display significantly smaller number of C1 transitions throughout all the QPS values, when compared to the first bar which represents the single node deployment. This is because the 2 nodes do not handle the entirety of the services and components of the application and they are not forced to exit the more idle state (C1) and transition to a more active C-state (C0) as frequently as the single node deployment which experiences higher utilization under the same workload. This aligns with the observed lower C0 residency for all QPS values that the 2-node mappings show compared to the higher C0 residency of the single node deployment.

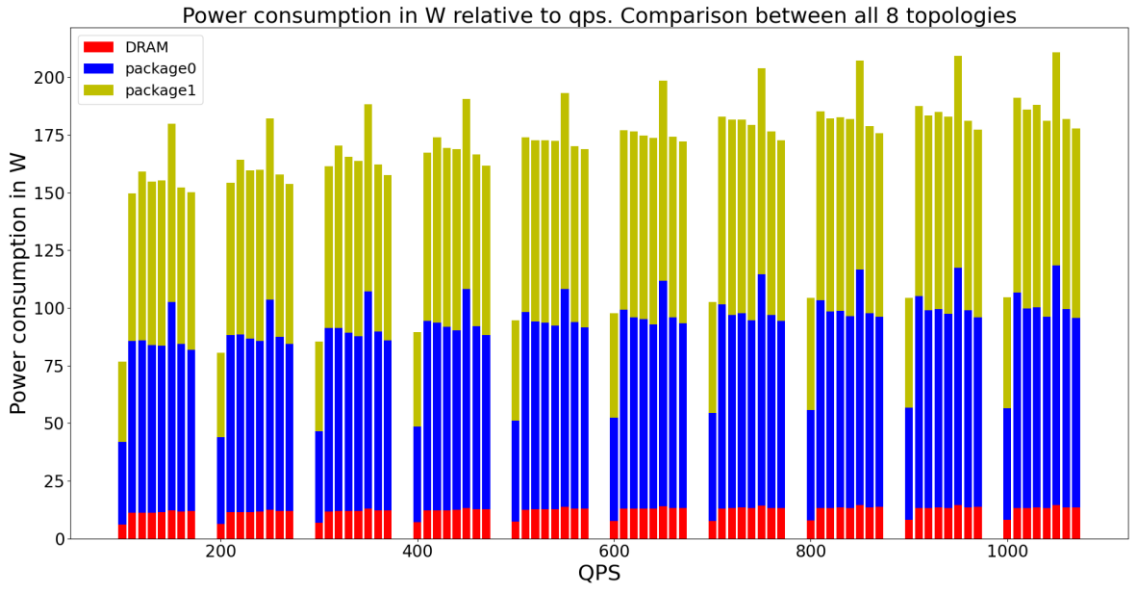

Figure 19 Power consumption in Watts relative to 100-1000 QPS, Comparison of all 8 mappings

In Figure 19 the power consumption in Watts is presented for all 8 mappings. For the 7 2-node mappings the sum of the power consumption from the 2 machines used is presented. Each of the 8 bars in figure 19 represent the values for each mapping in the order in which they appear in the text labels of Figures 14,15,16.

Using two machines means that the idle power consumption which is the power consumption when the machines have no workload to handle (before the workload of the experiment starts), will be twice as large as the idle power consumption of 1 machine. In Figure 19, we observe that this is also the case for active power consumption because power consumption for all 2 node mappings is twice as large as the single node deployment across the 100 – 1000 QPS range, with the exception of mapping F,L,C –D, which displays the largest power consumption amongst all 2-node mappings. It is interesting to note that this mapping which isolates Databases on one machine and the other 3 teams of services on the second machine, has the highest power consumption, as well as the best QoS with the lowest average and tail latency as seen previously in Figure 16. A possible reason for the higher power consumption of this mapping is the added overhead of transferring the large amounts of data from the databases between the 2 machines, as well as one of the two machines having to execute the more power intensive tasks that Frontend, Logic, and Caching require.

Chapter 8

Related work

My work shares many similarities with the undergraduate thesis ‘Cloud-Based Microservices’ by Stylianos Vassiliou. Both papers use the same benchmark application, delve into the architecture of microservices, the technologies used, the advantages and implications of microservices, as well as the impact of cluster size and different workloads on various latency critical systems’ performance evaluation metrics as well as C-states and power consumption. The focus of Stylianos Vassiliou’s experiments is the examination of the effect of different cluster sizes, different C-states enabled or disabled and isolating a specific service or services on a single machine.

The key differences in my work and experiments are that the machine characteristics remain the same throughout all experiments with no other C-states being enabled apart from C0 and C1. Additionally, my work expands on the isolation of specific services on individual machines by dividing all components and services of the application into teams based upon their functionality and purpose and running experiments using all the possible ways to map the teams on 2 nodes so as to determine the best mapping for each metric and identify the common mapping patterns that lead to worse or better application performance.

Furthermore, my work utilizes the Social Network benchmark application created by the SAIL team at Cornell University and expands on the experiments presented in the associated paper called DeathStarBench, by examining performance evaluation metrics for a greater range of QPS as well different ways of mapping the different components and services of the application.

Chapter 9

Conclusion

9.1 Conclusion

The microservice architecture is quite complex, and there are a lot of aspects that impact the performance of a microservices application to consider and study. However, if the right deployment strategies are implemented and used, efficiency can be greatly improved. Through my research and experiments in this thesis, I have gained some important insights into how microservices perform.

I have found that the use of a 2-node cluster does not ensure that the QoS will be necessarily better than deploying the services on a single node. The improvement of latency values depends heavily on the choice of separating different services and components of the application, while balancing the trade-offs between overwhelming certain nodes with more services and simultaneously considering the latency introduced by the communication overhead and data transfer between the nodes. The results from my experiments indicate that separating critical path services on different nodes introduces additional latency with the worst case being isolating Caching services.

Furthermore, I have observed that examining the behaviour of the processors regarding C-state residency, C-state transitions and power consumption provides a better insight when designing a system that has good power efficiency and QoS in mind. The results from my power consumption experiment indicate that the mapping that provided the best QoS also had the highest power consumption amongst all the different mappings. This means that when choosing

a mapping for the deployment of a microservices application we must consider the needs and priorities of the system and balance the trade-offs in order to achieve the desired power efficiency, while also providing the users of the application with the satisfactory QoS that falls within the set restrictions.

9.2 Further work

While my work focused on examining the effects of deploying the services of a microservices application on a single node and a 2-node cluster, using various mappings and the same machine configuration, many other areas can be explored to expand upon my work that can provide better insights on the performance of a microservices application.

Firstly, enabling turbo frequency technologies and increasing un-core frequency for the baseline machine configuration, can help determine if enhancing the machine performance can drastically improve the QoS of the application. Furthermore, enabling SMT can help us determine if the microservices leverage the parallelism capabilities of a machine that offers SMT to their advantage. Additionally, examining the effects of cluster sizes larger than the ones presented in this thesis, can provide better insights regarding the improvements in application and machine performance. Moreover, an additional helpful metric that could be captured during experiments is measuring and presenting the amounts of data transferred through the network between the 2 nodes used for the deployment of the application, in order to evaluate which types of services communicate and exchange the most amount of data. Lastly, deploying the benchmark application using a larger dataset of users than the one used in this thesis can help in the understanding of the limitations of the application and the nodes that host its services.

References

[1] S. team, “An Open-Source Benchmark Suite for Microservices and Their Hardware Software Implications for Cloud & Edge Systems,” in In Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), New York, 2019.

[2] Islam, Muhammed Tawfiqul. Cost-efficient Management of Cloud Resources for Big Data Applications. Diss. PhD thesis, The University of Melbourne, Australia, 2021.

[3] Figure from: https://www.nginx.com/page/23/?m=0

[4] Figure from: https://alokai.com/blog/microservices

[5] Cloudlab site: https://www.cloudlab.us/

[6] J. H. Yahya et al., "AgileWatts: An Energy-Efficient CPU Core Idle-State Architecture for Latency-Sensitive Server Applications," 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), Chicago, IL, USA, 2022, pp. 835-850, doi: 10.1109/MICRO56248.2022.00063.

[7] H. Volos, “Profiler,” [Online]. Available: https://github.com/hvolos/profiler.