Real-time Scheduling in Datacentre Clusters

FABIAN FRANKEL & SEPEHR TAYARI
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Real-time Scheduling in Datacentre Clusters

Fabian Frankel
fa7320fr-s@student.lu.se
fa.frankel@gmail.com
Sepehr Tayari
elt15sta@student.lu.se
sepehrtayari@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor: Per Skarin, William Tärneberg

Examiner: Maria Kihl

June 23, 2021

© 2021
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Industry 4.0 can be described as the next generation-factories that is characterised by putting a high demand for automation and flexible production lines. The proposed way to achieve this goal is through a large number of Industrial IoT devices (IIoT) in the factory, some having high availability- and low-latency requirements. This demands for a software that can monitor and manage the applications used by these devices. The current industry standard for managing clusters of applications is called Kubernetes. However, using Kubernetes in this environment means that it needs to meet the low latency demands of Industry 4.0. The purpose of this thesis was to investigate the possibility of achieving this support in Kubernetes. To investigate if this was possible, a Kubernetes cluster was deployed to Ericssons private cloud Xerces. On this cluster services was deployed that executed arbitrary tasks with the invocation of an HTTP-request. The difference in how fast these tasks executed was used as a metric to see the delay some request were experiencing due to non optimal scheduling on Kubernetes. Through investigations of how the underlying kernels schedules jobs on it's CPU proposed solutions to reduce the execution time was made. These solutions were then tested on the cluster and compared with the first measurements. The results of these measurements showed a reduction in execution time for the deployed services with the introduction of the preemptive scheduling policy `SCHED_FIFO` in Linux, which uses the first in first out algorithm. This improvement in execution time was at the cost of a degradation of the response time of the cluster. These findings points towards the conclusion that applying a real time scheduling policy to processes on a Kubernetes cluster is not cost free. However, a suggested path for how to further optimise a Kubernetes cluster in regards to response time has been proposed.

Popular Science Summary

In the near future, robots and humans might work hand in hand. At least, that is the intention of the future Industry 4.0 deployments where the industries will make use of real-time data generated by the great number of integrated sensors in the production lines. But in order for this sci-fi scenario to become a reality, all processes within the industry must work seamlessly and with high determinism.

For a seamless deployment of Industry 4.0, a software that monitors and controls the deployments running in the cloud must exist. Today, the industry standard for monitoring cloud deployments is Kubernetes. To determine its deterministic behaviour and suitability in a Industry 4.0 environment, this paper has studied the reliability of a cluster controlled by Kubernetes running in a virtualised environment. Using the Linux scheduling policy `SCHED_FIFO` and adding this priority using the deployment in Kubernetes resulted in a more deterministic cloud system where each executed task executed faster and with higher determinism when prioritised. Although, the prioritisation affected the response time of clusters negatively when the load was increased. This has been addressed, and a possible solution to this drawback has been presented in the final work.

The possibilities in Kubernetes are endless and thanks to its easy accessibility, users can simply make prioritisation to the processes which demands reliable execution by simply entering the relevant node and making few modifications. By prioritising crucial processes such as networking- and real-time sensitive processes in a Industry 4.0 system, the deterministic behaviour of the cloud could be increased quite easily even in a virtualised environment.

As Kubernetes grows and becomes a prevalent part of the cloud architecture, it is crucial to prove that Kubernetes also suits Industry 4.0 deployment to push the research area in the correct direction. Due to Kubernetes being a growing open-source project, missing out on Kubernetes mean that Industry 4.0 cannot make use of the latest technologies and available knowledge relating to cloud deployments.

Acknowledgements

We would like to express our gratitude to everyone who helped us throughout our thesis work. First off we would like to thank everyone at Ericsson for providing us with knowledge and the necessary resources to conduct this thesis. We would also like to thank our two supervisors, Per Skarin and William Tärneberg, who has been a big help with discussing ideas and solutions to the problems that came about. Lastly we would like to thank our friends and family for being a great mental support throughout this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has become the main source of chosen infrastructure for delivering IT-resources due to its ability to meet on demand scalability while simultaneously offering a high quality of service. This trend shows no signs of stopping and today most people use the cloud on a daily basis through storage, sharing documents, or playing online games to name a few. However, in order to meet the demands of increasingly more latency sensitive applications the cloud infrastructure needs to adapt.

In the next generation of industrial factories, the smart factories of Industry 4.0, there is a big push for automation and flexibility in order to stay competitive in the market. The solution to achieve this is through industrial internet of things IIoT. This demands for a software to manage the applications running on the IIoT devices. However, managing software clusters in an Industry 4.0 context poses significant demand on runtime automation, timeliness, low latency, and high reliability due to the real-time systems present. These properties need to be addressed in a cost-efficient manner for the development of Industry 4.0.

Software clusters are today implemented by deploying processes on virtual machines (VMs), and by using containerised operational system (OS) level virtualisation, where the latter solution provides better results in term of isolation and interference for real-time tasks [3, 4]. When using container-based virtualisation, the Linux kernel has various tools, such as real-time schedulers, to handle mentioned tasks which gives containerised virtualisation an edge compared to full virtualisation. [3]

Today the cloud native services provided by current cloud providers are not always suitable for low-latency real-time applications [5]. However, it is proposed that appropriate changes can be made to the system to make it viable for real-time applications. To orchestrate real-time scheduling for containerised virtualisation, a modified cloud native approach could be used which takes use of existing real-time schedulers. Currently, Kubernetes, which is a widely used open source tool for cloud orchestration has become the industry standard for cloud native deployments. Solving the real-time support for Kubernetes would imply a large benefit for future deployments.

Achieving a more deterministic system with lower variation time for incoming tasks to a cloud native cluster could point towards the conclusion that Kubernetes could be a viable contrainer software orchestration tool for an Industry 4.0 context. This would also mean that the constantly expanding array of tools to support Kubernetes could be available in this context as well [2].

1.1 Problem definition

Kubernetes is the current standard for cloud native containerised orchestration and provides benefits such as load balancing, automated roll-outs and rollbacks, and security benefits [2] which will be of high importance in an Industry 4.0 context.

The goal of the research is to through our findings from investigating the Linux scheduling policies and cloud native deployments, investigate Kubernetes service's viability for the deployment of real-time sensitive applications. With the help of the Linux kernel's prioritisation of tasks, we hope to reduce the variation of execution time for jobs deployed to the cluster while analyzing how the response time is affected.

The above mentioned goal can be summarised into three points:

- deduct the possibility of achieving real-time support for specified jobs in a Kubernetes cluster,
- reduce both execution time and variance in execution time for the specified prioritised jobs,
- evaluate the optimisation methods effect on the response time for prioritised jobs
- evaluate the possibility and present a method for deploying a Kubernetes cluster with support for real-time prioritisation.

1.2 Previous work

[6] discusses two main scheduling approaches. The first is multi-step scheduling which is the approach of scheduling each pod (a container with an abstraction layer on top) independently from one another. The second is Single-step scheduling which considers a set of pods to schedule. The second technique increases the

scheduling complexity while also resulting in a more optimised schedule. In a recent study [7] a custom scheduler as a hybrid of the aforementioned approaches is presented as a scheduler for real-time applications. This research presented a result that significantly decreased the execution time for tasks, as the number of tasks increased. This research has led to a significant increase in efficiency in a Kubernetes cluster, and we hope that our own research, introducing a well-established and proven scheduler, like the scheduler in the Linux kernel, will amplify this result.

A study [3] from 2019 proposes a way to guarantee temporal scheduling on co located containers. This is done by an extension of the SCHED_DEADLINE scheduling policy. This resulted in a scheduling policy that can be used on a LXC with multiple virtual CPUs. In addition to this the study, [4] presents how to successfully deploy virtual network functions as containers on a private cloud. To achieve this, the Linux kernel v4.16.0-rc1 with a patch extending SCHED_DEADLINE is used. Similarly, this research intends to use the Linux kernel's scheduler to improve the execution time for containerised applications.

In [8], progress-based container scheduling in a Kubernetes cluster is investigated. This aided in implementing the ProCon scheme. The Pro Con scheme schedules the incoming tasks based on, not only the current resources available, but also on the predicted future resources of a node. These findings resulted in a Kubernetes cluster with 23.0% better performance compared to a cluster with the default scheduler. This research can be useful in understanding how we can modify the Kubernetes scheduler for the integration with the Linux scheduler's properties.

1.3 Limitations

This thesis is limited to the scope of investigating the execution time of processes on a Kubernetes cluster.Although the execution time for processes on a cluster is an important aspect for a platform to be considered viable for the deployment of real-time applications. It is not all that is required.

What the end consumer is experiencing is in fact the response time. There has been researched done in this field, improving the latency of networking which will ultimately result in a reduced response time. In [9] network function virtualisation gives promising results in reducing response time latency in a cloud datacenter context. Another topic that this thesis does not address is the live migration of applications in a kubernetes cluster. The act of live migrating application refers to the relocation of applications while not disconnecting the client. The reason one would do this is in a loadbalancing context were one would want to offload a server. Another reason is when the hosts needs maintenance. The advancements in this field will therefore mean a lower downtime for applications in the mentioned scenarios, a very important topic in the real-time computing on a cloud infrastructure.

Chapter 2

Background

To understand what cloud currently means, we will be giving a brief history of how cloud services have evolved into what it is known as today. We will only then introduce further details around specific available cloud technologies.

Today, the cloud has somewhat turned into a buzzword as if it were something completely new. Some areas, such as cloud native, are relatively new. But the concept of cloud has been around for longer than most people think. In the 1970s, IBM released an OS named VM that would revolutionise the cloud by introducing virtual machines and the hypervisor. These technologies allowed users to work flexibly and made it possible for them to run their code on different OS'. Virtualisation lays the foundation for what the cloud looks like today, and now 50 years later, machine-level virtualisation is still heavily used in the cloud. [10]

2.1 Traditional Deployments

At the dawn of internet applications and websites, an organisation or person who wanted to deploy an application or web service had to deploy it on a physical server. This deployment method (often referred to as bare metal deployment or hardware deployment) required the developer to be very aware of the application's traffic, since the only option for alleviating load from one server was to start a new server that could handle some of the traffic. Today this process can be done dynamically, but there are still limits on how fast it can be done. [11] Another issue with bare metal deployments was that there were no way of constraining the resources used by one application. This issue led to the developers often underutilising the resources available by only designating servers for one application.

2.2 Machine Virtualised Deployments

A VM is a virtual environment running on top of existing hardware that emulates a computer's functionality with a specific set of hardware and operation system configurations. VMs are frequently used in order to allow a system to run multiple operating systems by utilising virtualisation. Without virtualisation, one would require to run multiple physical units to run different configurations, such as different OS'. VMs fulfill the need for different users to run multiple isolated virtual computers with unique OS and hardware configurations for specific applications in a cloud environment. [12]

In order to deploy cloud infrastructure using machine level virtualisation, a hypervisor is used. The hypervisors job is to allocate the physical computer's (commonly referred to as the host) resources to each of the running VMs (commonly regarded as the guests). Each guest is an isolated instance with a guest OS and resources such as CPU, memory, and storage allocated by the host from a shared pool that can be reallocated between existing and new VMs. [13]

There are two types of hypervisors, type-I and type-II. Type-I is running directly on the hardware, and type-II runs on top of the operating system on the host. Type-I offers better performance since it communicates directly with the hardware. Type-I hypervisors needs hardware-specific configurations, making it less flexible, were in type-II, the OS covers this [14]. A figure representing an app deployment using hardware virtualisation with hypervisor of type-II can be found in figure 2.1. A usable metric when operating in a virtual environment is the steal metric. Steal metric is defined by IBM as the percentage of time that a virtual CPU is ready to run and waits for a real CPU.

The machine virtualised deployment method allows the user to isolate applications by running the applications on separate VMs. Compared to the traditional deployments where one could only achieve isolation by starting a new server instance, this method allows for higher hardware utilisation and increased scalability. [12]

2.3 Containerised Deployments

Containers is a standard unit of software that solves the problem of reliably running an application with the same dependencies and configurations when moved between different computing instances. Containers solves this problem by bundling up the runtime environment, the application and its dependencies along with other binaries and all of the needed configurations to run the application. This allows the developer to not worry about the underlying environment when moving an application from one computer to another. In other words, containers ensure that applications tested and verified on one computer can be deployed on another by utilising container images. In recent years, container orchestration has been widely

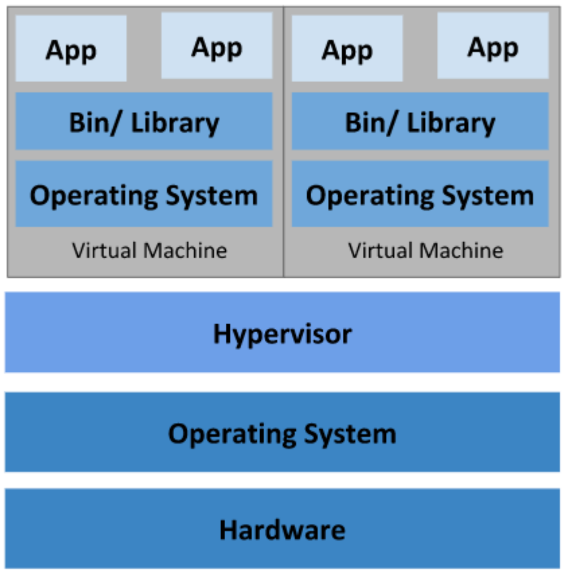


Figure 2.1: A representation of machine virtualisation with a hypervisor of type-II [2]

adopted as the industry standard in cloud native deployments. Rather than using machine level virtualisation as with VMs, container-based virtualisation is done on the operation system level. With this approach, the containers do not need a OS for every application, instead the container runtime is installed on the host and it is through that all containers on a computing instance share the host OS. As a result, the guests hardware architecture and kernel is the same as the hosts. This makes containers easier to deploy compared to machine level virtualisation due to the absence of the hypervisor and guest OS.[15] Due to their lightweight nature, containers are very suitable for fast and reliable scaling in a cloud native environment [16].

In order to run containers on the host, a container runtime is needed which is a software responsible for managing when and how the containers are allowed to use the host resources. A few popular container runtimes include docker, containerd, and lxd. A representation of the architecture for a containerised app deployment is presented in figure 2.2. [2]

To ensure fast scaling and high availability for deployed containers, a system for managing them is needed. If a container crashes it needs to be redeployed, and if an application is experiencing high traffic a copy of the container should be deployed to ease the load. The software to offer this service is called a container orchestration software and Kubernetes is the current industry standard. [2]

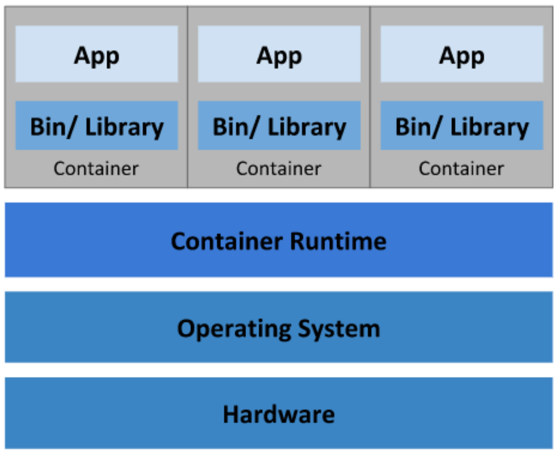


Figure 2.2: A representation of a containerised app deployment [2]

2.4 Cloud Computing Today

Cloud was in september 2011 defined by the National Institute of Standards and Technology as:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [17]

Cloud computing is today the most flexible option for delivering IT solutions. It has the architecture for ensuring high security, high flexibility, and its dynamic nature allows for greater resource allocation when compared to designated servers. [18]

The difference of the traditional deployments and cloud architectures used today are commonly explained with the allegory of *Pets vs. Cattle* [19]. With traditional cloud architecture, servers were treated as pets. If the server malfunctions or failed for some reason, the developers urgently needed to fix the problem. Usually, the servers were designed to not be down, and any downtime would affect the running system heavily. Cattle on the other hand are disposable. These servers are designed to tolerate failures using automated tools such as Kubernetes and consists of many computers in a cluster, or using the allegory, a *herd.* During failure, the cluster should have automated rollbacks and not require human support, by utilising full automation. Therefore, developers do not need to be concerned with if a server malfunctions or for other reasons becomes unresponsive. Today, this is what the cloud infrastructure looks like using a cloud-native approach. [20]

In 2007 Amazon was the first to offer a cloud computing service to the public. [21] Since then, many have followed to offer similar cloud computing services. However how a company can offer these services differ immensely and has therefore been grouped into different service models. A few of these service models will be given a brief introduction.

2.4.1 Software as a Service

Software as a Service (SaaS) is the service that offers the end-user an opportunity to use application or software that operates on the cloud. The consumer has no control nor perception of either the running application or the cloud infrastructure. SaaS is solely providing a user with an interface to the application running on the cloud. Since where and how the application runs are arbitrary to the end-user, this benefits the provider since they can share the running application between clients and increase resource utilisation. Providers can also direct the user to a new instance of the running application, meaning they can repair and upgrade applications with zero downtime for the clients. [22] [23]

2.4.2 Platform as a Service

Platform as a Service (PaaS) provides a platform for developers to run, test and deploy their applications. In contrast to SaaS which is directed more towards end users, PaaS is rather directed towards the developers. The providers offer the operating system, servers, systems for managing databases, software development tools, and analytic tools. This model is beneficial for small to medium-sized organisations to ensure they can speed up time to market without spending resources building their own infrastructure for the deployment process. [23]

2.4.3 Infrastructure as a Service

In [17] Infrastructure as a Service (IaaS) is described as the service where *"The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer can deploy and run arbitrary software, which can include operating systems and applications"*. This type of service gives the consumer freedom in implementing their software system. For example, a consumer can deploy their own choice of the platform providing service software to their infrastructure. Another benefit of this service model is it often becomes the cheaper alternative in the long run compared to PaaS. Although, the increased freedom of using IaaS also increases the complexity and could result in a higher barrier of entry for the consumer. [23]

2.4.4 Function as a Service

Functions as a Service (FaaS) or serverless computing offers a small container runtime where the consumer can execute code. In 2014 Amazon offered this type of service as AWS Lambda; since then, many providers, Google, Microsoft and IBM, to name a few, has offered solutions following the same service model. [24] This service model puts an abstraction on most of the infrastructure configuration and management, while the responsibility for allocating resources and scaling the functions when needed lies with the provider. With this model the consumers are only concerned with paying for the resources they use. [25] Because of its lightweight nature that enables event-driven resource allocation, FaaS has gained big popularity in the development of microservices since it offers a cost-efficient solution.[26] The name serverless can be misleading. The code is running on a server but the server is abstracted away from the developer, and the developer needs no concern with the underlying infrastructure. Hence it is regarded as serverless from the developers point of view. [24]

2.5 Real-time need in Clouds

In fields utilising smart-manufacturing such as industry 4.0, where high availability, high reliability, and high utilisation are demanded, current cloud services are not always an optimal solution. Industry 4.0 require ultra-low latency in magnitudes of milliseconds which cannot be guaranteed by today's cloud services. Neither can real-time execution be guaranteed due to the the lack of reliable determination between processes in the cloud. To accomplish a successful Industry 4.0 rollout, different systems in a cloud environment must work seamlessly and with deterministic outcomes. [27]

2.6 Future of Cloud Computing

It is estimated that more than 50 billion devices will be connected to the internet by 2025, partly due to the growing trend of Internet of Things (IoT) devices [28]. These devices mainly use the cloud for storage. However, some newly developed Industrial IoT (IIoT) systems require subsecond responses, which needs to be addressed.

A promising field that tackles the previously mentioned issue is researched increasingly called fog computing or edge computing. The idea is to bring cloud computing to the edge of the network. In the case of IIoT, this is achieved with a datacentre in the proximity of the industry. Through this proximity of the computations, it allows for lower latencies for the applications. [29]

In [30] a platform for orchestrating resources for an ultra-low latency edge cloud was built. The platform proved feasibility for deploying and orchestrating real-time sensitive applications. Although these specialised solutions exist, there is hope that ultra-low latency requirements will still be viable with the future standard cloud orchestration software.

Chapter 3

Theory

3.1 Kubernetes

Kubernetes is an open-source platform for orchestrating containers that was released in 2015 by Google to offer a more straightforward solution for managing containerised workloads deployed to a cloud. Since then Kubernetes has grown immensely in popularity and is today becoming the standard tool used for cloud deployments [31]. Today many cloud providers offer either IaaS or PaaS with Kubernetes to their customers in order to ease the deployment and management of their applications [32].

Kubernetes allows the user automatic monitoring over the containers, and redeployment while also aiding in the initial deployment of containers [2]. In addition to this, the increased popularity of an open-source platform brings a whole array of continuously developed tools aiding Kubernetes. Therefore, a system that is not compatible with Kubernetes in the world of cloud is many times missing out on valuable tools used for cloud native development.

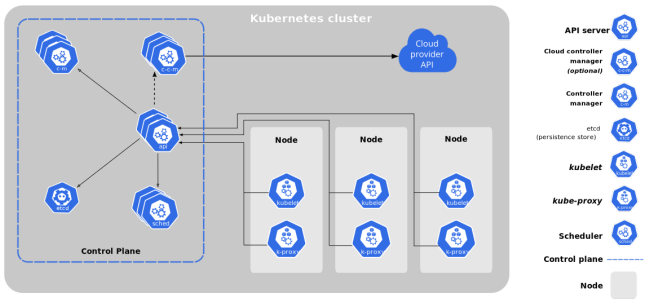


Figure 3.1: A graph of a Kubernetes cluster with three worker nodes [2]

3.1.1 Concepts

This section intends to give a brief understanding of the Kubernetes architecture. These concepts together forms a Kubernetes cluster and the core components will be discussed in in more details.

Cluster

A cluster is defined as a group of instances (physical machines, virtual machines, or containers) connected to the same network that work together to achieve a common goal, in our case provide computing capabilities. [32]

Pod

The pod is the smallest unit in a Kubernetes cluster. It is an abstraction of one or more containers and is seen as a unit of work in Kubernetes. All containers in a pod always run on the same node. All containers in a pod share the same networking and storage resources, and an application running on a pod views the pod as its host. Pods are considered to be ephemeral which means they can be torn down and built up with ease since their specification is stored as a deployment. [32]

Worker Node

A worker node is a computing instance in the Kubernetes cluster. The worker nodes main task is to compute pods deployed on the Kubernetes cluster. To achieve this every worker node contains the following three processes: kubelet, kube-proxy, and a container runtime.

Kubelet is an agent running on each node whose job is to monitor and ensure whatever is running on the node corresponds to the instructions given by the master node. Kubelet is also responsible for monitoring the status and health of running pods.

Kube-proxy is the process responsible for the networking rules of the nodes. it's job is to allow for communication with the pods running on the node from within or outside the cluster.

Container runtime is the last of the three components on the node and it is responsible for communicating with the hosts underlying kernel to run the containerised processes on each worker node. To the end user one can view the container runtime as responsible for running a container. In Kubernetes this component is responsible for running the containers that are located in the Kubernetes resource called pods. Kubernetes is compatible with the container runtimes Docker, containerd, and CRI-O, but also any container runtime with Kubernetes container runtime interface implemented. [2]

Master Node

The master node, sometimes called control plane, is the part of the cluster that monitors and manages the whole cluster. The master node can run on a single instance but also be distributed over several instances to ensure high availability when implementing larger clusters. The components in the master node include the kube-API server, kube-scheduler, etcd, and kube-controller manager.

Kube-API server is the process responsible for exposing the Kubernetes API used to control the cluster. With the API a user may for example deploy a new pod, or kill an existing one.

Kube-scheduler handles the scheduling of pods on the cluster. Once a pod has been deployed the kube-scheduler finds a suitable node for the pod to run on. When scheduling, the kube-scheduler takes into account constraints such as resource requirements, imposed pod specific constraints, and deadlines.

Etcd is a highly available key-value store for Kubernetes to store its data. Kubernetes uses the etcd to store the data and configurations of the cluster this includes the desired state of the cluster and the actual state of the cluster. Since, Kubernetes is distributed between multiple computing instances or nodes, the etcd needs to be a distributed and available on all nodes.

Kube-controller consists of five controllers combined into a single binary. The job of these five controllers includes noticing when nodes go down, watches for one-off tasks, and creates pods for these tasks to run on. It also manages the API access tokens for new name-spaces. [2]

Deployment

When creating a pod you are in fact only creating the blueprint for the pod, declared in a deployment, and the rest is handled by Kubernetes. Once the deployment is created the desired state is updated and kube-controller works to achieve this state in the cluster. It is because of this the pods can be considered ephemeral. Once a pod malfunction therefore is removed, the current state of the cluster no longer matches the desired state of the cluster and the kube-controller works to create the necessary resources for the desired state to be fulfilled again. [2]

3.1.2 Service

[2] In Kubernetes pods are seen as ephemeral. This means that a pod can be teared down and restarted without losing any of its function. When a pod is created it is given an ip-address in the cluster network. So imagine one pod called frontend being communicates and is dependent on another pod called backend. If the backend pod is torn down and afterwards restarted the frontend pod will have no reference to the backend pod anymore since its ip-address has changed. This is the problem the Kubernetes resource service solves. The service is bound to a deployment and could therefore have reference to a set of pods running an application. When a pod is restarted the same reference will hold.

How you would like to expose a set of pods for communication in a service can differ. Kubernetes allows the developers to choose from a few different service types dependent on their needs. We will discuss three of these service types briefly.

The first and default service type in Kubernetes is cluster-ip. This type exposes the set of pods only to an cluster internal ip meaning it will only be reachable from within the cluster.

The second service type is NodePort. NodePort exposes the set of pods to a static port on the nodes of the cluster. This means that the service can be reached externally at `<NODE_IP>:<NODE_PORT>`. Nodeport proxies the specified port to the the pods referenced by the service.

The third option is type LoadBalancer. The type LoadBalancer exposes the service externally using the cloud providers loadbalancer. This service then creates services of type cluster-ip and NodePort to which the LoadBalancer directs traffic. How the LoadBalancer directs traffic is decided by the cloud provider.

3.2 Fission [1]

Fission is an open source FaaS framework developed to be deployed to Kubernetes. Since it is a framework for Kubernetes many of Kubernetes concepts are used in Fission. It consists of three core concepts: functions, triggers, and environments.

Functions are a smaller piece of code (usually a single function or method) that the consumer wishes to execute in response to a certain event.

Environments are the language specific parts of Fission and is where the code is being executed. The environment contains enough software to run and compile the deployed functions code segment. The environment also has a HTTP server which allows a function to be triggered by the invocation of a HTTP request. Environments are created from a container image designed to be compatible with Fission.

Triggers are what binds an event to a certain function. It allows consumers to to connect an HTTP request to a certain URL to invoke a specific function. A figure representing the relationship between the function, environment, and triggers can be seen in figure 3.2

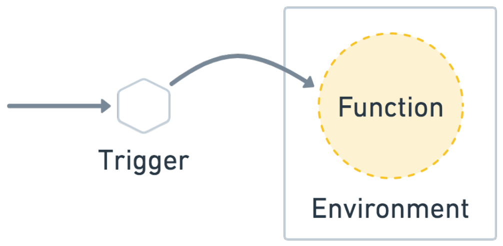


Figure 3.2: Representation of how the concepts trigger, function, and environment, relate [1]

A few of the core components that enables the three mentioned concepts will be discussed further. These core components are the executor, new-deploy, function pod, router, function service, function pod, and the function HPA.

Executor is the component in Fission that is responsible for starting new pod functions. At the moment there exists two executor types, namely pool manager and new-deploy. Pool manager is a great solution for when the consumer would like to minimize cold-start time and new-deploy is best used when a function can be exposed to heavy load since it allows for horizontal scaling.

When a function is created and needs to be executed new-deploy creates three resources. The first is the function service. This resource is a Kubernetes native resource that routes incoming traffic to a set of function pods, it therefore acts as a load-balancer for our function pods. The second resource created is the function deployment. It is this resource that makes Kubernetes create the function-pods specified by the function deployment. The third resource created is the function Horizontal Pod Autoscaler (HPA). It is this resource that allows for the horizontal scaling of pods when needed. Limitations of how much and how little the horizontal scaling can stretch can be set when creating a function.

The Fission router is a service that can be exposed to the outside of the cluster and it is this service that receives the HTTP requests specifying what function to be triggered.

A figure representation of the flow for the initial request to a function with new-deploy executor type can be seen in figure 3.3. The steps seen in this figure are as follows:

1. The router requests function service address for a function

2. Executor receives information regarding the functions resources from Kubernetes.
3. The executor invokes new-deploy to create the three resources necessary for the function using new-deploy executor type.
4. New-deploy creates the three resources
5. The service address of a function is returned to the router
6. The router redirects its traffic to the received address
7. The function service load balances and redirects traffic to a suitable function pod that executes the code.

After this sequence the function pod serves the request originating from the router service.

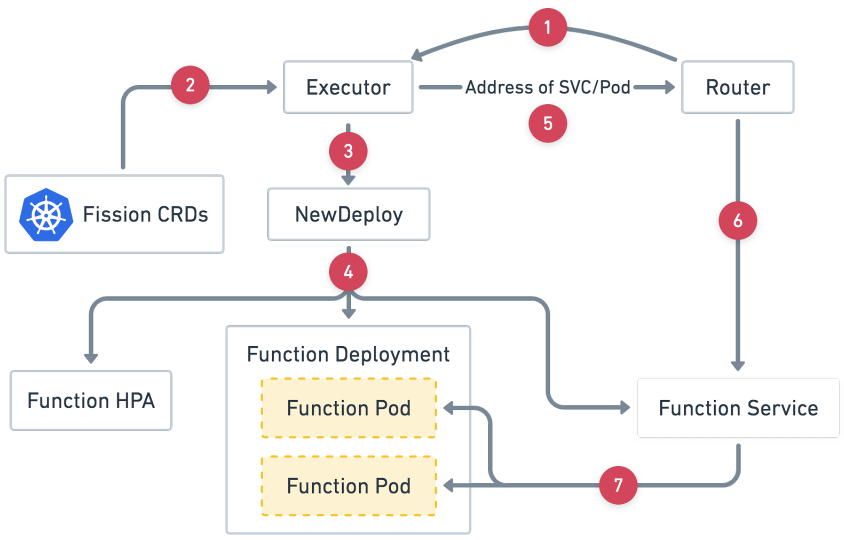


Figure 3.3: A flowchart representation of how a new function is created with executor type new-deploy. [1]

3.3 Linux

The Linux project started as a personal project by Linus Torvalds in 1991 to create a free operational system kernel. What first stared as a small project has today became by far the most popular operation system for servers deployment. This section intends to give some background on the Linux kernel and a few of the available scheduling policies available for Linux.

3.3.1 Linux Kernel [33]

The Linux kernel is the core of the OS and works as the interface between a computers hardware and its processes. More specifically, the kernel is responsible for managing memory management, process management, device drivers, and system calls and security [34]. In most cases the user has no need to modify or even understand the kernel since it works in the background.

Kernel mode is defined as the mode when an application requests services from the operating system via a system call. This means that an application is in kernel mode when the operating system is in control of the CPU. How the kernel handles this critical sections can be divided into two general groups. The non-preemptive kernel and preemptive kernel. The non preemptive means that the an application in kernel mode will run until it blocks, yields voluntarily, or exits kernel mode. The preemptive kernel allows processes to be preempted while running in kernel mode. This gives the preemptive kernel the opportunity to schedule higher priority processes even more frequently than compared to a non-preemptive kernel. This is applied to ensure even higher deterministic properties to processes with real time priorities.

3.3.2 Scheduling policies in Linux [35]

`SCHED_FIFO`: First in first out scheduling, is a preemptive real-time scheduler using the first in first out algorithm. It uses a static prioritisation between 1-99 where 1 is the lowest. As it is a real-time scheduler, it will always preempt non real-time threads. For `SCHED_FIFO` the following rules apply:

1. A running `SCHED_FIFO` thread that has been preempted by a process with higher priority will stay at the head of the list for its priority and resume when all threads with higher priority are blocked again.
2. When a blocked `SCHED_FIFO` thread becomes runnable, it will be inserted at the end of the list for its priority.

`SCHED_RR`: Round robin scheduling, is the same as `SCHED_FIFO` with the extended functionality that the thread is allowed to only run for a specified time. If a thread scheduled with round robin has been running for longer than or equal to the specified time it will be put at the end of the list for its priority. If the thread is preempted by a thread with higher priority it may only complete its remaining round robin time when allowed to run again.

`SCHED_OTHER`: Other is the default scheduler in Linux and is a non real-time policy with a static priority of 0. The thread to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice value (specified further down) and is increased for predetermined CPU period the thread is ready to run, but denied to run by the scheduler. This ensures fair progress among all SCHED_OTHER threads. SCHED_OTHER

Nice values: The nice value in Linux is a priority value to non real-time threads that influences the CPU scheduler to favor or disfavor a process. The values can be set between -19 to 20, where -19 is the highest priority. Negative priorities may only be set by the root user.

In order to set real-time priorities in Linux, the `chrt` command is used. This commands takes three arguments,

```
    $ chrt <scheduling_policy> --pid <priority> <pid>
```

where the first one, `scheduling_policy` is the scheduler to be used, `--fifo` for SCHED_FIFO, `--rr` for SCHED_RR. The second parameter is the priority, a value between 1-99, and third is the process ID (pid) of a running process. To reset a processes scheduling back to its default scheduling policy SCHED_OTHER, the scheduling parameter should be set to `--o` and the priority value has to be set to 0.

3.3.3 Limits on Real-time Resources

Since the real-time policies SCHED_RR and SCHED_FIFO will be prioritised before all SCHED_OTHER processes; a way to ensure that runaway real-time processes do not starve all other processes is needed. This can be done in two ways.

The first is to set a global limit on the resources that real-time processes can use. This is done in the path `proc/sys/kernel/` to the files `sched_rt_period_us` and `sched_rt_runtime_us`. The value set in `sched_rt_period_us` is equivalent to 100% of the CPU in milliseconds. The second file, `sched_rt_runtime_us`, is changed to a value corresponding to the allowed time real-time process can utilise the CPU and must be lower than than that in `sched_rt_period_us`. The percentage of time a real time process on one core is allowed to run corresponds to `sched_rt_period_us` divided by `sched_rt_period_us`. From this follows that setting `sched_rt_period_us` to 1s and `sched_rt_runtime_us` to 0.5s on a four core cpu corresponds to giving the real time process access to 2 CPU for 1 second and also 4 cpus for 0.5 seconds, for a period of 1 second. [36]

Another way to limit real-time processes access to resources is to use real-time group scheduling using `cgroups`. This is enabled by the flag CONFIG_RT_GROUP_SCHED while configuring the kernel. This flag enables the user to restrict the resources in the same way as previously mentioned but allowing this to be done on a given `cgroup`. A `cgroup` is a way for linux to group processes in order to limit their access to resources such as CPU-time or memory. With the use of the `CONFIG_RT_GROUP_SCHED` flag, the user can access the `cgroup` files `cpu.rt_period_us` and `cpu.rt_runtime_us`. These two files exist in a `cgroup` folder using a parent/children hierarchy where the total runtime of the children may not exceed the parents runtime. This implementation can be usefull when you want to restrict the resources consumed by a process but still giving the process the highest priority. [36]

Chapter 4

Default Kubernetes Cluster

This chapter presents the proposed solution for a baseline Kubernetes cluster with the purpose to measure execution and response time variance.The baseline cluster is used to evaluate improvements in the next chapter where various improved configurations are applied to the cluster.

4.1 Proposed Solution

4.1.1 Cloud infrastructure

Through cooperation with Ericsson there was access available to Ericssons private cloud Xerces. There are two main reasons specific to the purpose of this paper to opt for the alternative of Ericssons private cloud. The first being, Ericssons internal cloud is managed by the IaaS OpenStack which provides a high degree of freedom when configuring the architecture of the cloud's resources. This includes how the internal network is structured and also resource decisions concerning the CPU, memory and storage capacities of the nodes of the cluster. OpenStack is also widely accepted in the cloud community and this both, gives us access to a lot of community knowledge, and also enables the work of this thesis to be of more value the community. Using Xerces will also enable Ericsson to continue the work conducted in this master thesis more effectively.

Another aspect of using Ericssons private cloud Xerces is that the cloud shares the computing resources between multiple Virtual Machines. This statement is true for public cloud options as well. The way the infrastructure is designed is that a consumer requesting a computing instance is allocated a virtual machine. The virtual machine is then running on a hypervisor together with other virtual machines. This means that unless the consumer has control or can monitor what computing instances are running on the host machine, there is no way of ensuring that no other instances are responsible for loading the hosts CPU. [12]

Instead of deploying Kubernetes on virtual machines, it can be deployed directly on bare metal machines. This option ensures high reliability for an investigation regarding how a kernel is prioritising its processes because of the absence of the hypervisor. This gives full control over what processes are computed on the CPU. However, it is a nichė setup since in most cases regarding cloud computing the virtualisation is what makes it flexible and easy to scale, as described in section 2.2.

4.1.2 Loading the Cluster

A solution could be to deploy the measured processes with a HTTP server running in a container. This solution requires a load balancer to be configured since it intends to run on multiple pods. With the FaaS tool Fission the same setup was achieved while letting Fission handle the load balancing. A more detailed description of the architecture of the FaaS function Fission can be found in section 3.2. In essence this is a tool that allows for faster deployment of HTTP-servers that can run arbitrary code. Another great benefit of Fission is that it allows for the management of the deployed functions through the Fission command line interface (CLI).

With Fission the cluster can be loaded in a controlled manner by using predefined functions. For a system with a constant frequency of incoming functions, the CPU usage from the functions can be calculated using equation 4.1 where t_{load} is the execution time for the load function, n_{CPU} number of CPUs on the node, and P_{load} the period between each load call.

$$CPUusage = \frac{t_{load}}{n_{CPU} * P_{load}} \tag{4.1}$$

4.1.3 Measurements

Measuring the results of the cluster requires a setup with the possibility to measure various computation times. With the Industry 4.0 use case, it is relevant to measure the total time of an HTTP request and the total time for process execution deployed on a container. The measurement can be conducted by measuring multiple processes generated by FaaS functions. One process being the primary function that will be measured, while the other acts as a load function that will occupy computer resources. During the experiment presented in this chapter no process prioritisation will be made since the purpose of the experiment is to create a performance baseline of a cluster using native Kubernetes.

One method to measure the performance of a cluster consisting of a load function disturbing the ping function is to alter the call frequency of the load and measure performance by running the ping function, acting like a probe to take measurements and measure the Key Performance Indicators (KPIs) presented later in this chapter. In addition to this, resource constraints are made when creating functions using Fission, allowing for reduced uncertainty regarding the correlation between the variables in the test and the results by isolating parameters, which will be further explained in section 4.2.2.

Testing KPIs

With the mentioned test scenario, the following KPIs will be evaluated and used to understand the behaviour of the cluster after configuration with the aim to improve execution time variance.

- **Response time** is defined as the measured time starting from when the HTTP request for the FaaS function is sent from the client, to when the HTTP request's result is returned from the server back to the client.

- **Execution time** is defined as the measured time starting from the FaaS functions process is started at the server and ending when the process is finished.

- **Variance** is defined in equation 4.2 and is the variance in the data and will mainly be used to evaluate the execution time, where μ is the expected value for the data set, defined in equation 4.3, and x_i is a data point in the data set. For both equations, n is the length of the data set.

- **95th-percentile** is defined as the maximum value in 95% of the sorted data set and is used to measure reliability in the cluster.

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n} \tag{4.2}$$

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n} \tag{4.3}$$

4.2 Methodology

4.2.1 Architecture

Cloud environment

Xerces is managed through the cloud platform software OpenStack. This allowed for the resources on the cloud to be created through OpenStacks graphical interface where the Kubernetes cluster was designed and deployed. The cluster consisted of three computing instances on one private network. The configuration for these instances were as follows:

- 4 Virtual CPUs 2.6 GHz
- 8GB RAM
- 40GB Disk
- running OS Ubuntu 18.04

One of the VMs were to act as a controller for the cluster meaning it would be used to control the other nodes of the cluster through command line tools to monitor the Kubernetes status. One node acted as a worker node responsible for the computation of jobs deployed to the cluster and the last node deployed as the master node.

Deployment of Kubernetes

The deployment of Kubernetes to the computing instances was done with the deployment tool Kubespray. Kubespray is based on Ansible script and requires one to set up an SSH connection between the controller VM and the node VMs on the cluster. After the SSH connection was set up and the IP-addresses of the nodes were specified, Kubespray deploys a production ready Kubernetes cluster by running an Ansible script. The version of Kubespray used was v.2.15.0 and deployed Kubernetes v.1.20 with Docker v.19.3.15 as its container runtime.

4.2.2 Testing

Setting up the FaaS functions

To evaluate the performance of the cluster, two FaaS functions were deployed in order to simulate different tasks on the cluster. The first deployed function acted as a load on the cluster. The second function was used as a monitored function whose performance was evaluated, and later on improved by being scheduled using real-time schedulers in the next chapter. The two functions will be referred to as the load function and the ping function. With this setup the load functions request frequency will be varied in order to adjust for different total loads on the cluster. This function will consist of a longer execution time in range of seconds. The ping function will maintain a constant request frequency as it is only used evaluate its execution time when varying the load function. Therefore the ping function will have a much smaller execution time in order to act as a probe with a small impact on the cluster and will be in the range of milliseconds.

Fission was used to install the FaaS functions in the cluster. This requires Kubernetes CLI and Helm to be installed on the cluster and used NodePort as the routing service to direct incoming data traffic. On the controller VM, the following commands were executed:

```
$ export FISSION_NAMESPACE="Fission"
$ kubectl create namespace $FISSION_NAMESPACE
$ helm install  --namespace $FISSION_NAMESPACE \
                --name-template Fission \
                --set serviceType=NodePort,routerServiceType=\
    NodePort,logger.enableSecurityContext=true,prometheus.enabled=false \
    https://github.com/Fission/Fission/releases/download/1.12.0/Fission-all-1.12.0.tgz
```

Installing and deploying a Fission function requires three steps. First is installing the correct environment using a Fission image for the software language used. In this case, NodeJS, which is a JavaScript runtime environment. The second step is to create the Fission function from an already created code file. The third step is to route the function to a URL.

The creation of the Fission functions was done in the test-script by running the following commands:

```
$ fission env create --name node-debian --image fission/node-env-debian
```

This command creates an environment using an existing image which are accessed through Fissions docker-hub repository available online.

```
$ fission fn create --name load-func \
                    --env node-debian \
                    --code load_func.js \
                    --executortype newdeploy \
                    --minscale 5 \
```

`--executortype='poolmgr'`	Executor type for execution; one of 'poolmgr', 'newdeploy'
`--minscale=1`	Minimum number of pods (Uses resource inputs to configure HPA)
`--targetcpu=80`	Target average CPU usage percentage across pods for scaling

Table 4.1: Definitions of constraint parameters for Fission functions

```
                    --targetcpu 0.5


$ fission fn create --name ping \
                    --env node-debian \
                    --code load_func.js \
                    --executortype newdeploy \
                    --minscale 2 \
                    --targetcpu 0.5
```

The create command creates the functions which are used for the test with the parameters described in table 4.1.

```
$ fission route create --function ping --url /ping
```

The two functions consists of a simple while loop wrapped with a timer which are returned and measured. The load function uses a while loop iterating $5*10^8$ times which were constructed to have a runtime of around $1.5s$, the ping function loops 10^5 times which corresponds to around $3ms$, the size of the while loops were done empirically to match the desired execution times. The execution time is timed using the clock `process.hrtimer()`. This particular clock is preferred since it minimises clock drift and has a high resolution in nanoseconds.

4.2.3 Script

The test is automated using a script implemented in Python version 3.8. It uses the `asyncio` and `request_async` packages to allow for asynchronous testing with the following flow:

1. Load and ping functions are instantiated
    (a) Fission function with same name is deleted if it exists
    (b) Fission functions are created

   (c) Old function HTTP routes with same name are deleted if existing
   (d) HTTP routes are created
   (e) HTTP URL-triggers are created

2. Starting load function thread and running load loop
3. Starting ping function thread and running ping functions asynchronously
4. Gathering ping function execution time and response time
5. Saving the data output as a json object
6. Repeating for all specified test parameters n times

Each test takes the parameters `wait_time_ms`, later referred to as the Load Period, which is the period of how frequencly the HTTP request is sent to trigger the load function, `nbr_of_runs` which is the number of runs for each test, and the parameter `wait_time_ping_ms` which is the time the test waits between each ping function. This results in the total runtime of the test as `wait_time_ping_ms` $*$ `nbr_of_runs` since the test is constructed to stop when all ping functions have been executed. For all tests `wait_time_ping_ms` is constant and set to $2000ms$. This is chosen since the ping functions should not load the cluster but only measure it. Therefore it is not called too frequently and also has a low execution time.

The asynchronous test is needed in order to run the load and ping functions in parallel to simulate a realistic concurrent environment as it would run in an Industry 4.0 scenario.

The baseline is produced using the input parameters presented in table 4.2 where each test were run 1 time.

The values found in table 4.2 was proposed to stress to system to a sufficient degree while still not overloading the cluster by estimating the CPU usage with equation 4.1. This was later confirmed by running actual CPU measurement using the command

```
$ sar 1 500
```

This command measures the CPU usage every second for 500 iterations. After completion it gives an average of all the measured samples. The two metrics measured with this command was the percentage of steal (the time a virtual CPU is ready to run but waits for the resources of a real CPU) and percentage the CPU was idle.

`wait_time_ms`(ms)	`nbr_of_runs`
No Load	2000
2000	2000
1000	2000
500	2000
400	2000

Table 4.2: Input parameters for baseline test

Chapter 5

Modified Kubernetes Cluster

In this chapter, various modifications to the default Kubernetes cluster from previous chapter are presented and then applied to conduct three different experiments with the purpose to find different configurations which possibly reduced variance in execution time and response time. This is then followed by the method of how the three experiments are implemented.

5.1 Proposed solution

By taking advantage of the real-time properties in the Linux kernel, real-time scheduling policies can prioritise specific processes to reduce the execution time variance. To accomplish this, modifications must be done to the kernel of VMs to support real-time scheduling. If using container level virtualisation, the containers running on top of the VMs can take advantage of said scheduling policies because containers share the kernel with the underlying machine; as a result, giving the cluster and its pod's real-time support. What is interesting with these tests is to evaluate the overall performance, both in regards of execution time and response time since it is not clear how real-time support would affect the overall performance.

This chapter discusses various approaches to reduce the variation in execution time for a Kubernetes cluster when compared to the default Kubernetes configuration.

5.1.1 Reducing variance

In its default state, Kubernetes has no support for deploying clusters with real-time support since the real-time support comes from the underlying kernel. By taking advantage of the real-time properties of the Linux kernel, Kubernetes can be run with none, some, or fully support for real-time. The support for real-time can generally be grouped in two categories: preemptive scheduling and preemptive kernels which are discussed in 3.3.1 and will be used to give support for real-time applications in the cluster. By using real-time preemption, the clusters configuration can be altered and potentially reducing the execution and response time.

To reduce the variance in execution time for certain wanted processes, different experimental setups were deployed in order to evaluate the effects on execution time. The configurations used in the experiments evaluates performance on the following cluster modifications:

- non-preemptive kernel with preemptive real-time scheduling,
- preemptive kernel with preemptive real-time scheduling.

Using these two approaches, one possible modification to the cluster compared to the baseline is to take advantage of real-time scheduling policies in Linux. In the baseline, the processes are all scheduled using `SCHED_OTHER` which is a non-preemptive scheduling policy. By prioritising real-time processes and pods using a real-time scheduler such as `SCHED_FIFO`, better performance is expected.

Another setup is a cluster with preemptive scheduling but with a modified nonpreemptive kernel using `cgroups` by enabling the flag `CONFIG_RT_GROUP_SCHED`. This setup allows for the kernel to reserve CPU runtime for the prioritised real-time processes and allowing them to run undisturbed for the period specified in the process' `cgroup` file `cpu.rt_runtime_us`. If the CPU-period is not occupied by preemptive scheduled processes the resources are made available for nonpreemptive processes.

In order to allow for full preemption, the kernel must be patched with a real-time patch which is deployed in the final test setup. Using a preemptive kernel together with a preemptive scheduling policy allows the processes with prioritisation to execute with high reliability by preempting non-prioritised processes even when a process is in kernel mode.

The experiments' configurations are summarised in table 5.1 where each experiment also will be executed using the `SCHED_OTHER` scheduler which corresponds to the baseline configuration. Each test are compared to the baseline on the new kernel setup in order to allow for fair comparison of the modifications. Due to the fact that different VMs on the clusters may perform differently it is important to reduce the uncertainty from this variable by not comparing the setups between clusters.

In addition to kernel modifications and scheduling policies, support must be given to the containers on the cluster. In order to schedule processes in the container using the real-time scheduling policies the containers must enable the capability `SYS_NICE`. For a stand-alone docker container the `SYS_NICE` capability can be set with the following command:

Experiment	1	2	3
Kernel type	Nonpreemptive	Nonpreemptive `cgroup`	Preemptive
Scheduler	`SCHED_FIFO`	`SCHED_FIFO`	`SCHED_FIFO`

Table 5.1: Kernel and scheduling configurations for the various tests

```
$ docker run -it --cap-add=sys_nice \
                 fission/node-env-debian \
                 /bin/bash
```

This will create a running docker container using a Fission image with the ability to set real-time scheduling prioritisation. Although, it is not a suitable approach since it creates a stand alone container that cannot be used with Kubernetes.

In the experiment the containers should be created by Kubernetes where parameters such as capability can be configured in the deployment file. This is a potential entry point to enabling the `SYS_NICE` parameter, but also other parameters as well. Another alternative is to add parameters and capabilities to the `docker.service` file that starts all containers on a given node. This method will allow all pods to be configured with real-time scheduling. Since this is not the case in this experiment, the first method will be used to only grant selected pods the possibility to be scheduled with `SCHED_FIFO`. This is done by editing the depoylment file of the selected pod by running

```
    $ kubectl edit deployment \
                --namespace fission-function \
                <deployment-name>
```

and adding the `SYS_NICE` capability in the following way in the deployment file:

```
    spec:
      container:
        securityContext:
          capabilities:
            add:
            - SYS_NICE
```

5.1.2 Measurements and Testing

To ensure fair results when comparing the modified cluster to the default cluster setup, some aspects must be taken into account. The first aspect being that all

VMs on Xerces may not always perform identically. The total load on Xerces can vary day to day since it is a shared cloud with multiple users. It is an issue that the hosts resources are shared between multiple VMs. The second aspect is that there is no way to ensure that two VM's running at the same time of the day are granted the same resources on their host machine. To solve these problems, one solution can be to run both clusters simultaneously to eliminate the load problem. In addition to this, running both configurations on the same cluster (and removing the real-time parameters for one test instance) while alternating between the two may eliminate the uncertainty of resource allocation.

5.2 Methodology

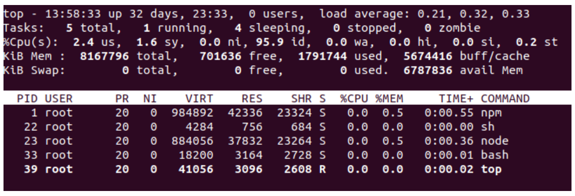
The cluster of the modified Kubernetes setup are identical to the default setup in terms of the VM specifications, resources, Fission functions, and number of nodes. Changes are made to the test script for the modified experiments where the Fission functions are *not* deployed and redeployed by the script. Instead the functions are instantiated manually once before conducting the tests. This must be done since creating new functions creates new pods and containers which inevitably removes the configurations made to existing pods. Without doing this, the prioritisation of pods would be removed with each new created pod. This section describes the added configurations present in the modified setups built on the default configuration for each one of the three experiments' cloud architecture.

For all tests the change to the test scripts and manual instantiation of Fission functions are made. The `SYS_NICE` capability is enabled using the Kubernetes deployment file for all tests as well.

5.2.1 Experiment 1 - Nonpreemptive Kernel

The setup of this experiment is conducted as for the default configuration using Ubuntu 18.04 as the guest OS with the Linux kernel version 4.16.18. This installation is handled by OpenStack during cluster setup and uses a non-modified, nonpreemptive kernel.

The preemptive scheduling policy `SCHED_FIFO` is used in order to prioritise the pods responsible of executing the Fission ping function. Prioritisation of processes requires manual modifications to the pods where the process is running and utilises the Linux kernels scheduling policies. To accomplish this, the pod names are identified using the kubernetes CTL `kubectl`. This command shows that two pods are created for the ping function where the function is executed which we wish to prioritise. The two pods are created in order to have a backup if one were to crash and are identical, and are configured using the Fission parameter `--minscale`. In order to give the pods prioritisation they need to be entered using



```
top - 13:58:33 up 32 days, 23:33,  0 users,  load average: 0.21, 0.32, 0.33
Tasks:   5 total,   1 running,   4 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.4 us,  1.6 sy,  0.0 ni, 95.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.2 st
KiB Mem :  8167796 total,   701636 free,  1791744 used,  5674416 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  6787836 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0  984892  42336  23324 S   0.0  0.5   0:00.55 npm
   22 root      20   0    4284    756    684 S   0.0  0.0   0:00.00 sh
   23 root      20   0  884056  37832  23264 S   0.0  0.5   0:00.36 node
   33 root      20   0   18200   3164   2728 S   0.0  0.0   0:00.01 bash
   39 root      20   0   41056   3096   2608 R   0.0  0.0   0:00.02 top
```

Figure 5.1: View of processes on pod before prioritisation.

the following command:

```
$ kubectl exec  --stdin \
                --tty \
                --namespace fission-function <pod-name> \
                -- /bin/bash
```

Running `top` inside the pod shows a list of running processes and their pid as demonstrated in figure 5.1. The process `node` with pid `23` is responsible of executing the code from Fission and is prioritised using the `chrt` [37] command which manipulates real-time attributes of a process.

The pid of the process that will be prioritised is passed together with the desired scheduling policy and prioritisation value, which will be put to the maximum value 99 in order to give it real-time prioritisation.

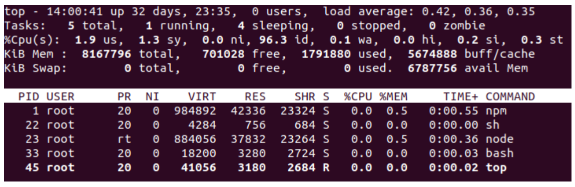
```
$ chrt --fifo --pid 99 23
```

This is done for both pods that are prioritised and the test script can be executed. After prioritisation the priority can be confirmed in the `top` view as showed in figure 5.2. Notice the priority of `node` is changed from `20` to `rt`.

With these changes applied to the cluster the test script can be executed using the same test parameters and number of runs as presented in subchapter 4.2.3. In order to measure the improvement of the newly introduced scheduler, a test was run as a comparison were all pods received the scheduling policy `SCHED_OTHER` directly after the conducted experiment. This test will be refered to as the baseline for this experiment.

5.2.2 Experiment 2 - Preemptive Scheduling with Nonpreemptive Kernel using cgroups

The second experimental setup used Ubuntu 18.04 with an altered Linux kernel version 4.16.18 where the possibility to set real-time resource limitations to



```
top - 14:00:41 up 32 days, 23:35,  0 users,  load average: 0.42, 0.36, 0.35
Tasks:   5 total,   1 running,   4 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.9 us,  1.3 sy,  0.0 ni, 96.3 id,  0.1 wa,  0.0 hi,  0.2 si,  0.3 st
KiB Mem :  8167796 total,   701028 free,  1791880 used,  5674888 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  6787756 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0  984892  42336  23324 S   0.0  0.5   0:00.55 npm
   22 root      20   0    4284    756    684 S   0.0  0.0   0:00.00 sh
   23 root      rt   0  884056  37832  23264 S   0.0  0.5   0:00.36 node
   33 root      20   0   18200   3280   2724 S   0.0  0.0   0:00.03 bash
   45 root      20   0   41056   3180   2684 R   0.0  0.0   0:00.02 top
```

Figure 5.2: View of processes on pod after real-time prioritisation.

`cpu.rt_runtime_us`in `cgroups` has been made possible to allow for controlled real-time process scheduling. This is done by enabling the flag `CONFIG_RT_GROUP_SCHED` by installing a new kernel as follows:

- downloading and unzipping the Linux kernel files (v4.16.18) on the machine and change directory to the kernel folder,
- installing the following build dependencies using `sudo apt install`

  ```
  $ build-essential
  $ kernel-package
  $ fakeroot
  $ libncurses5-dev
  $ libssl-dev
  $ ccache
  $ bison
  $ flex
  ```

  and,
- opening the kernel configuration interface by running `$make menuconfig` and enabling the following settings:
  - `Group scheduling for SCHED_RR/FIFO` located at General setup -> Control Group Support -> CPU controller
  - enable access to `.config` at General setup Kernel .config support,
- compiling and installing the kernel with the following commands:

  ```
  $ make -j20
  $ sudo make modules_install -j20
  $ sudo make install -j20
  ```

The installation of the new kernel is finished after the machine is rebooted.

Hierarchy level	1	2	3
`cpu.rt_runtime_us`	950000	475000	158333

Table 5.2: Configuration values for hierarchy in figure 5.3.

Deploying Pods

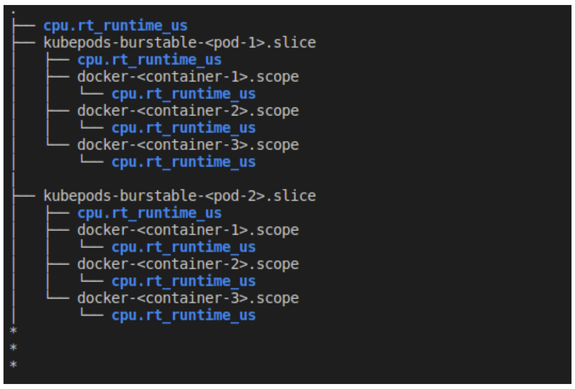
Due to the lack of real-time support in Kubernets, parts of the modified setup must be done manually. The creation of Fission functions are conducted as previously, but the test is not conducted until after all modifications are finished in order to manipulate the pods. The test script is also altered to neither remove nor create new functions as previously mentioned.

Cgroups

Whenever Kubernetes creates a pod, the pod is given a `cgroup` folder on the machine it is deployed. Here, its resources' limitations can be controlled, which is needed to allow for real-time priority of the pod's containers. In this experimental setup, the Fission function `hello` is given prioritisation by altering the `cgroup` files `cpu.rt_runtime_us`.

For each deployment of a Fission function, three containers are created and during the installation of the function in section 4.2.2, the deployment was scaled with a factor of 2, giving us two pods with three containers each. For other use cases Kubernetes and Fission may add more pods and/or containers which must be taken in account in case altering the experiment. This experiment focuses only on giving priority to the deployed Fission ping function, but other pods could also be prioritised.

The total CPU runtime reserved by the real-time processes needs to divide the total resource between containers pods for equal prioritisation (prioritisation could also be divided unequally using this method). Linux `cgroups` uses a child/parent hierarchy where the sum of the children's runtime may not exceed their parent's. In order to avoid CPU starvation, the top parent `cpu.rt_runtime_us` is set to $950000\mu s$ which is slightly lower than the maximum value of the file `cpu.rt_period_us`, which is $1000000\mu s$. This corresponds to real-time processes being able to occupy 95% of the CPU time. The total `cpu.rt_runtime_us` resource is shared between its children resulting in the distribution presented in table 5.2 with the parent/children hierarchy seen in figure 5.3



```
.
├── cpu.rt_runtime_us
├── kubepods-burstable-<pod-1>.slice
│   ├── cpu.rt_runtime_us
│   ├── docker-<container-1>.scope
│   │   └── cpu.rt_runtime_us
│   ├── docker-<container-2>.scope
│   │   └── cpu.rt_runtime_us
│   └── docker-<container-3>.scope
│       └── cpu.rt_runtime_us
│
├── kubepods-burstable-<pod-2>.slice
│   ├── cpu.rt_runtime_us
│   ├── docker-<container-1>.scope
│   │   └── cpu.rt_runtime_us
│   ├── docker-<container-2>.scope
│   │   └── cpu.rt_runtime_us
│   └── docker-<container-3>.scope
│       └── cpu.rt_runtime_us
*
*
*
```

Figure 5.3: Parent/children hierarchy for two deployed Kubernetes pods.

Prioritisation of Processes

As with the previous experiment setup, this setup requires manual prioritisation of the running pods. This is done in the same way with entering the pods using `kubectl` and running

```
$ kubectl exec  --stdin \
                --tty \
                --namespace fission-function <pod-name> \
                -- /bin/bash
```

followed by prioritising the `node` process by running

```
$ chrt --fifo --pid 99 23
```

The difference between this experiment and the previous is that when using `cgroups` it is only possible to give prioritisation to processes with `cpu.rt_runtime_us` > 0 and the prioritised processes may not execute longer than what is specified in the `cpu.rt_runtime_us` file for every period set in `cpu.rt_period_us`.

5.2.3 Experiment 3 - Preemptive Kernel

The installation of a preemptive kernel requires a kernel to be patched with a real-time extension.

This experiment uses the kernel version 4.16.18 together with the same version patch. Both packages are downloaded and kernel packages are unzipped. The kernel is then patched by running

```
$ gzip -cd ../patch-4.9.115-rt93.patch.gz | patch -p1 --verbose
```

followed by the installation of necessary packages and opening the configuration GUI by running

```
$ sudo apt-get install libncurses-dev libssl-dev
$ make menuconfig
```

In the configuration GUI the following parameters are altered in order to acquire a fully preemptive kernel:

- Check `Fully Preemptive Kernel (RT)` found in settings Processor type and features –> Preemption Model
- Uncheck `Check for stack overflows` found in Kernel hacking –> Memory Debugging

Finally the kernel is compiled by running

```
$ make -j20
$ sudo make modules_install -j20
$ sudo make install -j20
```

This installs a fully preemptive kernel and in order to take advantage of preemptive scheduling `SCHED_FIFO` is used as in previous experiments. Just as before, the desired pod should first be entered and the `node` process prioritised using the `chrt` command. After this, the test script may be executed.

5.2.4 Experiment 4 - Bare metal experiment on a single node Cluster

In order to investigate what of the previously observed results was due to running on a virtual machine environment we conducted a test on a bare metal environment. Another benefit of conducting this test on bare metal was to reduce the effects on the response time due to networking.

A Kubernetes was deployed locally on a hardware setup on bare metal that consisted of the following specifications:

- 8 CPU cores
- Intel i5-8350U CPU 1.70GHz
- 16 GB RAM

- Ubuntu 18.04
- nonpreemptive kernel

The Kubernetes cluster consisted of one master node running on the hardware setup. The cluster was controlled via the same machine due to convenience. The experiment mentioned in section was then conducted on the new local environment with two additional minor adjustments. Since the computing instance had more cores and at different speeds when compared to the previous virtual deployment, the load function which were used to stress the system had to be adjusted. This was done empirically by measuring the CPU-usage while loading the cluster and using the following command:

```
$ sar 1 500
```

The number of iterations in the while loop for the load function were then increased in order to correlate to the same load as for the cloud experiments, in the other experiments the CPU usage were 87% for the 500ms test as seen in table 6.1. The bare metal machine gained a CPU-usage of approximately 87% for the same load as previous tests when the number of iterations in the while loop were set to 10^9 loops.

The second difference from that described in section was that in this test we increased the execution time for the measured function, the ping function. This was done to increase the duration for which the function could be interrupted and therefore give a clearer result of the impact of applying `SCHED_FIFO`. The ping function was increased to loop for 5×10^7 iterations and resulted in an execution time of around 150ms.

For comparison a single node cluster was also created on Xerces. The load period was here also adjusted to achieve a 85% CPU-usage when requesting the load function every 500ms. This resulted in a load function which iterated through the while loop 4×10^8 iterations.

The developed script described in subchapter 4.2.3 was then ran on both environments with `SCHED_OTHER` and with the same load periods (2000ms, 1000ms, 500ms, and no load) and collected 400 samples of the ping function. The test was then repeated after applying the scheduling policy `SCHED_FIFO` to the ping function processes.

Chapter 6

Results

6.1 Baseline Results

The baseline test were run on the cluster with the default configuration described in section 4.2. Four tests were conducted, each with the period between each request to the load function being $2000ms$, $1000ms$, $500ms$, and one with no load function to vary the total load of the CPU from close to 0% up to around 100%. The ping was requested every 2000ms for every test. Since the ping functions job was only to measure the state of the cluster it was designed to have a small impact on the system.

In order to validate that the various load periods put the system under sufficient stress and conformed with the calculated estimates of the CPU usage found in section X. A measurement of the CPU usage described further in section X, was conducted during the four tests on the baseline cluster. In table 6.1 the measurements of the actual CPU usage for different load periods are presented together with the theoretically estimated CPU load (only taking in account the CPU usage contribution from the load functions) value calculated using equation 4.1 where t_{load} is $1.5s$, n_{CPU} is 4 cores, and P_{load} corresponds to load period in the table.

The measurement for the lowest load period $400ms$ was only conducted to see the breaking point for how high we could load our cluster. The scatter plot of this measurement can be found in figure 6.1. Note that the points at value zero indicate that the request somehow failed and an HTTP 200 response were not returned.

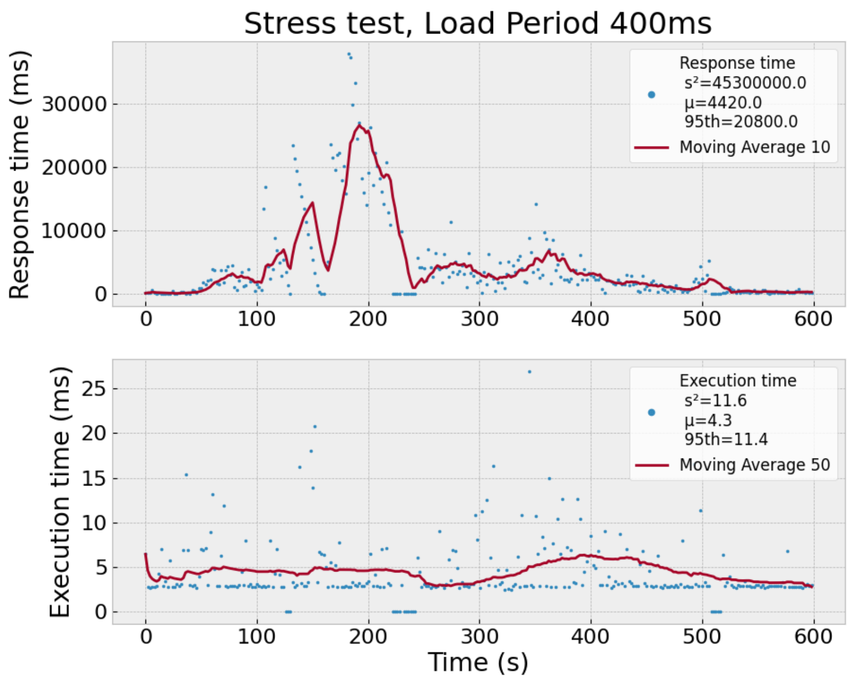


Figure 6.1: Response (upper diagram) and execution (lower diagram) time for the stress test conducted on the baseline experiment configuration. Blue points represent measured data with presented variance (s^2), mean (μ), and 95^{th} percentile in each diagram. Red solid lines represent moving average with window size 10 for response time, and 50 for execution time. Failures for a sample is represented by a value of zero.

Measured Average CPU metrics			
Load period (ms)	CPU Usage %	CPU Load (calc) %	Steal %
No load	21.34	-	0.26
2000	39.94	18.75	0.22
1000	57.22	37.5	0.15
500	87.14	75	0.08
400	99.52	93	0.03

Table 6.1: Measured average CPU usage, and Steal in percentage during the baseline measurements for the different load periods and theoretically calculated CPU usage from load functions (CPU Load).

The response time and execution time of the $2000ms$ test conducted on the baseline cluster is presented in figure 6.2 and the KPIs of all test are presented in table 6.2.

Baseline				
Load period (ms)	None	2000	1000	500
t_E σ^2	1.41	5.84	5.68	4.58
t_E σ	1.19	2.42	2.38	2.14
t_E μ (ms)	6.53	5.58	4.48	3.86
t_E $P_{95\%}$ (ms)	8.18	9.17	7.86	7.29
t_E $P_{5\%}$ (ms)	6.01	2.77	2.73	2.73
t_R σ^2	328000	11700	19500	14000
t_R σ	573	108.0	140	118
t_R μ (ms)	110	79.5	74	87.7
t_R $P_{95\%}$ (ms)	152	134.0	118	216
t_R $P_{5\%}$ (ms)	50.7	42.8	38.5	39.3

Table 6.2: Table over the variance (σ^2), standard deviation, mean (μ), 95^{th} and 5^{th} percentile ($P_{i\%}$) in execution time (t_E) and response time (t_R) for every load period

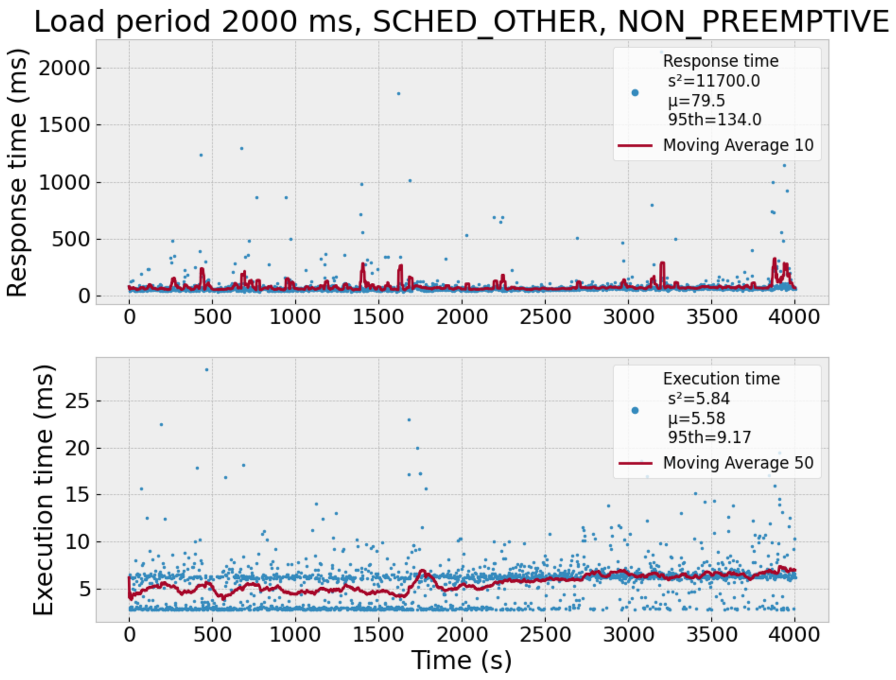


Figure 6.2: Response (upper diagram) and execution (lower diagram) time for baseline experiment. Blue points represent measured data with presented variance (s^2), mean (μ), and 95^{th} percentile in each diagram. Red solid lines represent moving average with window size 10 for response time, and 50 for execution time.

6.2 Experiment 1 - Preemptive Scheduling with Nonpreemptive Kernel

Experiment 1, as described in section 5.2.4 with the preemptive scheduler `SCHED_FIFO` and a nonpreemptive kernel were run with the wait time between the load function set to $2000ms$, $1000ms$, $500ms$, and one test with no load function. In figure 6.3 the response and execution time for the test with wait time $2000ms$ is presented.

In table 6.3 the KPIs of the four test are shown with values from both the test using a real-time scheduler `SCHED_FIFO`, and with `SCHED_OTHER` as a baseline. The baseline configuration were run on the same cluster immediately after the test using the real-time scheduler.

Nonpreemptive Kernel				
SCHED_FIFO / SCHED_OTHER				
Load period (ms)	None	2000	1000	500
t_E σ^2	0.514/1.41	2.65/5.84	2.28/5.68	1.1/4.58
t_E σ	0.717/1.19	1.63/2.42	1.51/2.38	1.05/2.14
t_E μ (ms)	6.17/6.53	4.21/5.58	3.89/4.48	3.26/3.86
t_E $P_{95\%}$ (ms)	6.53/8.18	6.35/9.17	6.31/7.86	6.17/7.29
t_E $P_{5\%}$ (ms)	5.99/6.01	2.74/2.77	2.73/2.73	2.73/2.73
t_R $\sigma^2 \times 10^4$	962/32,8	19.6/1.17	14.1/1.95	125/1,4
t_R σ	3100/573	442/108.0	376/140	1120/118
t_R μ (ms)	386/110	93.5/79.5	96.4/74	262/ 87.7
t_R $P_{95\%}$ (ms)	208/152	96.1/134.0	137/118	612/216
t_R $P_{5\%}$ (ms)	48.5/50.7	2.74/42.8	37.9/38.5	38.7/39.3

Table 6.3: Table with the variance (σ^2), standard deviation (σ), mean (μ), 95^{th} and 5^{th} percentile ($P_{i\%}$) in execution time (t_E) and response time (t_R) for every load period

In figure 6.4 the cumulative distribution function (CDF) of the experiment and baseline with load period $2000ms$ are presented.

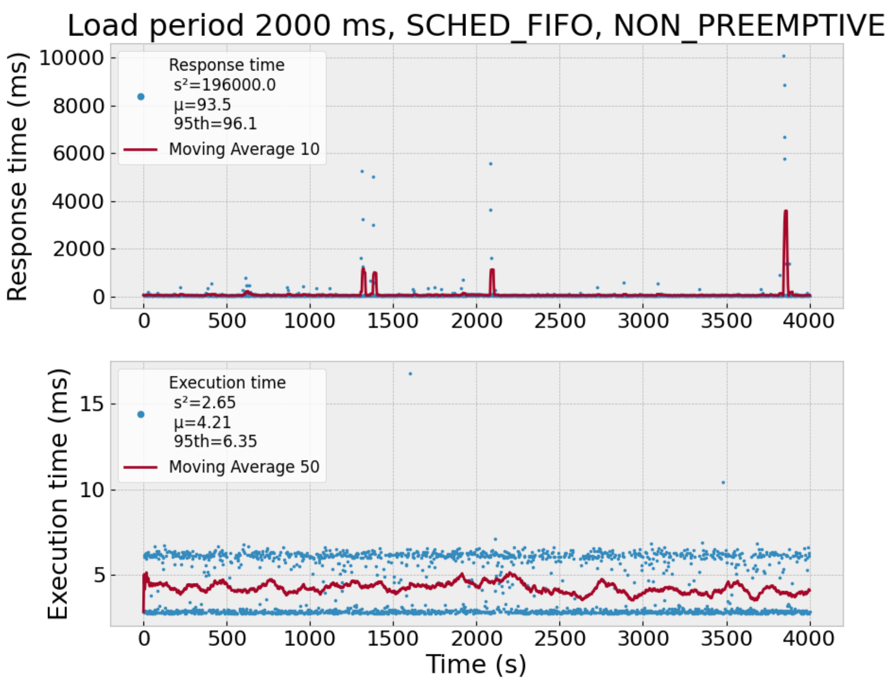


Figure 6.3: Response (upper diagram) and execution (lower diagram) time for experiment 1. Blue points represent measured data with presented variance (s^2), mean (μ), and 95^{th} percentile in each diagram. Red solid lines represent moving average with window size 10 for response time, and 50 for execution time.

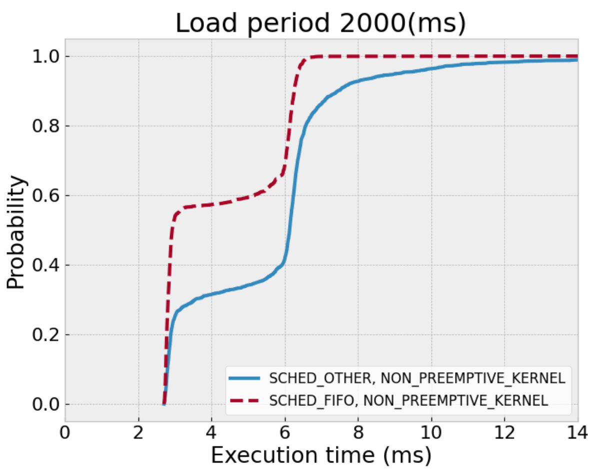


Figure 6.4: Cumulative distribution for experiment 1 with preemptive (SCHED_FIFO) and nonpreemptive (SCHED_OTHER) scheduling on a nonpreemptive kernel.

In figure 6.5 and 6.6 boxplot representation of the results from the experiment using `SCHED_FIFO` compared to `SCHED_OTHER` are shown for load period $2000ms$ and $1000ms$ respectively. In figure 6.7 boxplot of all load periods are presented using scheduler `SCHED_FIFO`.

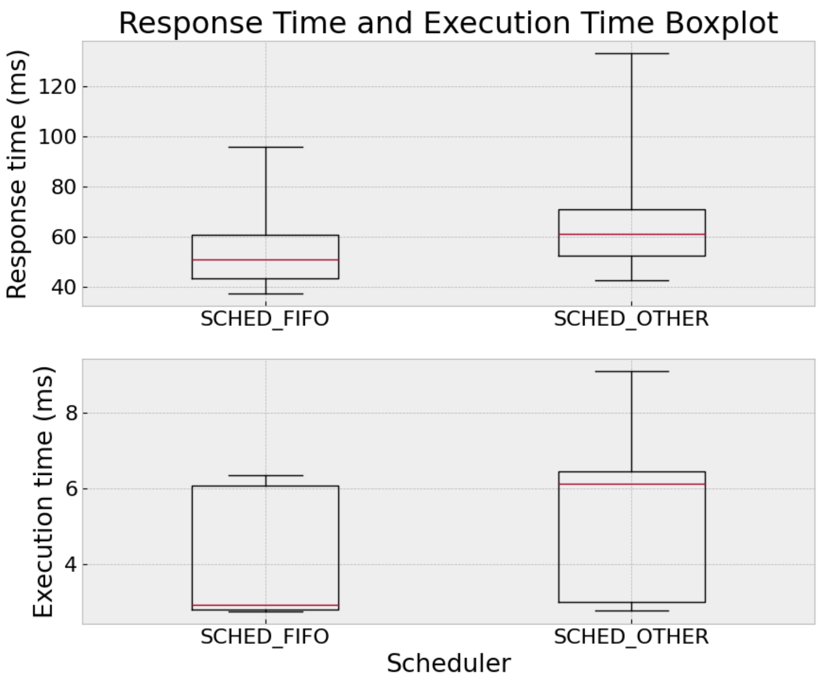


Figure 6.5: Boxplot representing the distribution of execution time and response time for experiment 1 and baseline with the load function $2000ms$ on a nonpreemptive kernel. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th} percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box" in the "box".

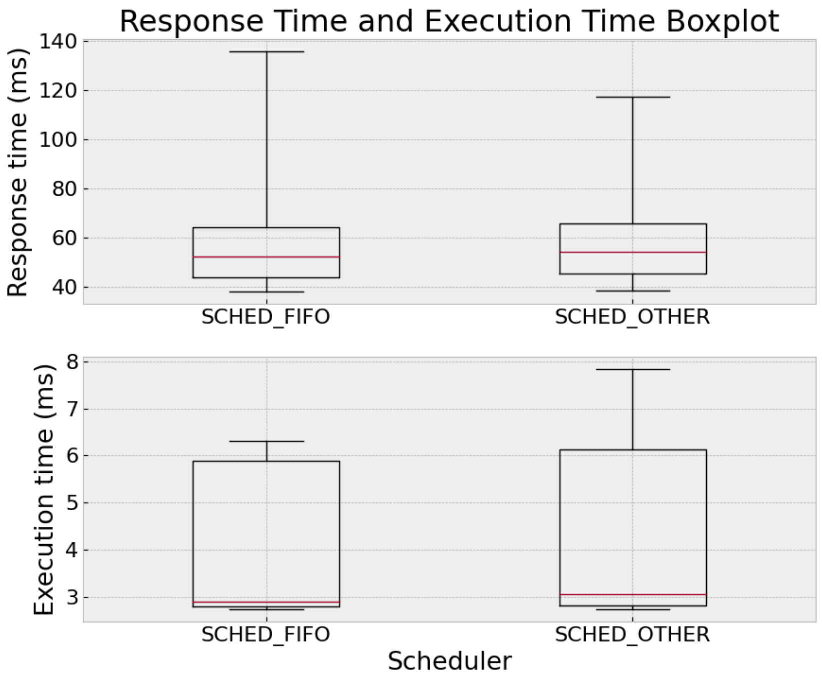


Figure 6.6: Boxplot representing the distribution of execution time and response time for experiment 1 and baseline with the load function 1000ms on a nonpreemptive kernel. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th} percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box".

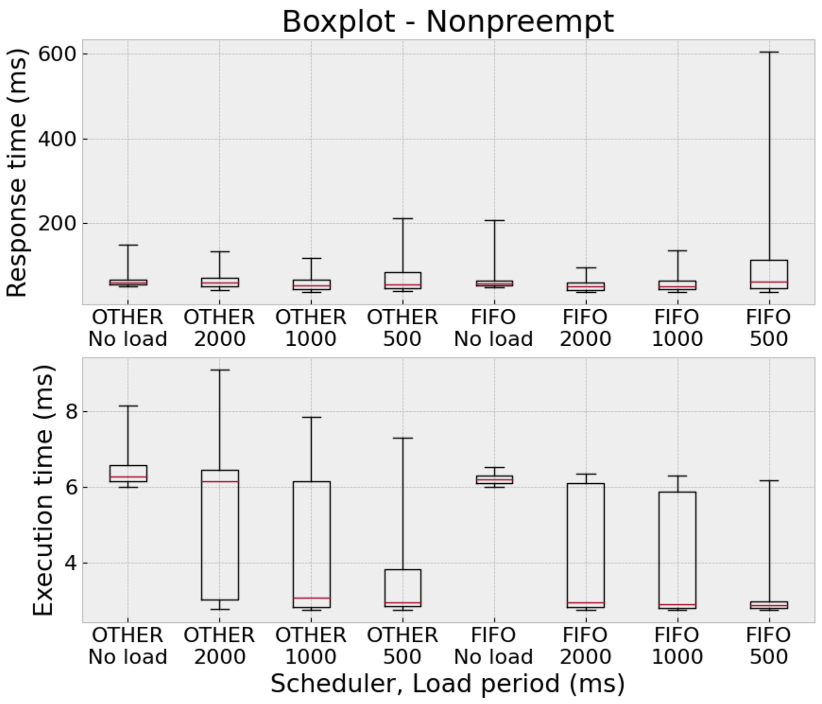


Figure 6.7: Boxplot representation of response and execution time for various load periods in experiment 1. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th} percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box".

6.3 Experiment 2 - Preemptive Scheduling with Nonpreemptive Kernel using cgroups

The second experiment, as described in 5.2.2 with a preemptive scheduler using `SCHED_FIFO` and a nonpreemptive kernel with the `CONFIG_RT_GROUP_SCHED` enabled to give support for real-time `cgroup` configurations were run with the wait time between the load function set to 2000ms, 1000ms, 500ms, and one test with no load function. In figure 6.8 the response and execution time for the test with wait time 2000ms is presented.

In table 6.4 the KPIs of the four test are shown for both the real-time scheduler `SCHED_FIFO`and the baseline with `SCHED_OTHER` which ran on the cluster directly after on the same kernel to eliminate uncertainty from kernel modifications.

Nonpreemptive Kernel with Cgroup limitations				
SCHED_FIFO / SCHED_OTHER				
wait_time (ms)	None	2000	1000	500
t_E σ^2	1.36/1.88	1.47/4.75	1.76/5.35	1.03/4.81
t_E σ	1.17/1.37	1.21/2.18	1.33/2.31	1.02/2.19
t_E μ (ms)	5.96/6.25	3.47/4.62	3.55/4.44	3.21/3.85
t_E $P_{95\%}$ (ms)	6.67/7.77	6.19/7.75	6.26/8.02	6.09/7.24
t_E $P_{5\%}$ (ms)	3.04/2.96	2.74/2.74	2.73/2.74	2.73/2.73
t_R $\sigma^2 \times 10^4$	3.7/2.48	0.436/0.742	21.9/9.64	6.43/3.67
t_R σ	192/157	66/86.1	148/310	254/192
t_R μ (ms)	71.8/85	57.1/67.8	71.3/86.5	107/99.7
t_R $P_{95\%}$ (ms)	85.3/188	85.5/115	145/124	307/235
t_R $P_{5\%}$ (ms)	41.6/42.6	31.6/38.6	31.6/38.7	35.3/38.8

Table 6.4: Table over the variance (σ^2), standard deviation, mean, 95^{th} and 5^{th} percentile ($P_{i\%}$) in execution time (t_E) and response time (t_R) for every load period

In figure 6.9 the CDF of the experiment and baseline with load period 2000ms is presented.

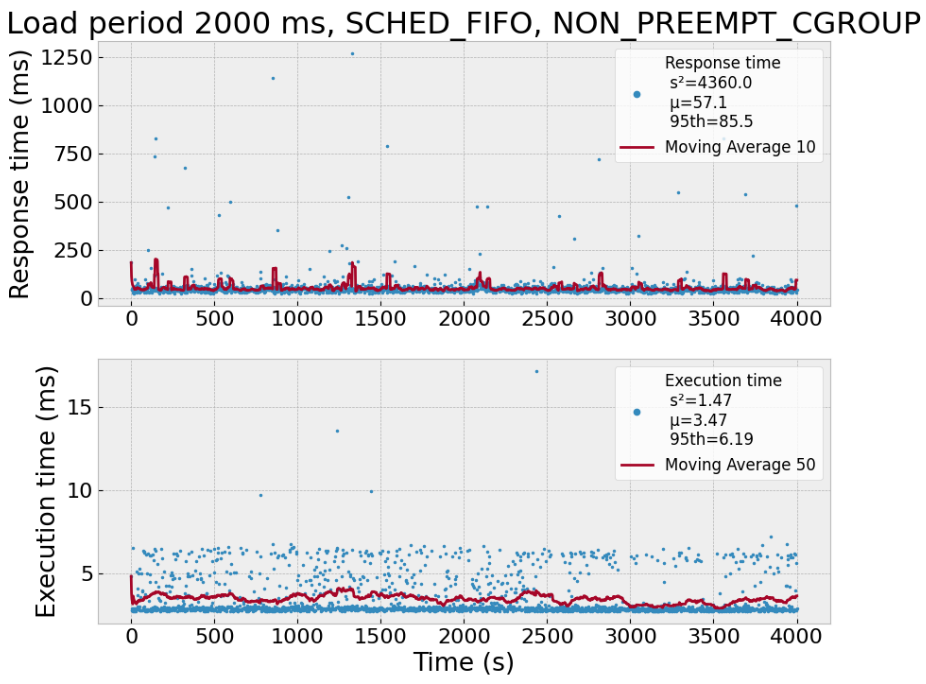


Figure 6.8: Response (upper diagram) and execution (lower diagram) time for experiment 2 with cgroups. Blue points represent measured data with presented variance (s^2), mean (μ), and 95^{th} percentile in each diagram. Red solid lines represent moving average with window size 10 for response time, and 50 for execution time.

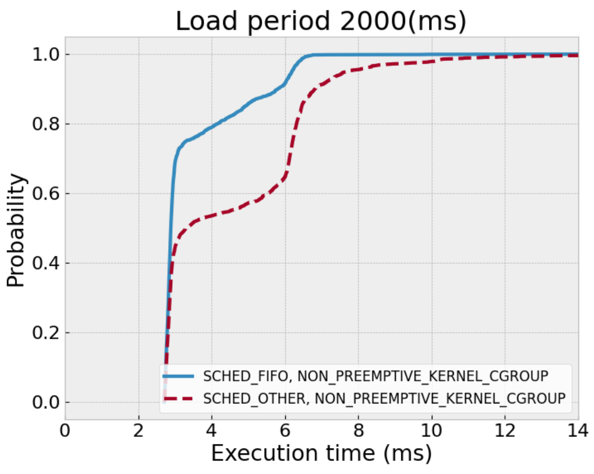


Figure 6.9: Cumulative distribution of preemptive (SCHED_FIFO) and nonpreemptive (SCHED_OTHER) scheduling on a nonpreemptive kernel using cgroups.

In figure 6.10 boxplot representation of scheduler `SCHED_FIFO` compared to `SCHED_OTHER` is shown for load period $2000ms$, and in figure 6.11 a boxplot of the experiment with the `SCHED_FIFO` scheduler is presented for every load period.

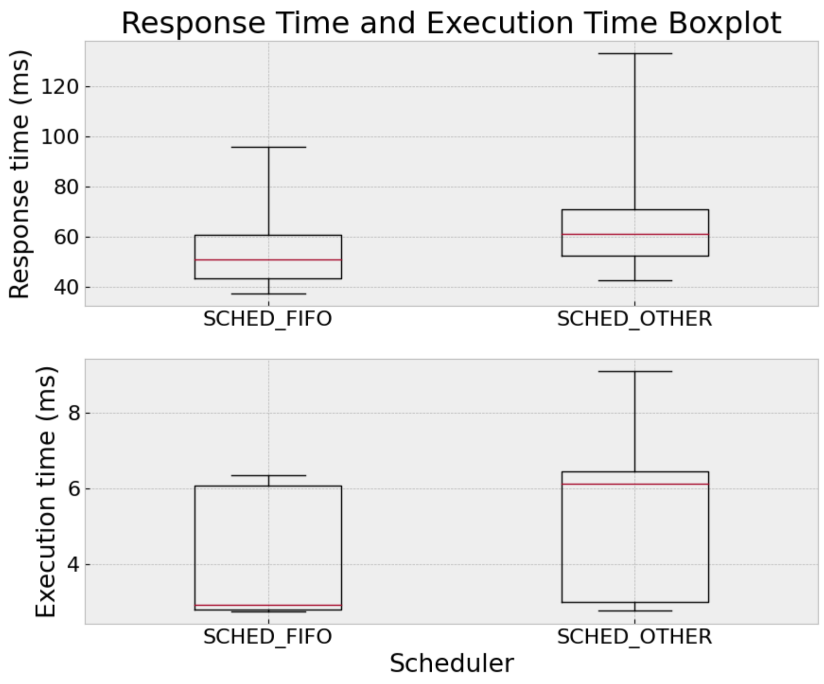


Figure 6.10: Boxplot representing the distribution of execution time and response time for experiment 2 and baseline with the load function $2000ms$ on a nonpreemptive kernel using cgroup. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th} percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box".

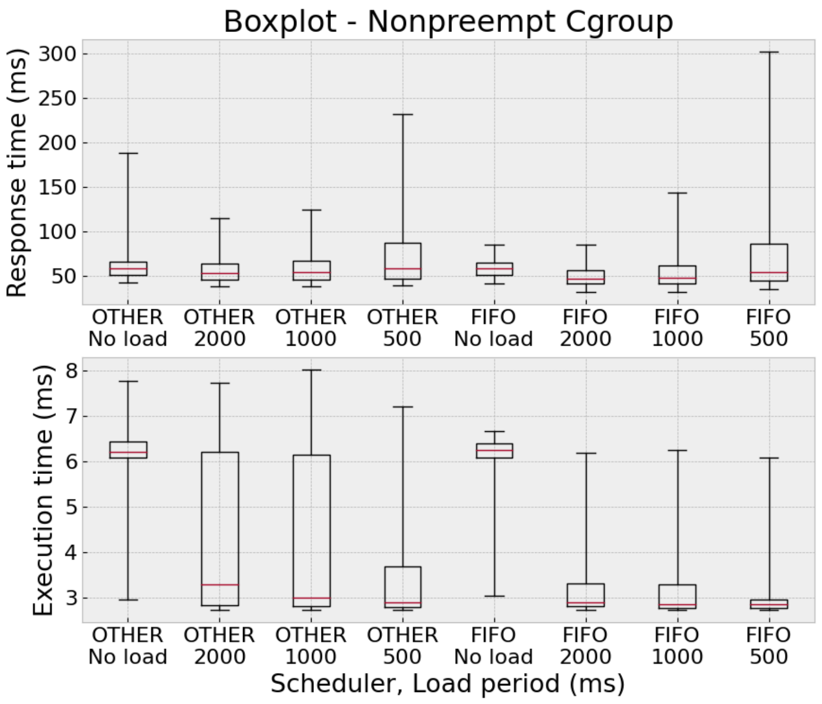


Figure 6.11: Boxplot representation of response and execution time for various load periods in experiment 2. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th} percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box".

6.4 Experiment 3 - Preemptive Scheduling with Preemptive Kernel

The third experiment, as described in 5.2.3 with the preemptive scheduling policy `SCHED_FIFO` and a preemptive kernel the tests were run with the wait time between the load function set to $2000ms$, $1000ms$, $500ms$, and one test with no load function. In figure 6.12 the response and execution time for the test with wait time $2000ms$ is presented.

In table 6.5 the KPIs of the four test are shown as well as the baseline values ran on the cluster directly after.

Preemptive Kernel				
SCHED_FIFO / SCHED_OTHER				
wait_time (ms)	None	2000	1000	500
t_E σ^2	1.49/3.72	3.09/6.72	1.58/4.23	0.524/3.79
t_E σ	1.22/1.93	1.76/2.59	1.26/2.06	0.724/1.95
t_E μ (ms)	6.18/6.5	5.08/5.21	3.52/4	3.06/3.74
t_E $P_{95\%}$ (ms)	7.42/9.56	7.3/9.53	6.38/7.63	4.02/7.17
t_E $P_{5\%}$ (ms)	2.94/2.93	2.77/2.77	2.74/2.72	3.06/2.72
t_R $\sigma^2 \times 10^4$	0.912/0.7	0.381/0.375	1.37/0.401	3.74/8.74
t_R σ	95.5/83.7	61.7/61.2	117/63.3	193/296
t_R μ (ms)	105/102	99.3/89.1	92.5/82.7	130/158
t_R $P_{95\%}$ (ms)	198/192	192/157	182/159	362/452
t_R $P_{5\%}$ (ms)	60/61.2	52.6/49.8	47/48.1	51.6/51

Table 6.5: Table over the variance (σ^2), standard deviation, mean, 95th and 5th-percentile ($P_{i\%}$) in execution time (t_E) and response time (t_R) for every load period

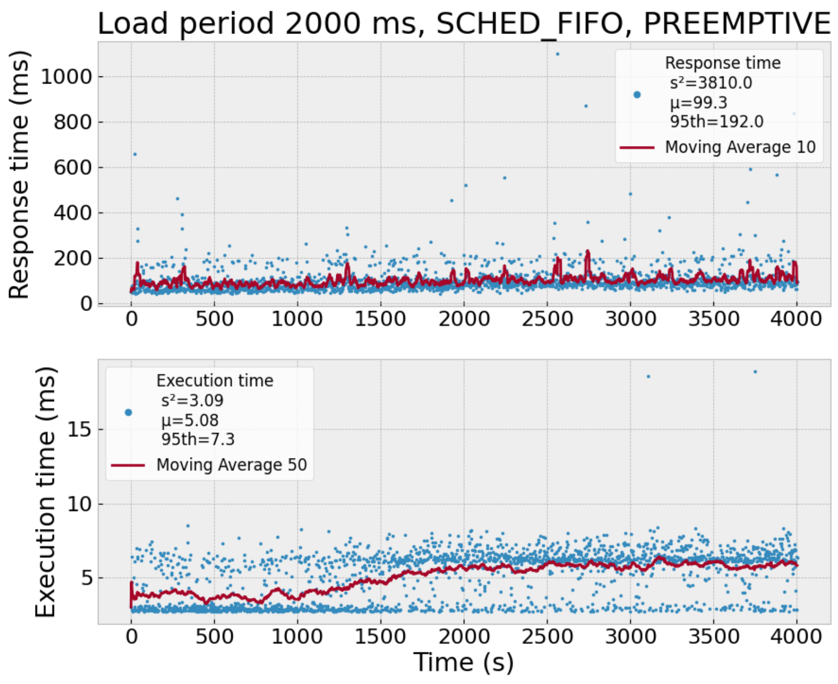


Figure 6.12: Response (upper diagram) and execution (lower diagram) time for experiment 3. Blue points represent measured data with presented variance (s^2), mean (μ), and 95^{th} percentile in each diagram. Red solid lines represent moving average with window size 10 for response time, and 50 for execution time.

In figure 6.13 the CDF of the experiment and baseline with wait time $2000ms$ are presented.

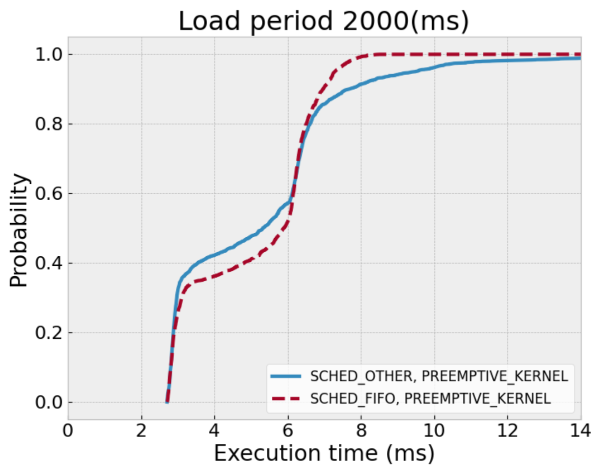


Figure 6.13: Cumulative distribution of preemptive (SCHED_FIFO) and nonpreemptive (SCHED_OTHER) scheduling on a preemptive kernel.

In figure 6.14 a modified box plot for the experiment and baseline with wait time $2000ms$ is represented. The lower whisker on the boxplot represents the 5th percentile and the upper the 95^{th}. The rectangle starts at the 25^{th} percentile and ends at the 75^{th}. The 50^{th} percentile is represented by a line in the rectangle.

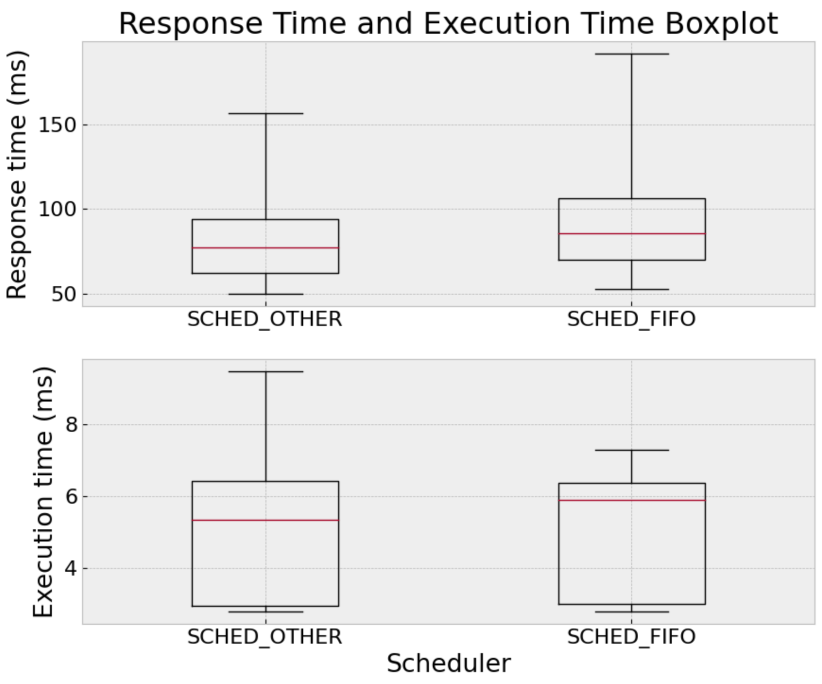


Figure 6.14: Boxplot representing the distribution of execution time and response time for two schedulers ran with the load function 2000ms on a preemptive kernel. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th}-percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box".

In figure 6.14 and 6.15, a modified box plot for the experiment and baseline with wait time 2000ms and the experiment with every load period is represented respectively. The lower whisker on the boxplot represents the 5^{th} and the upper the 5^{th}-percentile. The rectangle starts at the 25^{th} percentile and ends at the 75^{th}. The 50^{th} percentile is represented by a line in the rectangle.

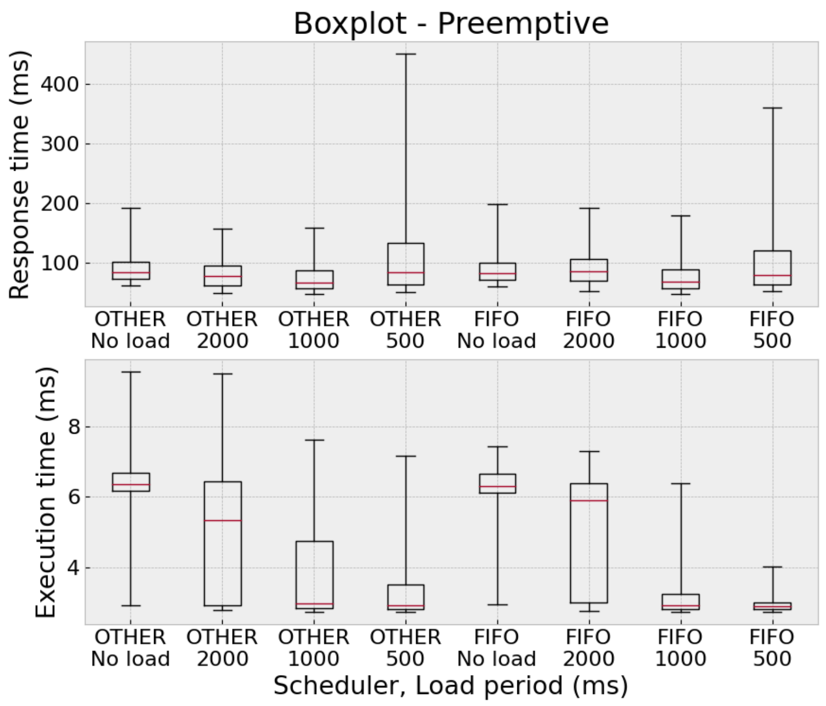


Figure 6.15: Boxplot representation of response and execution time for various load periods in experiment 3. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th} percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box".

6.5 Comparison

In figure 6.16 a comparison of the cumulative distribution function for the measurements conducted with `SCHED_OTHER` and the load period 2000ms on the different kernel and kernel modifications is presented to evaluate potential performance differences on various kernels.

In figure 6.17 a comparison of the cumulative distribution function for the measurements conducted with `SCHED_FIFO` and the load period 2000ms on the different kernel and kernel modifications is presented, and in figure 6.18 the same for load period 500ms is presented. Since the behaviours in figure 6.16 is very similar regarding the upper percentiles we can ensure that the comparison in figure 6.17 is fair when considering the upper percentiles. In addition the distribution for the different experiments are presented in figure 6.19

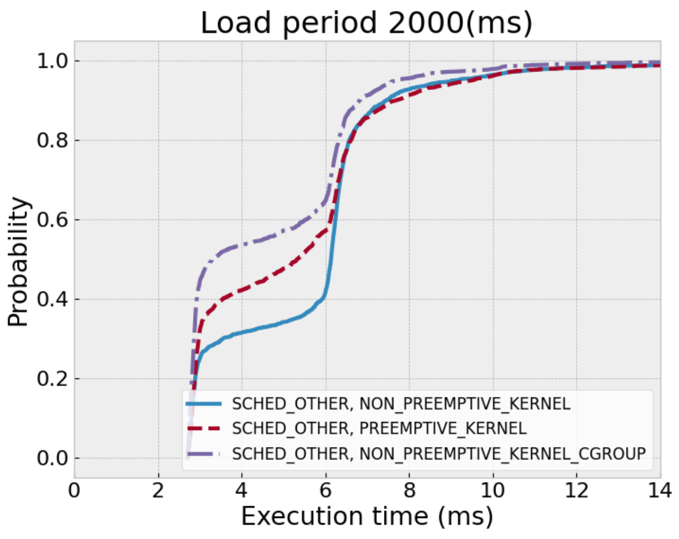


Figure 6.16: Comparison of cumulative distribution of baseline configurations on different kernels and for load periods 2000 ms.

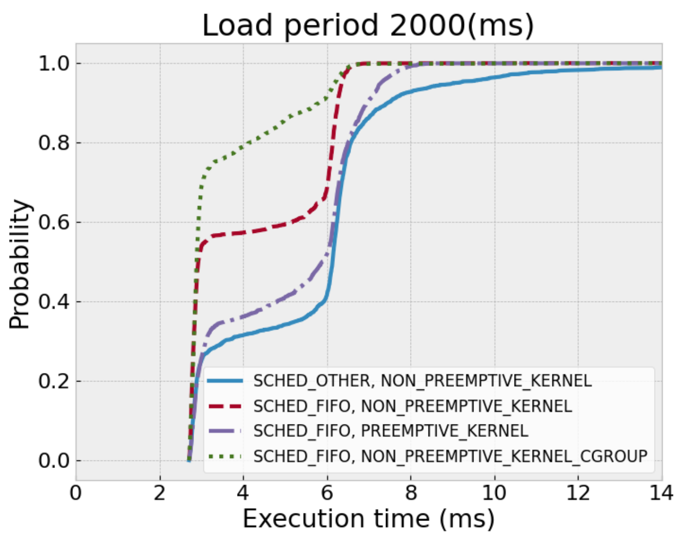


Figure 6.17: Comparison of cumulative distribution preemptive scheduling (SCHED_FIFO) of different kernels for load periods 2000 ms.

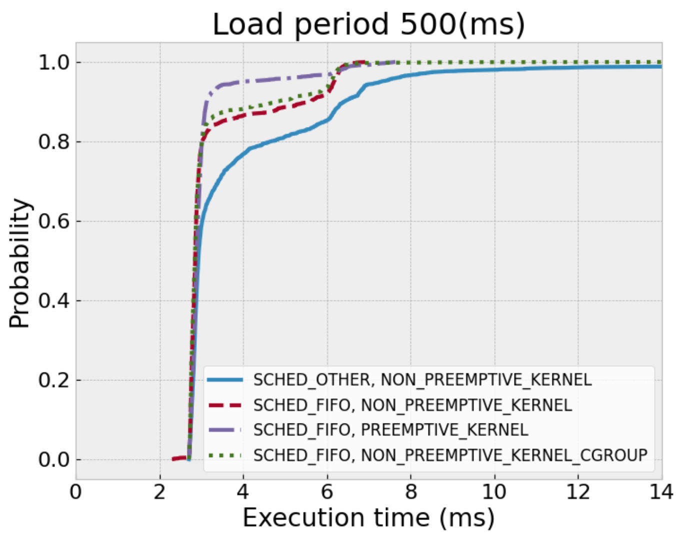


Figure 6.18: Comparison of cumulative distribution preemptive scheduling (SCHED_FIFO) of different kernels for load periods 500 ms.

6.6 Single Node Experiment

Additional tests were run in order to investigate the performance of execution and response time on a cluster only consisting of one node, a master node. One of the

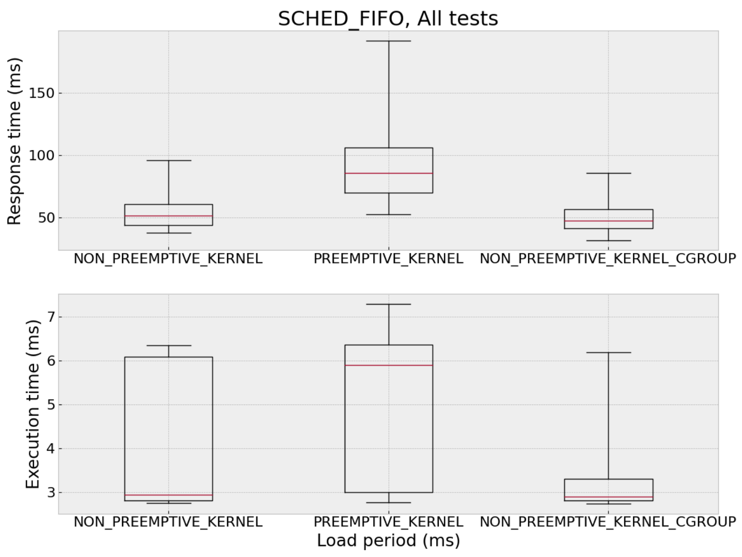


Figure 6.19: Comparison of boxplot for all experiments using SCHED_FIFO and load period 2000.

test were run on a single node bare metal configuration, and the other were run on a single node cloud on the private cloud Xerces. This was done to investigate the performance on bare metal with a comparable cloud test.

The hardware setup on bare metal consisted of the following specifications:

- 8 CPU cores
- Intel i5-8350U CPU 1.70GHz
- 16 GB RAM
- Ubuntu 18.04
- nonpreemptive kernel

The test output for the KPI's for the bare metal configuration are presented in table 6.6 and a boxplot presenting the response and execution time for load periods $2000ms$, $1000ms$, $500ms$, and no load period are presented in figure 6.20. The test for the cluster with only one node are presented with KPI's in table 6.7 with the corresponding boxplot in figure 6.21.

These tests are not identical to the previous tests. The main difference here is that the execution time was increased to about $150ms$ and that the load function which were loading the clusters were increased to correspond to about 80% load when running with a load period of $500ms$. This was done due to different hardware present on the bare metal machine.

Single Node Bare Metal				
SCHED_FIFO / SCHED_OTHER				
wait_time (ms)	None	2000	1000	500
t_E σ^2	12.5/13.2	16.8/59.5	22.3/30.4	21.9/52.7
t_E σ	3.54/3.64	4.10/7.71	4.72/5.52	4.68/7.26
t_E μ (ms)	122/124	142/146	162/166	163/169
t_E $P_{95\%}$ (ms)	129/131	147/161	189/174	168/203
t_E $P_{5\%}$ (ms)	119/120	138/140	158/161	160/176
t_R σ^2	31.6/22.4	30.3/89.8	103/110	94.6/316
t_R σ	5.63/4.73	5.50/9.47	10.2/10.5	9.72/17.8
t_R μ (ms)	134/135	154/159	179/184	181/188
t_R $P_{95\%}$ (ms)	142/143	160/176	189/196	193/203
t_R $P_{5\%}$ (ms)	129/130	150/151	172/176	173/176

Table 6.6: Table over the variance (σ^2), standard deviation, mean, 95^{th} and 5^{th}-percentile ($P_{i\%}$) in execution time (t_E) and response time (t_R) for every load period executed on a single node cluster in a bare metal environment

Single Node on Xerces				
SCHED_FIFO / SCHED_OTHER				
wait_time (ms)	None	2000	1000	500
t_E σ^2	1390/2020	2020/2980	2110/2590	1010/1370
t_E σ	37.3/44.9	45.0/54.6	46/50.9	31.8/37.1
t_E μ (ms)	246/268	194/200	183/191	161/179
t_E $P_{95\%}$ (ms)	311/341	267/308	268/297	229/255
t_E $P_{5\%}$ (ms)	193/198	141/143	140/141	140/143
t_R σ^2	2480/3150	3590/9020	11600/6220	6000/4830
t_R σ	49.8/56.1	59.9/95	108/78.9	77.5/69.5
t_R μ (ms)	300/325	241/251	242/238	225/238
t_R $P_{95\%}$ (ms)	366/402	323/378	357/356	326/341
t_R $P_{5\%}$ (ms)	245/252	166/171	166/168	170/178

Table 6.7: Table over the variance (σ^2), standard deviation, mean, 95^{th} and 5^{th}-percentile ($P_{i\%}$) in execution time (t_E) and response time (t_R) for every load period executed on a single node cluster on the private cloud Xerces

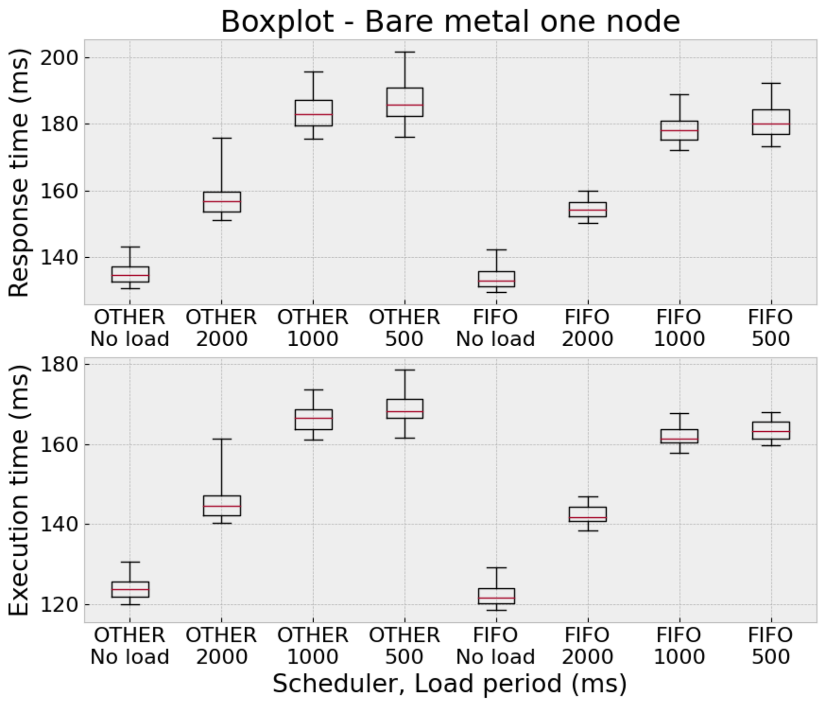


Figure 6.20: Boxplot representing the distribution of execution time and response time for two schedulers ran with the load function 2000ms on a nonpreemptive kernel on bare metal with only one node. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th}-percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box".

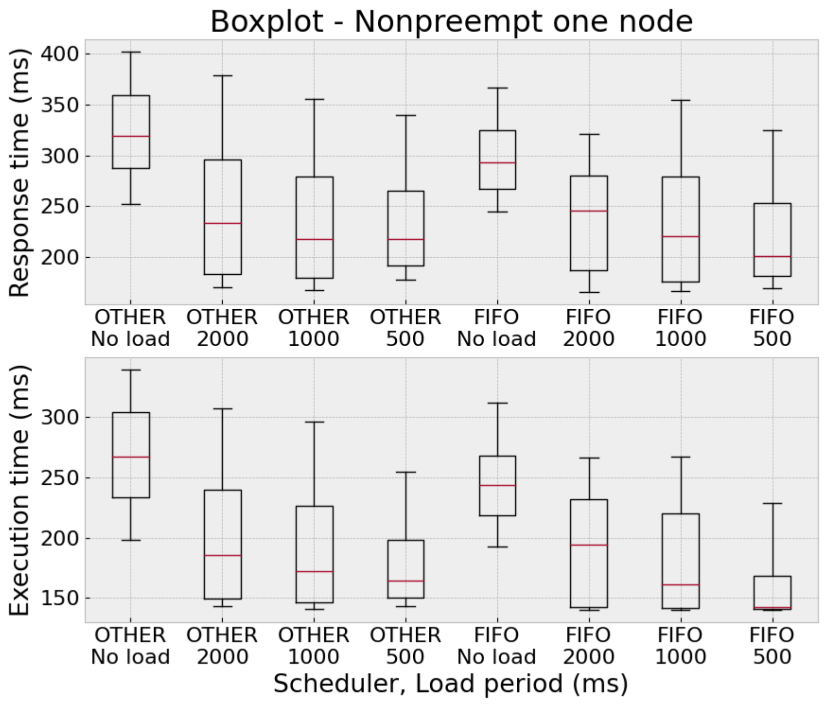


Figure 6.21: Boxplot representing the distribution of execution time and response time for two schedulers ran with the load function 2000ms on a nonpreemptive kernel with only one node. The whiskers upper and lower boundaries represent the 95^{th} and 5^{th}-percentile respectively and the upper and lower boundary for the box represents the 75^{th} and 25^{th}-percentile respectively. The median is represented with a line in the "box".

Chapter 7

Discussion

The experiments that were conducted were designed in a way to gradually increase the load of the cluster from no load up until close to 100% load which can be seen in table 6.1 where the real CPU usage is shown together with the predicted calculated value. The difference between the real value and calculated value is to be expected since the calculated value only takes in account an estimated execution time of $1.5s$ for the load function. This would in reality experience some variance that would load the CPU further. In addition to this, the machine uses other processes than the running function as well such as pods responsible for networking and communicating with Kubernetes which adds additional overhead.

The last run, with a load period of $400ms$ took use of almost all of the CPU, which were in line with the calculated estimation of a CPU usage contribution of 93% from the load functions. It could be predicted that this configuration would have full CPU utilisation due to overhead from other present processes. This prediction were correct when consulting figure 6.1 where it can be seen the system is not able to finish all of its incoming tasks with some response and execution times equal to zero. Zero values are values corresponding to an error message. Due to this, all future experiments only show results for experiments with no load, and load periods, $2000ms$, $1000ms$, and $500ms$ where the cluster is able to handle all incoming tasks.

Default Baseline

While analysing the results in table 6.2 it was seen that the baseline measurements resulted in the 95^{th} percentile measuring to $8.18ms$, $9.17ms$, $7.86ms$, $7.29ms$ for the no load and $2000ms$, $1000ms$, and $500ms$ load period respectively, this for a group of 2000 samples. Comparing this to the 5^{th} percentile gives us a difference of $2.17ms$, $6.4ms$, $5.13ms$ and $4.56ms$. This difference can be seen as the unwanted delay in execution time intended to mitigate. It is interesting to note that no observations of a reduction in execution time due to an increased load period could be seen but rather the opposite, a decrease in execution time due to the increased load. The fact that a lower load period corresponds to an actual increase in load on the CPU was also verified through measurements on the CPU load on the underlying node shown in table 6.1. One reasoning why no observations of an increase in execution time is that the incoming processes are also scheduled under the policy `SCHED_OTHER`. When consulting the Linux manual [35] it is stated that a `SCHED_OTHER` process do not have the authority to preempt another `SCHED_OTHER` process during its allocated time slice. This would then mean that once a process has started to execute it is allowed to finish during its given time slice. It is important to note that this cannot be verified with the baseline experiment alone. Although, this could be a potential explanation for why the execution time does not increase for higher loads, it cannot explain the decrease in execution time for the higher load.

A possible explanation for this behaviour could be that `SCHED_OTHER` uses the completely fair scheduling (CFS) algorithm. This algorithm was developed by the Linux community for normal tasks on a desktop computer and is therefore geared towards a higher user experience. This means that the scheduling policy prioritises interactive tasks. The way the scheduler decides which tasks are interactive is dependent on the time a task is sleeping since it is therefore likely to wait for an input. [11] [38] This could, in our case, result in that the load function is treated as an interactive task by the processor, which would then give higher priority to the load function for tests with a higher interval between the load function requests. A higher priority for the load, would mean a relatively lower priority for the ping function. However, this cannot be an explanation for why the test with no load performs worse than for the highest load period ($500ms$). A potential explanation for this behaviour occurs was examined by a test producing the findings presented in table 6.1. In this table it can be seen that the CPU steal percentage decreases as the load on the CPU increases. This means that the virtual CPU is waiting on average for a shorter duration to be scheduled on the actual host's CPU for higher loads. This could result in a higher execution time for lower virtual CPU loads. This statement would preferably be confirmed by running a similar test on a bare metal environment.

When analysing the response time in table 6.2 and in figure 6.7, an increased response time for when the load is increased (load period, i.e. period between incoming loads happen more frequently) can be seen. This was expected and can be explained by the fact that processes are idle while waiting for resources on the CPU. Another aspect that is important to mention is that the response is dependent on a large set of processes and is therefore not as easily argued for. In section 3.2 it is describer how Fission handles an incoming HTTP request. All these processes are affecting the response time and in addition to this there are networking effects from the private cloud Xerces present. With this said it is an important metric to take into account when using an alternative scheduling policy.

The variance in the execution time in the baseline experiment is likely due to

Linux processes with higher priority than the priority of the ping function interrupting this function. In order to investigate this the upcoming three experiments that utilise the real-time scheduler `SCHED_FIFO` are conducted. It would then be expected that the execution time variance would be lower when compared to the baseline since the ping function would always have priority over processes with a prioritised `nice` value, and have equal priority to other real-time processes in the kernel scheduled with `SCHED_FIFO` or `SCHED_RR`.

Experiment 1

The first conducted experiment to optimise performance, experiment 1, with the scheduling policy `SCHED_FIFO` on a nonpreemptive kernel's response time and execution time can be seen in figure 6.3 for the test with load period $2000ms$. It can be seen that the execution time is pretty much stable around $2.8ms$ and $6.2ms$. These two separate levels may be a result from the hypervisor allocating the resources between various VMs on our and other clusters on the cloud network and may be dependent on the total load and scheduling policy on the hypervisor in the cluster [39]. This is suspected since the behaviour were not consistent during the development of the experiments and occurred from time to time and could not be replicated in a controlled manner. Sometimes this behaviour were not present. One way to verify if this behavior is from running multiple VMs on the cluster, is to deploy our cluster on bare metal. In that case, the behavior can be expected to disappear. Another approach could be to run an isolated cloud with only one VM to guarantee no other VMs are disturbing. This will further be discussed in subsection Bare metal.

Due to these distinct levels, the variance (σ^2, or s^2 in the figure) cannot be used to evaluate the performance of the experiment by itself since the distribution between the lower and upper levels could vary from day to day because of the total load on the cloud Xerces which affects variance and mean.

Together with the variance, the 95^{th}-percentile value is therefore also used to evaluate if the experiment behaves better than the baseline on the same kernel by verifying that 95% of the time, the measured values are not too far from the mean execution time. The outliers are mostly located far away from the mean.

In the experiment, the execution time is indeed lower than the baseline and not much higher than the previously mentioned stable level at 6.2ms as seen in figure 6.3 where the 95^{th} is $6.35ms$. In table 6.3 it can be seen that the execution time showed an improvement (a reduction) for all presented KPIs in the table. Additionally, for a more graphic verification of the improved performance the CDF in figure 6.4 and distribution using boxplot in figure 6.5 for the experiment with load period $2000ms$ can be examined. In the CDF diagram, it can be seen that `SCHED_FIFO` always lies left to `SCHED_OTHER` (the baseline on the same kernel), which implies a lower value for all percentiles. It should be noted that the middle part of the slope, where the derivative is reduces and again increased is due to the two distinct levels of execution time. The boxplot gives a visual verification that the 95th-percentile is indeed lower for `SCHED_FIFO` and that the 25th and 75th are slightly lower for the experiment. A high reduction in the median can also be seen, but this is not necessarily due to the scheduler since it can be the effect of multiple clusters on Xerces.

These results show that the variance, percentiles, median, and mean for execution time in experiment 1 are improved. This can be expected since the pods responsible of executing the FaaS functions are prioritised.

In contrary, the measured KPI's in table 6.3 for the response time shows a degradation (an increase) with the implementation of the real-time scheduler for all values except the 95th percentile for the test with load period (wait_time in the table) 2000ms. This particular measurement is the one showed in the plot in figure 6.3 and boxplot in figure 6.5. The boxplot shows an overall reduction for this measurement, but for the other measurements in the last mentioned table, the values are all increased. This could indicate that this particular measurement is only a one time occurrence due to unknown parameters. In figure 6.6 the boxplot of `SCHED_FIFO` compared to `SCHED_OTHER` for the configuration with a load period of $1000ms$ is shown, where a increase in response time can be seen.

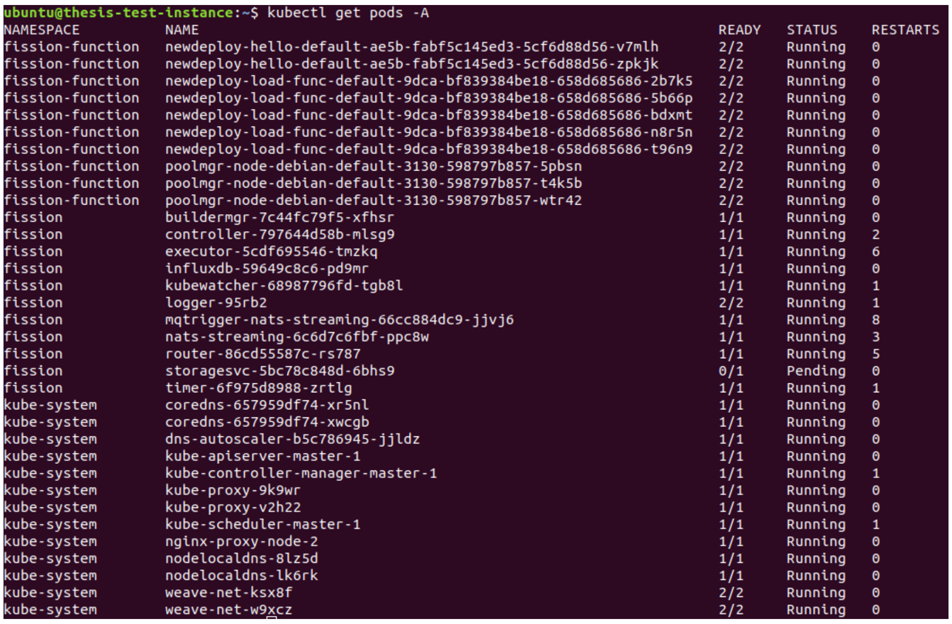
One possible explanation for the increased response time is that by prioritising the Fission function pods, we are inevitably down prioritising other pods in the cluster which influences the over all performance negatively. Prioritisation of other pods than the function pods could influence the response time (but also the execution time) and could be a potential alternation of the experiment to achieve better results. With this in mind, these results are unwanted since the response time is not improved, but are reasonable since the prioritisation of the function pods affects performance of other pods.

In order to improve the results, an optimal set of pods to be prioritised should be found. To do so, more time would be needed for development due to the number of pods present in the cluster, which can be seen in 7.1 where more than 30 pods are present. But with the goal to optimise for response time, a potential candidate for prioritisation could be the router pod which is responsible of forwarding HTTP-requests to the function pods [1]. This argument would be true for experiment 2 and 3 as well.

Experiment 2

The second experiment used a nonpreemptive kernel with real-time group scheduling by configuring limitations in `cgroup`.

The results shows an improvement in execution time when using `SCHED_FIFO` over `SCHED_OTHER` as seen in figure 6.11 showing the boxplots, and in table 6.4 where all KPI's shows a reduction for tests with no load and load periods $2000ms$, $1000ms$,



```
ubuntu@thesis-test-instance:~$ kubectl get pods -A
NAMESPACE          NAME                                                          READY   STATUS    RESTARTS
fission-function   newdeploy-hello-default-ae5b-fabf5c145ed3-5cf6d88d56-v7mlh    2/2     Running   0
fission-function   newdeploy-hello-default-ae5b-fabf5c145ed3-5cf6d88d56-zpkjk    2/2     Running   0
fission-function   newdeploy-load-func-default-9dca-bf839384be18-658d685686-2b7k5  2/2   Running   0
fission-function   newdeploy-load-func-default-9dca-bf839384be18-658d685686-5b66p  2/2   Running   0
fission-function   newdeploy-load-func-default-9dca-bf839384be18-658d685686-bdxmt  2/2   Running   0
fission-function   newdeploy-load-func-default-9dca-bf839384be18-658d685686-n8r5n  2/2   Running   0
fission-function   newdeploy-load-func-default-9dca-bf839384be18-658d685686-t96n9  2/2   Running   0
fission-function   poolmgr-node-debian-default-3130-598797b857-5pbsn             2/2     Running   0
fission-function   poolmgr-node-debian-default-3130-598797b857-t4k5b             2/2     Running   0
fission-function   poolmgr-node-debian-default-3130-598797b857-wtr42             2/2     Running   0
fission            buildermgr-7c44fc79f5-xfhsr                                   1/1     Running   0
fission            controller-797644d58b-mlsg9                                   1/1     Running   2
fission            executor-5cdf695546-tmzkq                                     1/1     Running   6
fission            influxdb-59649c8c6-pd9mr                                      1/1     Running   0
fission            kubewatcher-68987796fd-tgb8l                                  1/1     Running   1
fission            logger-95rb2                                                  2/2     Running   1
fission            mqtrigger-nats-streaming-66cc884dc9-jjvj6                     1/1     Running   8
fission            nats-streaming-6c6d7c6fbf-ppc8w                               1/1     Running   3
fission            router-86cd55587c-rs787                                       1/1     Running   5
fission            storagesvc-5bc78c848d-6bhs9                                   0/1     Pending   0
fission            timer-6f975d8988-zrtlg                                        1/1     Running   1
kube-system        coredns-657959df74-xr5nl                                      1/1     Running   0
kube-system        coredns-657959df74-xwcgb                                      1/1     Running   0
kube-system        dns-autoscaler-b5c786945-jjldz                                1/1     Running   0
kube-system        kube-apiserver-master-1                                       1/1     Running   0
kube-system        kube-controller-manager-master-1                              1/1     Running   1
kube-system        kube-proxy-9k9wr                                              1/1     Running   0
kube-system        kube-proxy-v2h22                                              1/1     Running   0
kube-system        kube-scheduler-master-1                                       1/1     Running   1
kube-system        nginx-proxy-node-2                                            1/1     Running   0
kube-system        nodelocaldns-8lz5d                                            1/1     Running   0
kube-system        nodelocaldns-lk6rk                                            1/1     Running   0
kube-system        weave-net-ksx8f                                               2/2     Running   0
kube-system        weave-net-w9xcz                                               2/2     Running   0
```

Figure 7.1: List of pods in the Kubernetes cluster.

and 500ms in execution time. This can also be seen in the CDF for test with load period 2000ms in figure 6.9 where the CDF outperforms the baseline with a much faster arrival to 1 indicating a lower maximum value and lower value for all percentiles. When comparing the CDF of experiment 2 to the others in figure 6.17, it can be seen to outperform the others since it lies most left in the graph, but the maximum value for experiment 2 and experiment 1 (nonpreemptive cgroup vs nonpreemptive) are equal since they intersect at probability 1. This could imply that experiment 2 is as good as experiment 1, which would be expected since `cgroup` only works as a limitation [40]. The reason why the data points towards a solid improvement when comparing experiment 2 and 1 in tables 6.4 and 6.3 respectively, and in the CDF, could be that this has to do with the influence of the hypervisor [39] on the cluster as mentioned before for experiment 1, since the distribution between the upper and lower stable level of execution time is weighted towards the lower, as seen in figure 6.8. To confirm that this has to do with the hypervisor and not with the implementation of `cgroup`, the test must be ran on a cluster with controller resource allocation, such as in bare metal or with no other VMs disturbing our cluster which is not possible with our current cloud architecture. As previously mentioned, this will be discussed in subsection Bare metal.

On the other hand, the limitation of `cgroup` could actually be advantageous since it limits the FaaS function pods and grants more resources to the rest of the pods. Since we cannot be sure if prioritising the FaaS function pods are the most optimal solution without testing other pods to prioritise, the improvements in this experiment could point towards `cgroup` limitations allowing for better performance without the need to know which exact pods should be prioritised for best performance. This could be true for both execution time and response time.

In table 6.4, it can be seen that the response time in this experiment shows an improvement for no load and load period $2000ms$, and a disimprovement for $1000ms$ and $500ms$ when comparing the 95^{th}-percentile results. The variance is worse for all tests except for load period $2000ms$. Regarding the response time it could be argued that variance is not a very good measurement since the response time can suffer from extremely high outlier which lies much higher than the 95^{th}-percentile, this would mean the variance can increase without the increase of the 95^{th}-percentile and a high variance and low 95^{th} points towards worse results regarding the higher percentiles. For use cases which accepts reliability 95% of the time the 95^{th}-percentile would be a good indicator of a stable cluster but in most commercial use cases much higher uptimes needs to be guaranteed. For example, Amazon Web Services guarantees its customers a 99.99% uptime and partly refunds down to 95%. Below 95% the entire month is refunded [41]. The 95th-percentile should therefore not be used if comparing to commerical clouds, but can be used as a indicator in our cluster since performance is not optimal.

Experiment 3

Another proposed way of achieving a lower execution time for the cluster was to introduce a fully preemptive kernel together with the preemptive scheduler `SCHED_FIFO`.

As presented in the CDF in figure 6.13 an improvement when running `SCHED_FIFO` on the preemptive kernel compared to processes using `SCHED_OTHER` when comparing 80^{th}-99^{th} percentile can be seen. However, when comparing the preemptive kernel with a nonpreemptive kernel in figure 6.17 a worsening of the execution time for the preemptive kernel using `SCHED_FIFO` can be observed. This result contradicts previous research which shows the opposite. In related research [42] execution time delay with CPU bound background workloads was measured and the preemptive kernel outperformed the nonpreemptive kernel. It is important to note that the experiment in research [42] was conducted on a bare metal environment were our environment consists of virtual machines. In [43] it is mentioned that additional configurations to the lower levels, i.e the host of the virtual machines needs to be configured in order to achieve the same degree of reduces latency as in a bare metal environment. This is something that is abstracted away from us in our current cloud and therefore cannot be achieved with our current infrastructure. What is needed to confirm this problem is a similar experiment conducted on a bare metal environment to confirm this issue. It is important to note that when comparing to the previously mentioned ways for improving the latency, this method requires the user to install a new kernel on their virtual machine. This is not something that always is possible for public cloud providers.

An interesting finding from the table 6.5 and boxplot in figure 6.15 is that the execution time is drastically improved for higher loads as seen for when the load periods are $1000ms$ and $500ms$. Intuitively, a higher load on the cluster should correspond to a higher execution time while this shows the opposite. For no load and load period $2000ms$, less than 50% of the CPU is occupied, whereas for $1000ms$ and $500ms$ uses 57% and 87% of the CPU as seen in table 6.1 (these values are from the baseline experiment but can be used as an estimated CPU usage in this experiment). This could imply that the preemptive kernel has greater performance when the load is higher. As the load is increased, the steal is decreased and in figure 6.18 it can be seen that the preemptive kernel in experiment three perform better than the other experiments. One possible explanation to this is that when the preemptive kernel experiences steal, it cannot utilise full preemption. But as the steal decreases, the preemptive kernel may preempt more often resulting in a performance that is closer to bare metal performance, where the preemptive kernel is said to performe better than other kernels [42].

Bare metal

When running a cluster on VMs, the cluster will experience some overhead from interference between different VMs due to the hypervisor [39]. Instead of using a hypervisor, it is possible to run the cluster on bare metal instead of on VMs. By doing this, better performance can be expected but it will limit the clouds flexibility in a real use case since new VMs can no longer be created on the go. Instead, new hardware must be installed which is both costly and time consuming.

A cluster setup with bare metal was done to examine the execution and response time when using real-time scheduling. Unfortunately, it was not possible to run the cluster with multiple nodes with a master/worker hierarchy due to limited resources. In the table 6.6 the bare metal results are presented with all the given KPI's which also can be seen in the boxplot in figure 6.20. This setup was compared to a cloud solution with the same architecture with only one node. The results from this test can be found in table 6.7 with the boxplot representation in figure 6.21.

These results show a few interesting findings that has earlier been discussed. When comparing the two boxplots, it can be seen that the bare metal deployment suffers from an increase in execution time when experiencing a higher load, while the cloud deployment on Xerces shows the opposite (as for all previous experiments deployed on Xerces). This behavior were suspected to be due to the steal metric present for VMs, where the VMs were waiting to be scheduled on the hosts resources by the hypervisor. On bare metal, this behavior is not present which leads us to believe that the steal metric is the best candidate to describe why an increased load correlates with a decrease in execution time on VMs.

On the bare metal setup, it could be expected that when scheduled with `SCHED_OTHER`, the execution time is increased when presented with a greater load since all

SCHED_OTHER processes share the same resources and the measured function may not execute before prioritised nice processes present in the Linux kernel. Although, in the case with processes scheduled with SCHED_FIFO, it would be expected to not experience an increase in execution time when experiencing a higher load since the executed and prioritised ping function should always be allowed to preempt the load function. It could be argued that since all processes share the same resources, a higher load would still affect the execution time to some degree since more processes will be active. Some of these may also be scheduling with real-time schedulers. But since our tests do not utilise 100% CPU usage (the highest load 500ms utilises approximately 80%) it is not clear why the execution time experiences such a high increase since resources are still available, and that other real-time scheduled processes should not preempt the ping function.

When using the real-time scheduler SCHED_FIFO on bare metal, the execution time is lower than when using SCHED_OTHER, more importantly, the variance is 6%, 72%, 27%, and 58% lower for the tests with no load, load period , 2000ms, 1000ms, and 500ms when taking the ratio of the variance of the two schedulers in table 6.6. This shows that SCHED_FIFO is indeed better.

Another interesting finding is the decrease in response time when scheduling with SCHED_FIFO compared to SCHED_OTHER for both the bare metal and Xerces test. In previous experiments, SCHED_FIFO has generally performed worse than SCHED_OTHER in terms of response time. Especially for the test with 500ms. It was previously discussed that the experiments that uses SCHED_FIFO possibly experiences an increase in response time due to the prioritisation of Fission function pods, which would inevitably down prioritise the other pods in Kubernetes and Fission. Some of these pods are responsible of the networking between nodes in the cluster and pods there within. These pods By running these tests on only one node, this networking is no longer needed. Through analysing the presented results, it can most likely be concluded that the increased response time on experiment 1, 2, and 3 are due to networking pods being interrupted. This also mean that it is likely possible to further reduce the response time in the previous experiments that consist of more than one node by prioritising the pods resposible of network between various Kubernetes instances.

Comparisons

The three conducted experiments all show a general reduction in execution time, but analyzing the CDF in figure 6.17 it can be seen that experiment 2 (using cgroup) performs best followed by experiment 1 and then experiment 3. If looking at the response time in the boxplot comparison in figure 6.19 once again experiment 2 has the best performance, followed by experiment 1 and then experiment 3.

When comparing the different methods of implementation to each other, one must take into account the usability of said methods.

For experiment 1, the implementation is straight forward and easy. If implementing it, the user only needs to be concerned with prioritising the containers that are running and need no further configurations of the kernel. This method allows for real-time configurations without doing changes to the existing environment if already using a nonpreemptive kernel, which mostly is the case.

For experiment 2, quite some effort must be done. First of all, the kernel must be modified to enable the flag `CONFIG_RT_GROUP_SCHED` which may not be suitable in all deployment use cases where developers should not be modifying or changing the kernel. Secondly, the user must modify the `cgroup` in order to allow for prioritisation of processes to be set. This can in worst case affect the behavior of the cluster to become unstable if not configured properly, especially if the user modified the `cpu.rt_period_us` since the implementation of modifying this parameter is not yet finished [40]. Also, this requires the user `sudo` access which is not the case for the other experiments. After this is done, the containers must also be entered and modified.

For experiment 3, it is not as time consuming and complicated to set up as for experiment 2, but more complicated than experiment 1 since the kernel must be patched with a real-time patch. This might not be suitable for all use cases since running in a preemptive kernel might

The three experiments all require the user to set prioritisation in the containers manually. One problem with this approach is that if the pod is terminated (which is the case in most use cases) the pods and its prioritisation are reset. A way to work around this problem is to set the command for prioritisation in the deployment file of the pods that will be prioritised and would be a great improvement of the current test setups. This would allow for security benefits as the deployment file of the pods could be configured by admin only to only grant prioritisation for specific pods. To accomplish this, the pod must be given `SYS_NICE` capability as described in the method section, and then passed the command that priorities the pod's process. Since not the pid of the process can differ, the `pgrep` command is used which returns a pid of a process name. This is done by adding the following to the deployment yaml file:

```
spec:
  container:
    securityContext:
      capabilities:
        add:
        - SYS_NICE
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh",
                    "-c",
                    "chrt --fifo --pid $(pgrep node"]
```

Another solution could also be to add this directly to the container image. This solution were not tested and verified in our experiment but can be a possible way to get a real-time ready container image. First the image must be given the `SYS_NICE` capability in the Dockerfile during initialisation, and then the following should be added in the Dockerfile to run the prioritisation:

```
CMD ["/bin/bash", "-c", "chrt --fifo --pid 99 $(pgrep node)"]
```

Although, this would not work on experiment 2 since the `cgroup` would need to be reconfigured each time a pod is terminated. To allow for this to work, more development regarding support for Kubernetes with real-time group scheduling must be conducted. Approaches to succeed with this implementation can be to run a script that configures the `cgroup` outside of Kubernetes, or to manage to integrate custom docker commands when running containers with Kubernetes. With dockers, it is possible to configure the `cgroup` file by running:

```
$ docker run -it \
--cpu-rt-runtime=950000 \
--ulimit rtprio=99 \
--cap-add=sys_nice \
debian:jessie
```

Although, a solution for where this could be added was not found.

Chapter 8

Conclusion

The goal of this research was divided into four points:

- reduce both execution time and variance in execution time for the specified prioritised jobs,
- deduct the possibility of achieving real-time support for specified jobs in a Kubernetes cluster,
- evaluate the optimisation methods effect on the response time for prioritised jobs,
- evaluate the possibility and present a method for deploying a Kubernetes cluster with support for real-time prioritisation.

Starting with the first point, to reduce both execution time and variance in execution time for our prioritized jobs. We saw that experiment 1 (running on the non-preemptive kernel), experiment 2 (running on the non preemptive kernel with cgroup limitations) and experiment 3 (running on the preemptive kernel) all saw a clear improvement for the execution time and variance when comparing to the baseline measurements. Since a real-time system needs to be deterministic with respect to a given tolerance, the upper percentiles for the execution time directly indicates whether a system is viable for a deployment model or not. For our measurements we saw that the introduction of `SCHED_FIFO` had a big impact in lowering this metric. Hence the first goal of this thesis has been achieved.

The second goal, to deducting the possibility of achieving real-time support for specified jobs in a Kubernetes cluster. Despite the fact that we have achieved a lower execution time, we have not established an environment that supports real-time deployments in VMs, this since the response time is still beyond values that will be acceptable in a real-time system. However, we have proved that the execution of processes will not be a limiting factor for achieving real-time support in a Kubernetes cluster. Through our findings from the bare metal experiment, we suspect that the increase in response time in VMs is directly correlating to the inevitably down prioritisation of networking pods in the cluster. We suggest the method of prioritising a different set of pods, which includes various networking pods, to reduce the response time and its variance in order to decrease response time in a cluster running on multiple VMs.

The third goal was to evaluate the optimisation methods effect on the response time for prioritised jobs. Our results show a slight degradation in response time for the tests that ran in a virtual machine environment. However when increasing the weight of the tasks for the measured services we saw an improvement for the response time. These results points towards the conclusion that applying a real-time scheduler to prioritised tasks might interfere and cause a degradation in performance for other crucial services. This could still be a viable option depending on the weight of the prioritised tasks deployed to the cluster.

Lastly a method for automating the deployment of application with support for real-time prioritisation has been proposed. Through specifications in the deployment file in a Kubernetes cluster, capabilities allowing for the container to prioritise processes has been presented for both the non-preemptive kernel and preemptive kernel using `SCHED_FIFO`. This can in combination with container configurations allows a user to automate the support for real-time scheduling policies. However, for the `cgroup` rt-limitation no sensible way to automate the process has been achieved integrated in Kubernetes.

References

[1] "fission documentation v1.12.0," visited 2021-05-24. [Online]. Available: https://docs.fission.io/docs/

[2] "Kubernetes documentation," visited 2021-02-24. [Online]. Available: https://kubernetes.io/docs

[3] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[4] T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, "Virtual network functions as real-time containers in private clouds," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD).* IEEE, 2018, pp. 916–919.

[5] I. Pelle, J. Czentye, J. Dóka, and B. Sonkoly, "Towards latency sensitive cloud native applications: A performance study on aws," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD).* IEEE, 2019, pp. 272–280.

[6] A. I. Bucur and D. H. Epema, "Local versus global schedulers with processor co-allocation in multicluster systems," in *Workshop on Job Scheduling Strategies for Parallel Processing.* Springer, 2002, pp. 184–204.

[7] M. C. Ogbuachi, C. Gore, A. Reale, P. Suskovics, and B. Kovács, "Context-aware k8s scheduler for real time distributed 5g edge computing applications," in *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM).* IEEE, 2019, pp. 1–6.

[8] Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao, "Progress-based container scheduling for short-lived applications in a kubernetes cluster," in *2019 IEEE International Conference on Big Data (Big Data).* IEEE, 2019, pp. 278–287.

[9] D. Cho, J. Taheri, A. Y. Zomaya, and P. Bouvry, "Real-time virtual network function (vnf) migration toward low network latency in cloud environments,"

in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 798–801.

[10] IBM, "Cloud computing history," Jan 2017, visited 2021-04-20. [Online]. Available: https://www.ibm.com/cloud/blog/cloud-computing-history

[11] C. S. Pabla, "Completely fair scheduler," Aug 2009, visited 2021-03-25. [Online]. Available: https://www.linuxjournal.com/node/10267

[12] "What is a virtual machine (vm)?" visited 2021-03-22. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine

[13] "What is a hypervisor?" visited 2021-03-15. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor

[14] B. Bermejo and C. Juiz, "Virtual machine consolidation: a systematic review of its overhead influencing factors," *The Journal of Supercomputing*, vol. 76, no. 1, pp. 324–361, 2020.

[15] "What is a container?" visited 2021-03-25. [Online]. Available: https://www.docker.com/resources/what-container

[16] K. Kumar and M. Kurhekar, "Economically efficient virtualization over cloud using docker containers," in *2016 IEEE international conference on cloud computing in emerging markets (CCEM)*. IEEE, 2016, pp. 95–100.

[17] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing," 2011.

[18] T. Diaby and B. B. Rad, "Cloud computing: a review of the concepts and deployment models," *International Journal of Information Technology and Computer Science*, vol. 9, no. 6, pp. 50–58, 2017.

[19] "The history of pets vs cattle and how to use the analogy properly," visited 2021-05-24. [Online]. Available: http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/

[20] S. Tilkov, "The modern cloud-based platform," *IEEE Software*, vol. 32, no. 2, pp. 116–116, 2015.

[21] B. Sosinsky, *Cloud computing bible.* John Wiley & Sons, 2010, vol. 762, pp. 1–22.

[22] M. J. Sadeeq and S. R. M. Zeebaree, "Semantic Search Engine Optimisation (SSEO) for Dynamic Websites: A Review," *International Journal of Science and Business*, vol. 5, no. 3, pp. 148–158, 2021. [Online]. Available: https://ideas.repec.org/a/aif/journl/v5y2021i3p148-158.html

[23] N. R. Tadapaneni, "Different types of cloud service models," 2017.

[24] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 442–450.

[25] R. A. P. Rajan, “A review on serverless architectures-function as a service (faas) in cloud computing,” *TELKOMNIKA*, vol. 18, no. 1, pp. 530–537, 2020.

[26] S. K. Mohanty, G. Premsankar, M. Di Francesco *et al.*, “An evaluation of open source serverless computing frameworks.” in *CloudCom*, 2018, pp. 115–120.

[27] “What is the real-time cloud and how do we get there?” visited 2021-03-25. [Online]. Available: https://www.ericsson.com/en/blog/2020/11/what-is-real-time-cloud

[28] J. Pan and J. McElhannon, “Future edge cloud and edge computing for internet of things applications,” *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 439–449, 2018.

[29] R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. S. Netto, A. N. Toosi, M. A. Rodriguez, I. M. Llorente, S. D. C. D. Vimercati, P. Samarati, D. Milojicic, C. Varela, R. Bahsoon, M. D. D. Assuncao, O. Rana, W. Zhou, H. Jin, W. Gentzsch, A. Y. Zomaya, and H. Shen, “A manifesto for future generation cloud computing: Research directions for the next decade,” *ACM Comput. Surv.*, vol. 51, no. 5, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3241737

[30] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, “Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds,” in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1270–1278.

[31] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[32] G. Sayfan, *Mastering Kubernetes: large scale container deployment and management.* Packt, 2017.

[33] The Linux Kernel Archives, visited 2021-05-24. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt

[34] RedHat, visited 2021-03-28. [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel

[35] “Linux manual page,” Mar 2021, visited 2021-03-21. [Online]. Available: https://man7.org/linux/man-pages/man7/sched.7.html

[36] The kernel development community, “Real-time group scheduling,” visited 2021-03-22. [Online]. Available: https://www.kernel.org/doc/html/latest/scheduler/sched-rt-group.html

[37] Linux manual, visited 2021-03-28. [Online]. Available: https://man7.org/linux/man-pages/man1/chrt.1.html

[38] P. Pawar, S. Dhotre, and S. Patil, “Cfs for addressing cpu resources in multicore processors with aa tree,” *International Journal of Computer Science and Information Technologies*, vol. 5, no. 1, pp. 913–917, 2014.

[39] Z. Yang, H. Fang, Y. Wu, C. Li, B. Zhao, and H. H. Huang, “Understanding the effects of hypervisor i/o scheduling for virtual machine performance interference,” in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, 2012, pp. 34–41.

[40] “Real time group scheduling,” visited 2021-03-28. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt

[41] Amazon Web Services, visited 2021-04-15. [Online]. Available: https://aws.amazon.com/compute/sla/

[42] F. Brandenburg, “A comparison of scheduling latency in linux, preempt rt, and litmusrt,” *OSPERT 2013*, p. 20, 2013.

[43] J. Danisevskis, M. Peter, and J. Nordholz, “Minimizing event-handling latencies in secure virtual machines,” *arXiv preprint arXiv:1806.01147*, 2018.

Printed by Tryckeriet i E-huset, Lund 2021

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2021-829
http://www.eit.lth.se