

Master of Science in Telecommunication Systems  
June 2019



# Performance Optimization of a Service in Virtual and Non-Virtual Environment

Monica Tamanampudi  
Mohith Kumar Reddy Sannareddy

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Telecommunication Systems. The thesis is equivalent to 20 weeks of full time studies.

The Authors in this research paper grants to Blekinge Institute of technology non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Authors warrants that the work does not contain any text, pictures, references and materials that violate the copyright laws.

**Contact Information:**

Author(s):

Monica Tamanampudi

E-mail: mota17@student.bth.se

Mohith Kumar Reddy Sannareddy

E-mail: mosb17@student.bth.se

University advisor:

Dr. Patrik Arlos

Department of Communication Systems

---

## Abstract

In recent times Cloud Computing has become an accessible technology which makes it possible to provide online services to end user by the network of remote servers. With the increase in remote servers and resources allocated to these remote servers leads to performance degradation of service.

In such a case, the environment on which service is made run plays a significant role in order to provide better performance and adds up to Quality of Service. This paper focuses on Bare metal and Linux container environments as request response time is one of the performance metrics to determine the QOS. To improve request response time platforms are customized using real-time kernel and compiler optimization flags to optimize the performance of a service. UDP packets are served to the service made run in these customized environments. From the experiments performed, it concludes that Bare metal using real-time kernel and level 3 Compiler optimization flag gives better performance of a service.

**Keywords:** Cloud Computing, Computing resource, Linux Containers.

## Acknowledgements

Firstly, we would like to express our gratitude to our professor Dr. Patrik Arlos, who helped us through his immense support, guidance, and suggestions throughout our thesis work.

We would also like to thank our fellow mates Saaish Bhonagiri, Chaitanya Ivvala, Prathisrihas Reddy Konduru, and Vamsi Krishna Bandari Swamy Devender for their continuous support and suggestions which helped us to complete this research work.

Last but not least we would like to thank our family and friends who stood as a pillar of support throughout our career.

---

# Nomenclature

Bm	Bare metal
DPMI	Distributive Passive Measurement Infrastructure
DUT	Device under test
ECDF	Empirical Cumulative Distribution Function
GCC	GNU Compiler Collection
LXC	Linux Containers
MAC	Measurement Area Controller
MP	Measurement Point
PKtlen	Packet Length
Pkts	Packets
Preempt Full	Preempted Full Kernel
Preempt LL	Preempted Low Latency Kernel
QoS	Quality of Service
RT kernel	Real-Time kernel
UDP	User Datagram Protocol
VMM	Virtual Machine Monitor
Wt	wait time

---

## List of Figures

2.1	Characteristics of a Cloud . . . . .	5
2.2	Architectural comparison of different virtualization techniques . .	7
2.3	Linux Containers . . . . .	9
5.1	Client-Server Architecture with response and requests . . . . .	19
5.2	Client-Server Architecture . . . . .	21
5.3	Empherical cummulative distrubution function . . . . .	24
6.1	Latency w.r.t wait time 0 and 1,000,000 packets . . . . .	26
6.2	Latency w.r.t wait time 0 and 1,000,000 packets . . . . .	27
6.3	Latency w.r.t wait time 0 and 1,000,000 packets . . . . .	28
6.4	Latency w.r.t wait time 0 and 1,000,000 packets . . . . .	29
6.5	Latency w.r.t wait time 0 and 1,000,000 packets . . . . .	31
6.6	Latency w.r.t wait time 0 and 1,000,000 packets . . . . .	32
7.1	Comparision of 50% average values . . . . .	35

---

## List of Tables

3.1	Comparison of different factors under different Kernels . . . . .	14
5.1	Traffic Specifications . . . . .	22
5.2	System components specifications in Bare metal Environment(Ubuntu 18.04) . . . . .	22
5.3	System components specifications in Bare metal Environment(Ubuntu 16.04) . . . . .	22
5.4	System components specifications in LXC Environment . . . . .	23
6.1	stat-1 . . . . .	26
6.2	stat-2 . . . . .	27
6.3	stat-3 . . . . .	29
6.4	stat-4 . . . . .	30
6.5	stat-5 . . . . .	31
6.6	stat-6 . . . . .	32

---

# Contents

<b>Abstract</b>	<b>i</b>
<b>Nomenclature</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Scope of the thesis . . . . .	2
1.3 Aim and Objective . . . . .	2
1.4 Research Questions . . . . .	2
1.5 Research Method . . . . .	2
1.6 Thesis Outline . . . . .	3
<b>2 Background</b>	<b>4</b>
2.1 Cloud Computing . . . . .	4
2.2 Virtualization . . . . .	6
2.2.1 Server Virtualization Techniques . . . . .	6
2.3 Bare metal . . . . .	7
2.4 Containers . . . . .	7
2.4.1 Linux Containers . . . . .	8
2.5 Service Architecture . . . . .	9
2.5.1 Monolithic Application . . . . .	9
2.5.2 Microservice architecture . . . . .	9
2.6 Kernels . . . . .	10
2.7 Compiler . . . . .	10
2.7.1 GCC . . . . .	11
2.7.2 ICC . . . . .	11
<b>3 List of Choices</b>	<b>12</b>
3.1 Choosing Linux or Docker Containers . . . . .	12
3.2 Choice of Operating system . . . . .	12
3.3 Choice of Kernels . . . . .	12
3.4 Choosing Compiler Optimization flag . . . . .	14
<b>4 Related Work</b>	<b>16</b>

<b>5</b>	<b>Methodology</b>	<b>18</b>
5.1	Literature Study . . . . .	18
5.2	Modeling the service architecture . . . . .	19
5.3	Implementation . . . . .	19
5.4	Experimental setup and Data Collection . . . . .	20
5.4.1	Test Beds . . . . .	21
5.5	Analysis . . . . .	23
<b>6</b>	<b>Result and Analysis</b>	<b>25</b>
6.1	Optimized Vs Non-Optimized . . . . .	25
6.2	Optimized Ubuntu 16.04 Vs Ubuntu 18.04 . . . . .	27
6.3	Bare metal Vs Linux Container using different kernels . . . . .	28
6.3.1	Performance comparison for optimized service in two different generic kernel environments . . . . .	28
6.3.2	Performance Comparison of Optimized service in Bm and Bm Low Latency Kernel . . . . .	29
6.3.3	Performance Comparison between Optimized services deployed in LXC using generic and Low Latency Kernels . . . . .	31
6.3.4	Performance Comparison between BmLL VS LXCLL . . . . .	32
<b>7</b>	<b>Conclusion and future work</b>	<b>34</b>
7.1	Research Question and Answers . . . . .	35
7.2	Future work . . . . .	35
	<b>References</b>	<b>37</b>
<b>A</b>	<b>To plot empirical cumulative distribution function</b>	<b>40</b>

Over the years infrastructural changes were made quite often, which led to the difficulty in making changes to the software and patching as long as the hardware changes took place. With all these difficulties faced, cloud computing came into existence and has grown drastically[1][2]. Also, with the numerous advantages provided by the cloud, there has been a trend from monolithic to microservice architecture. With this emergence, services became easier to build and maintain when they are broken down into smaller composable ones, which have advantages like developer independency, scalability, isolation, and resilience. Cloud Computing covers a wide range of applications from online services for the end user due to this servers were used in abundance by the customers. With increase in demand even a small network delay leads to performance degradation of service.Hence the request-response time is the performance metric that is under study to improve the Quality of Service.[3].

The request-response varies from one environment to another environment for a service made run and can be measured at all layers.Since the packets traverse through all the components in a network,it is important to measure the accurate request-response times for the transmission of packets from a client to a server and vice-versa. Hence request-response time is chosen as the performance metric which is measured at the data link layer in our study. UDP traffic is chosen over TCP since there are parameters like inter-frame gap and packet length which is used to stabilize and analyze the system behaviour when a higher number of packets are served. It is also important to consider a kernel and a compiler where the kernel manages the operations of a computer and its hardware whereas a compiler plays a crucial role in the execution of an application which in turn has a effect on the performance of a service in different environments. Different environments considered in this study are Bare metal, and Linux containers were customized using compiler optimization and Kernel choices, which plays a vital role in optimizing the performance of a service. The experiments are performed using different test cases with a combination of kernels and compiler optimizations. The Empirical cumulative distribution is made use for plotting the graphs and analyzing the performance of a service in each test case.

## 1.1 Motivation

The performance of an application running in cloud depends on the resources allocated to an application. With the increase in demand for computing there is an increase in network delays which leads to a performance degradation. Hence, there is a severe need to improve the performance of a service to deliver the best QOS to the customers. Hence it would be exciting to study and analyze the performance of a service when the service is made run in a virtual and non-virtual environment.

## 1.2 Scope of the thesis

The main focus of this thesis work is to run a service on Bare metal and Linux containers by switching to a real-time kernel and by choosing different optimization flags. The impact on the performance of a service is obtained from the request response time. This can be tested in various environments such as Bare metal, Containers, Hypervisors, and Virtual Machines. As per the choices made during the literature study, this thesis work is limited to Bare metal and Containers.

## 1.3 Aim and Objective

Aim of this study is to analyze the performance of service in the virtual and non-virtual environment to see how the performance is affected in a standard configured operating system and a customized operating system in Bare metal, Linux containers.

The objective of this study is to improve the request response time of service by running it on a customized environment to provide the best quality of service.

## 1.4 Research Questions

This research focuses on the following research questions:

- If we use a real-time kernel for the platforms such as Bare metal and Linux container, do we improve the performance of a service?
- Can we achieve a gain in the performance of a service by doing Optimization on the service?

## 1.5 Research Method

There are many ways in order to conduct a research on a particular topic of interest which involves the following with a step by step approach firstly a background

study is made to gain a deep knowledge on the topics involved in doing a research work and later the experiments are conducted thereby the analysis is carried out on the experimented data. Some of those are by doing simulation, emulation. Simulation is a process of adopting an abstracted logical component to depict the actual functionality of the system. Emulation is a process of imitating a hardware/Software program or a platform on another program or platform. In this study, the emulation process is performed by setting up a test bed with a virtual and non-virtual environment and performing experiments by running a service in these environments. The results are analyzed, and the impact on the performance of a service is analyzed by using the request response time. The stages mentioned below are followed to carrying out the thesis work.

- **Background and Literature study:** During this phase, multiple research papers are studied which related to environments, kernels, and Compiler Optimizations.
- **Modelling of service architecture:** By considering the requirements in order to conduct the experiments, a client/server architecture is implemented.
- **Implementation:** In this stage, the experimental setup is developed, and the tests are performed by running the service on Bare metal and Linux containers.
- **Evaluation:** The experiments done are evaluated, and the impact on the performance of a service is studied by obtaining the request response times.
- **Results and Analysis:** From the evaluation made the results are analyzed, and the conclusions are drawn from the results for the impact on the performance of a service by calculating the empirical cumulative distribution function and the graphs are plotted using the ECDF values.(For more details chapter 3 )

## 1.6 Thesis Outline

This report is organized as follows. Chapter 2 provides an overview and background of the research work. Chapter 3 provides the list of choices opted according to the literature study. Chapter 4 provides an insight into the thesis works done related to this area of research. Chapter 5 deals with the methodology followed and also described the experimental testbed setup and implementation. Chapter 6 contains the results and analysis, and Chapter 7 gives the conclusions and future work.

### 2.1 Cloud Computing

The NIST organization defines Cloud Computing as a model for providing on-demand, convenient, network access to different data centers residing at the cloud provider. It is like a shared pool of configurable computing resources like networks, databases, servers, and storage that can be provided on the spot and deallocated easily[4]. There are three types of cloud deployment models those are Public Cloud, Private Cloud, and Hybrid Cloud.

- **Public Cloud-** Whole computing infrastructure is set on the locations of a cloud computing company that offers the cloud service.
- **Private Cloud-** Hosting all your computing infrastructure yourself and is not shared, The security and control level is highest while using a private network.
- **Hybrid Cloud-** Using both private and public clouds, depending on their purpose. You host your most important applications on your servers to keep them more secure and secondary applications elsewhere.

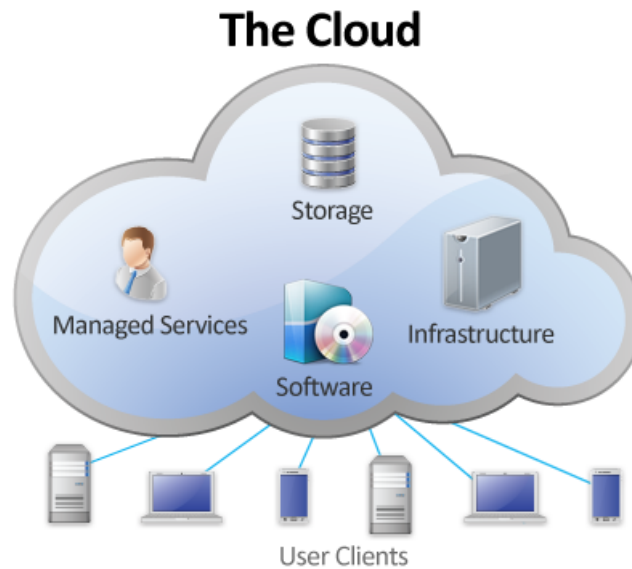


Figure 2.1: Characteristics of a Cloud

Cloud computing services fall into four categories: infrastructure as a service (IaaS), platform as a service (PaaS), software as a service (SaaS) and FaaS (functions as a service). These are sometimes called the cloud computing stack because they build on top of one another.

IaaS is the most basic category of cloud computing service that allows to rent IT infrastructure (servers or VM's) from a cloud provider on a pay-as-you-go basis. Platform-as-a-service (PaaS) refers to the supply of an on-demand environment for developing, testing, delivering, and managing software applications. It is designed to quickly create web or mobile apps, without worrying about setting up or managing the underlying infrastructure of servers, storage, network, and databases needed for development. Software-as-a-service (SaaS) is a method for delivering software applications over the Internet as per the demand and on a subscription basis. SaaS helps you host and manage the software application and underlying infrastructure and handle any maintenance (software upgrades and security patching). FaaS adds another layer of abstraction to PaaS so that developers are completely insulated from everything in the stack below their code. Instead of handling the hassles of virtual servers, containers, and application runtimes, they upload narrowly functional blocks of code and set them to be triggered by a specific event. FaaS applications consume no IaaS resources until an event occurs, reducing pay-per-use fees[5].

In order to improve the system performance, different methods come into the picture, which in turn can affect the system performance[6]. Some of the methods that can be considered in the process of improving system performance are as follows:

## 2.2 Virtualization

Virtualization is the abstraction of computer resources. It abstracts the programming from the existing hardware infrastructure. Accordingly, it wipes out the issue of utilizing a particular software stack to a specific server; thus by empowering more adaptable control of both hardware and software resources. Virtualization allows several virtual servers to be centralized into a single physical machine. There are different virtual technologies which follow different virtualization methods. All these virtualization standards are facilitated using a hypervisor that runs on the host system. Different cloud providers use different standards and techniques in adopting the hypervisor, thereby enabling the resource allocation to the users. There are various types of virtual technologies such as Xen, VMware, Virtual box, KVM[7].

### 2.2.1 Server Virtualization Techniques

There are different types of Virtualization Techniques based on the abstraction of resources. Mentioned below are the different virtualization Techniques:

- Operating System Virtualization
- Full Virtualization
- Para-Virtualization

#### **Operating System Virtualization:**

OS virtualization is also called container-based virtualization. The isolation is provided to guests from the underlying hosts, but hardware resources are not virtualized in this type of virtualization. This type of virtualization technologies patches the kernel of host OS, thereby providing features like process isolation and resource management. Those features come handy if there is a need for deployment of dozens or hundreds of virtual machines in the environments.

#### **Full Virtualization:**

Full virtualization sometimes called hardware emulation. It simulates the whole hardware. In this Virtualization Guest, OS is unmodified and believes it is running on the same hardware as the host OS.

#### **Para-Virtualization:**

Paravirtualization also uses a hypervisor as like as full virtualization. It is the method wherein a modified guest OS can communicate directly to the hypervisor. This direct communication reduces translation time, and overhead as the symbiotic relationship of the two is more efficient. However, unlike full virtualization,

Paravirtualization requires changes to the virtualized operating system. It allows the direct interaction of guest OS with host systems hardware, thereby benefiting the performance of guest OS[8].

## Architectural Comparison

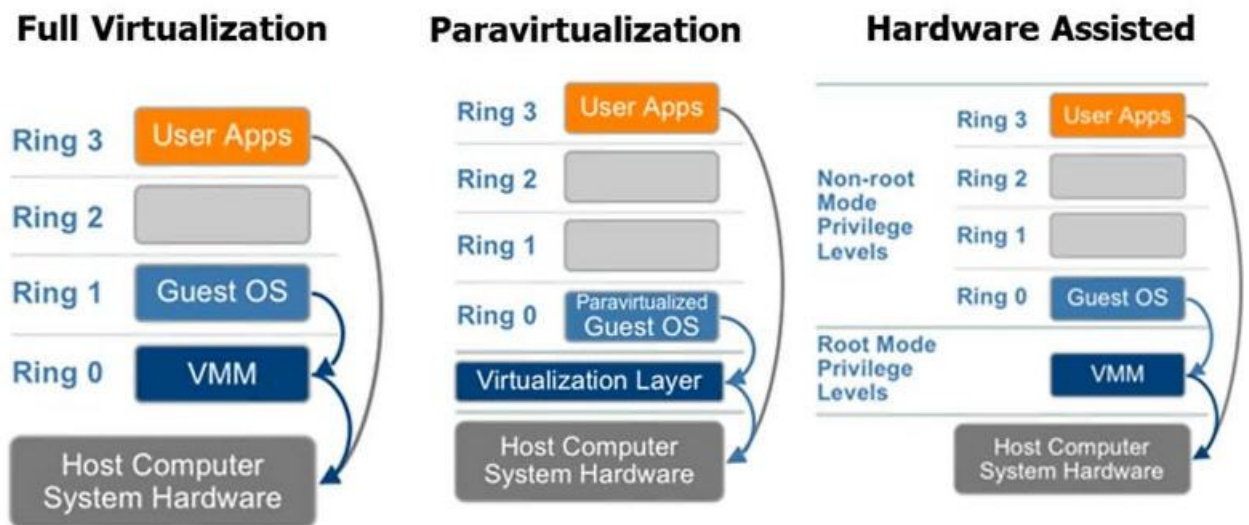


Figure 2.2: Architectural comparison of different virtualization techniques

## 2.3 Bare metal

Bare metal is a computer which consists of hardware components such as hard disks, processors, motherboard etc excluding software. User can access the firmware in order to install any operating system of their requirement. It is a type of virtualization environment in which the virtualization hypervisor is directly installed and executed from the hardware. It doesn't require the need of host operating system. It directly communicates with the underlying hardware to run virtual machine specific processes[9].

## 2.4 Containers

Containers are packages that rely on virtual isolation to deploy and run applications that access a shared operating system (OS) kernel without the need for

virtual machines Containers hold the components necessary to run desired software. These components include files, environment variables, dependencies, and libraries. The host OS constrains the container's access to physical resources, such as CPU, storage, and memory, so a single container cannot consume all of a host's physical resources. There are two types of containers: A system container which runs an operating system inside it and an application container which runs an application in it[10].

### 2.4.1 Linux Containers

Linux containers (LXC) represent a different method of OS-level virtualization. It allows multiple isolated Linux systems (containers) to be run on a single host operating system. The host kernel provides process isolation and performs resource management. It means that even though all the containers are running under the same kernel, each container is a virtual environment that has its file system, processes, memory, devices, etc. LXC relies on the Linux kernel C groups functionality. It also depend on other kinds of namespace isolation functionality, which were developed and integrated into the mainline Linux kernel. It is using CGroups to manage resources that include core count, memory limit, disk I/O limit, etc. The container within LXC shares the kernel with the OS, so its file systems and running processes are visible and manageable from the Host OS[11].

#### Namespaces

The kernel provides process isolation by creating separate namespaces for containers. Namespaces enable creating an abstraction of a particular global system resource and make it look as a separated instance to processes within a namespace. Consequently, several containers can use the same resource simultaneously without creating a conflict.

#### Control Groups (C groups)

The kernel uses C groups to group processes for system resource management. Cgroups allocate CPU time, system memory, network bandwidth, or combinations of these among user-defined groups of tasks.

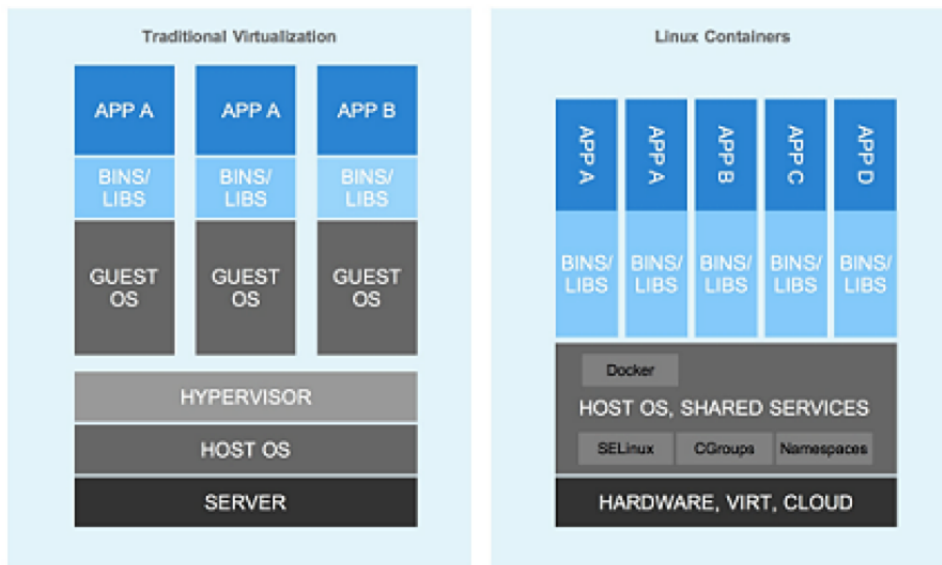


Figure 2.3: Linux Containers

## 2.5 Service Architecture

In order to deploy a service, it is vital to adapt to architecture, and hence service architecture plays an essential role in an endpoint communication, and service architecture is a collection of services which involves data transfer or communication between services due to the Challenges faced by the traditional monolithic application development strategies led to Micro service architectural development[12].

### 2.5.1 Monolithic Application

Monolithic architectures are made running on a single application layer that tends to bundle all the functionalities needed by the architecture. A monolithic application is made as a single unit. The services in such applications are often integrated with the interfaces. Enterprise applications are often built with client-side user interfaces, a database, and a server-side application.

### 2.5.2 Microservice architecture

Microservice architecture is the ones that copy SOAs with minimal, independent Unix-inspired services. It is a small application that runs its logic; they have different features and functions. Containers are better suited for microservices than virtual machines are because micro-services can start up and shut down more quickly. Since microservices are not dependent on each other, they can be made run easily, and any modifications can be done most simply[13].

## 2.6 Kernels

The kernel is the main component of an operating system. It is known as the heart of an operating system since it extends an interface between user applications and the hardware. Its main functions are memory management, process management, task management, and disk management.

When selecting an Operating system, the determining factors should be kept in mind also the kernel choices impact on the performance. Thus, kernel choices can enhance the efficiency of Network Operations and Functions.

According to researches, by taking generic kernel as the baseline and comparing this with the low latency and kernel patched to be fully preemptive or also called real-time kernel gives some impressive results depending on the workloads. For the real-time characteristics and the real-world workloads, it is shown that the low latency kernel performs the best and offers balanced, has high performance for virtualization infrastructure[14].

A massive number of workloads run on Ubuntu, which is most production tested nowadays mainly in the cloud environments where the scaling has been more popular and reliable. It is most important to choose the right kernel for a right service to provide excellent support, and the Ubuntu supports the generic Linux kernel as well as low latency Ubuntu kernel. Also, in order to pick the right kernel efficiently, some key metrics must be considered. Those are:

- **Response times:** For the kernel to respond immediately to the service requests. Preemption points are added to the kernel's subroutines. These allow lower priority system calls to be preempted to higher priority real-time task reducing latency for real-time task.
- **The balance between tasks:** There should be a balance between the present task and tasks that must be taken priority. If any of it goes the other way, then the kernel will respond slowly to external requests and will become inactive to non-priority requests.

## 2.7 Compiler

A compiler is a program that converts a high-level programming language into a low-level language so that can be understood and executed by the computer. The main goal of a compiler is to generate an executable program for a given architecture from the source code. For every high-level command in the source code, there are many machine instructions, and the compiler selects these possibilities and chooses among them. The two modern C/C++ compiler nowadays are GCC and ICC. Both these compilers support general levels of optimization[15].

### **2.7.1 GCC**

The GNU Compiler Collection is a compiler system produced by the GNU Project supporting various programming languages. GCC is a vital module of the GNU toolchain, and the standard compiler for most projects related to GNU and Linux, the most notable is the Linux kernel[16].

### **2.7.2 ICC**

The Intel C++ Compiler (ICC) is a collection of Intel C and C++ compilers that are available for Linux, Microsoft Windows, and Mac OS X. It consists of a commercial tool, developed at Intel Corporation; licenses for the academy are provided without costs. The ICC supports the compilation sibling x86 based architectures[17].

### 3.1 Choosing Linux or Docker Containers

Fundamentally both Docker and Linux containers are similar. They both are userspace and lightweight virtualization platforms which implement cgroups and namespaces for resource isolation. However, they have distinct differences between them. Docker restricts container to run as a single process. If an application has many concurrent processes docker will run an equal number of containers each with a separate process which is not the same in Linux containers which runs a container with init process and can run multiple processes in the same container. Docker supports persistent storage for example if a docker image is created it will consist of read-only layers and this state will not change but during runtime, if the process of the container makes any changes to its internal state the current state of the image will still be the same until the container is deleted[11].

By considering these points though docker containers have additional advantages. a choice made to pick up Linux containers instead of Docker as it beats over Docker in both process management and State Management[18].

### 3.2 Choice of Operating system

The primary difference between Linux and many other popular contemporary operating systems is that the Linux kernel and other components are free and open-source software. Linux is not the only such operating system, although it is by far the most widely used. Hence in our study we have made use of a Ubuntu 18.04 LTS and Ubuntu 16.04 LTS.

### 3.3 Choice of Kernels

Different real-time kernels have been considered in order to decide which kernel to opt. Here the kernel comparisons are explained in detail. Under different workloads, the following kernels are considered:

- Generic Kernel

- Low Latency Kernel
- Kernel patched to be fully preemptive
- Kernel patched to be preemptive with low latency
  
- **Generic kernel:** This is the stock kernel that is provided by default in Ubuntu.
- **Preempt kernel:** This kernel is based on the -generic kernel source tree but is built with different configurations (settings) to reduce latency. Also known as a soft real-time kernel.
- **rt kernel:** is based on the Ubuntu kernel source tree with Ingo Molnar maintained `PREEMPT_RT` patch applied to it. Also known as a hard real time kernel.
- **low latency kernel:** very similar to the -preempt kernel and based on the -generic kernel source tree, but uses a more aggressive configuration to reduce latency further. Also known as a soft real-time kernel.

Preempt Full Kernels Consume more CPU which in turn results in a reduction of throughput and user process than generic kernels and are also costly to maintain whereas low latency kernels are favorable for both throughput and user process which reduces the overall overhead of the system providing a near real-time performance. It does not require any kernel patch.

Low Latency Kernels is an excellent option for servicing real-time traffic and low latency requirements. Hence Low latency Kernel is used in our research work[14].

Metrics	Generic	Low Latency	Preempt full	Preempt LL
CPU Consumption	More favorable for throughput for x86-64	More favorable for throughput and user space CPU access	More CPU usage resulting reduction in throughput on user process compared to generic	More CPU usage resulting reduction in throughput on user process compared to generic
Performance	Offers near real-time performance	Offers near real-time with a reduction in overall system overhead	Provides real-time performance	Provides real-time performance
Cost	Less cost than preempt kernels	Requires less cost than preempt kernels	Costly to maintain	Costly to maintain
Power consumption	Less CPU utilization	Less CPU Utilization	More than generic Adds 2-3 microseconds extra to latencies than generic	More CPU Utilization
Latencies	Better than preempt kernels	Improved latencies than preempt	Preempt may not win over low latency under certain loads	Preempt may not win over low latency under certain loads

Table 3.1: Comparison of different factors under different Kernels

### 3.4 Choosing Compiler Optimization flag

The optimization of code usually involves applying different rules and algorithms to the user code in order to reach specific goals of making it faster, efficient, and smaller. Right compiler optimizations to a program have a significant impact on the performance of a program. The effect on the compiler optimization is decided by the environment, architecture, and application, which is defined by the setting of compiler optimizations and compiler heuristics[19].

To improve the performance at execution time, it is necessary to utilize the compiler optimizations. The two popular C Compilers used in recent times are GCC and ICC. In this study, GCC Compiler is chosen where GCC compiler is a GNU project that supports a wide variety of programming languages also it is available for a large number of platforms. As discussed, an optimized code produces faster execution time, but different optimization codes produce a different gain in the performance depending on the source code, and hence there are

different flags for a different level of optimizations those are explained in detail below. There are three levels of optimizations provided by the flags[20]: In our study we are focusing on improving the performance of a service and hence we are optimizing the application that is made run on two different environments.

- **O**: This is a default flag which is used when the optimization is not required. It Turns off Optimization.
- **O1**: This flag decreases the code size and increases the performance speed.
- **O2**: As long as the code size is not increased, it optimizes for speed. Loop Unrolling and function inlining are not performed.
- **O3**: Optimizes for speed while generating a larger code size. Includes `-O2` optimizations.
- **Loop Unrolling**: It is a loop transformation technique where it optimizes a programs execution speed.
- **Function inlining**: Function inlining is C++ enhancement feature to increase the execution time of the program. Functions can be instructed to the compiler to make them inline so that compiler can replace those function definitions wherever those are being called.

From the above explanation, level 3 optimization flag is considered which is the highest level of optimization, which reduces the execution speed of a program and gives better results.

This section describes the literature study that is done before and study of papers related to the area of work. The existence of cloud computing and its merits led to evolvement in architectural styles from a monolithic architecture to microservice architecture in these paper author has analyzed and tested the microservice architectural patterns which where deployed in cloud as a set of small services which has the functions such as sclaibilt, operatability, and has been easy to upgrade which in turn has a reduction in complexity. The author has tested an application which has been deployed in cloud using both monolithic and microservice architectures and the challenges in implementing these architectures were described in brief[13]. The author has discussed the importance of cloud computing and its characteristics such as performance, scalability, availability and security of cloud computing. The security issues are discussed and how to increase the security of cloud is discussed such as using different encryption algorithms. The author has discussed how to improve the characteristics of a cloud. [6]. The author has described the changing trend towards cloud computing and how the performance in affected in different environments under different workloads. Also the analysis and evaluation is done in order to improve the performance in cloud computing environments[1]. In this paper the author has concentrated on the concepts of virtualization and has discussed various virtualization techniques. The pros and cons of virtualization has beene discussed in detail. The security breaches in virtualization has been addressed so that the implementation can be done keeping in mind of the security vulnerabilities [21]. This is a paper regarding the Canonical team that has made a research on various kernels by performing tests using ubuntu 18.04 and how the change in kernel affects the performance . This paper addresses the kernel comparison under different workloads which focuses on minimizing latency, jitter, system throughput, scheduling the overhead and having a balanced operational efficiency. There is comparison of kernel under low, medium and heavy workloads having both x 64 and aarch 64 hardware architectures. Four different kernels were considered in their case where the latency ditributions in idle system, under light load and heavy load were tested by considering both hardware architectures and the four different kernels and the comparisons are made under the three scenarios[14]. In this work the author has evaluated the

performance with the use of optimization flags for both GCC and ICC Compilers using different APIs. Two architectures were considered for the experiments such as Intel Xeon and AMD Opteron. The different optimization levels and the detailed description of what each level performs is described in this paper. The comparison between GCC and ICC compilers are made based on the optimization levels used. This paper concludes that the use of optimization flags improves the performance and also the use of application scheduler has an impact on the performance. [20]. In this paper the author has addressed the difficulties in implementing a genetic algorithm to run and compile each program for several times to find the optimized compiler options. The author had implemented a new tool called Analysis of compiler options via simulated annealing which compares the running time with the optimization sets which is obtained from the tool. [15]. The author has identified the demand and need to scale an application in a cloud environment thereby introducing the concepts of monolithic and microservice architectures and has addressed different challenges in deploying microservices. The comparison is made between monolithic and microservice architecture by considering the parameters such as response time, throughput etc. [12]. The author has discussed both the container based and hypervisor based virtualization and discuss the role of container based virtualization in cloud technologies. The benefits of both the container based (LXC) and docker containers are mentioned in detail. The Linux containers and the function of c groups are discussed. The author has discussed how container technologies emerged and how it fits into the cloud and the challenges faced with the container technologies. [10]. This paper addresses the concept of a client server model. It addresses the issues faced by the system architects and explains the concepts of communication bus, middleware technologies, application interfaces, application encapsulation, application framework based on a client server model. The communication between a client server is explained in detail. [22].

This research involves systematic planning of gathering data and information and its analysis. Which gives scope to find the answers to the proposed research questions through a systematic approach. The method followed here is a qualitative research method where values/measurements are obtained by following this method. This research method includes the following stages:

- Literature Study
- Solution assessment (Discussed in chapter 3)
- Design and Implementation
- Experimentation
- Data Collection and Analysis

Firstly, a literature study is carried out on existing Container Technologies, Optimization flags, and Real-time kernels, and their uses and limitations are analyzed and assessed. Thereby different solutions come into picture during this process of a literature survey, and a particular solution is followed, which is suitable for the proposed questions. By these observations as a basis, the implementation for the solution is designed. Tests are performed on the implemented design in different environments and the impact on the performance of a service, and its request response times are analyzed.

### 5.1 Literature Study

At first, the literature study starts with the study on different environments such as Bare metal, Linux containers, and their limitations are determined (as discussed in chapter 2). In order to understand the working of these environments, a service is made run on these environments and are tested initially with a different inter-frame gap, packet size, and a number of packets. Later a study is conducted on Real-time kernels, and the comparison between the kernels is made to identify which could give a better performance. There are also different GCC Compiler Optimization flags that have been used. Compiler optimization involves three levels of optimization flags, and level 3 optimization flag is used which is best suited for this study.

The study is made on finding different solutions to the problem stated in the chapter, and suitable choices are made and implemented. Thus, the main focus is to analyze the request response time in different environments and see the impact on the performance of service when considering different optimization levels and kernels.

## 5.2 Modeling the service architecture

To deploy the services and to analyze the performance, an architecture model is required. The architecture adopted here is the client-server architecture as shown in the fig 5.1 so that if a request is sent from a client to a server, it responds to a request. Client/server works when the client computer sends a request to the server over the network connection, which is then processed and delivered to the client. A server computer can manage several clients simultaneously, whereas one client can be connected to several servers at a time, each providing a different set of services[22][23].

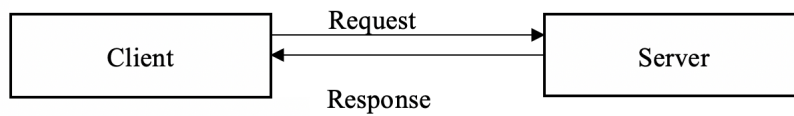


Figure 5.1: Client-Server Architecture with response and requests

## 5.3 Implementation

This section deals with the process of how the implementation has been executed and all parameters that are considered during this implementation are discussed. Here there is a device that has been tested and the source parameters from which the requests are sent to the Device under Test (DUT) are changed according to the survey made and also parameters of the DUT are also varied according to the requirements needed. In this thesis, the DUT has an application that is made run on it where we have two environments to test on (Bare metal and Linux containers). As discussed earlier different kernels are chosen in order to test the performance of a service using different kernels. The optimization flags are also chosen based on the survey made. The Source parameters here are nothing but the parameters of the requests that are sent to the DUT those are the packet sizes(Pktlen), inter-frame gap(WT) and the number of packets(N). We have considered the packet size of 750 bytes and inter-frame gap of 100 microseconds and 1 Million are sent (discussed in chapter 3). The data that is obtained from the

experimentation are taken from the trace files. The Request response time or also called Latency time are collected from these files, and this data is analyzed and studied.

## 5.4 Experimental setup and Data Collection

The experimental setup consists of two test beds those are Bare metal and Linux containers. All these scenarios will have a standard measurement point that collects the measurement data in the data link layer and sends it to the consumers, which build the trace files as the research is oriented towards analyzing the request response time in the virtual and non-virtual environment. The environments under debate are Bare-metal and Linux container. A test bed is shown in Figure 1 has been set up for each environment, and a detailed description of these test beds are shown. Traffic generators are used as an application that is made run on the device. The client serves one million requests to the server with the inter-frame gap of 100 microseconds and the packet size of 750. The packets are travelled from the client to server where measurement point is present in between which measures the times of the packets that are sent and received. The consumer collects the packets and data is stored in the form of trace files where request response time are seen using consumer one-way delay application. These experiments are done in two environments where we are changing from a standard configured operating system to a customized operating system and also performing the same tests by changing the kernel from a generic kernel to a real-time kernel.

### Measurement Point

A Measurement Point (MP) is a system that measures the overall times of the packets received. Both sender and receiver times for a particular packet are obtained by MP with the help of wiretaps, which capture and duplicate the packets at both the ends. The MP consists of Data Acquisition and Generation (DAG) cards, which are synchronized concerning time and frequency by using GPS. These DAG cards have a time stamp resolution of 20ns in the network.

### Consumer

A consumer is a device controlled by the user which accepts the packets as specified by the system and filters the content of the measurement frame. It stores the replicated packets captured by Data Acquisition and Generation (DAG) cards, and the stored files can be used for further analysis.

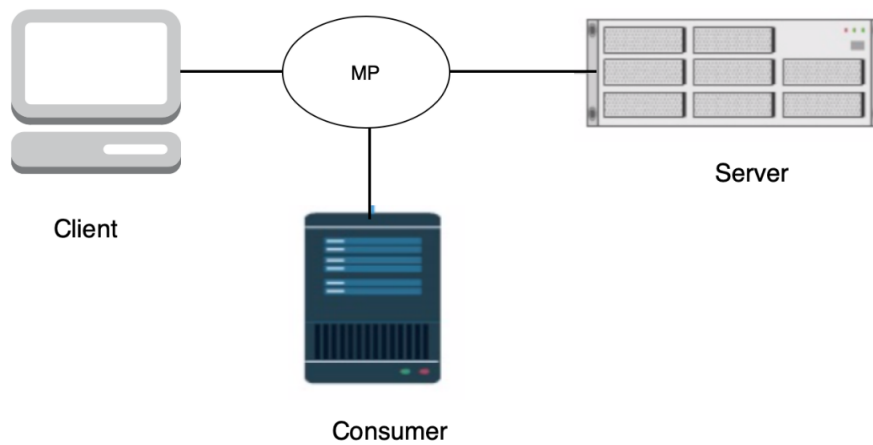


Figure 5.2: Client-Server Architecture

### 5.4.1 Test Beds

#### **Case1: Bare metal**

Specifications of Platform, Kernel and OS under Bare metal case

##### Subcase 1:

**Platform:** Bare metal

**Kernel:** Generic

**Operating System:** Standard Configured OS Ubuntu 18.04 LTS

##### Subcase 2:

**Platform:** Bare metal

**Kernel:** Generic

**Operating System:** Standard Configured OS Ubuntu 16.04 LTS

##### Subcase 3:

**Platform:** Bare metal

**Kernel:** Generic

**Operating System:** Standard Configured OS Ubuntu 18.04 LTS

##### Subcase 4:

**Platform:** Bare metal

**Kernel:** Low Latency /real time

**Operating System:** Standard Configured OS Ubuntu 18.04 LTS

Traffic Parameters	Traffic Specifications
No of Packets	1 million
Packet length	750 bytes
Inter-frame gap	1 microseconds

Table 5.1: Traffic Specifications

The traffic specifications mentioned in table 5.1 is used for all the experiments performed.

System Components	Server
OS	Ubuntu 18.04 LTS
Kernel version	4.15.0-48-generic and 4.15.0-51-lowlatency
RAM	8GB
CPU	4 Cores amd64

Table 5.2: System components specifications in Bare metal Environment(Ubuntu 18.04)

System Components	Server
OS	Ubuntu 16.04 LTS
Kernel version	4.4.0-150-generic
RAM	8GB
CPU	4 Cores amd64

Table 5.3: System components specifications in Bare metal Environment(Ubuntu 16.04)

This section describes an experimental setup where the request response times are measured from the experiments performed. The server is having OS installed on it is running Ubuntu 16.04 LTS and also with Ubuntu 18.04 LTS.

The OS and hardware specifications that are used are mentioned below:

The Same experiment is performed under different OS and Kernel those are mentioned below

### Case2: Linux containers

Specifications of Platform, Kernel and OS under Linux container case

Subcase 1:

**Platform:** Linux containers

**Kernel:** Generic

**Operating System:** Standard Configured OS Ubuntu 18.04 LTS

Subcase 2:

**Platform:** Linux containers

**Kernel:** Low Latency /Real Time Kernel

**Operating System:** Standard Configured OS Ubuntu 18.04 LTS

This section describes an experimental setup where all the service times are measured. The server is having OS installed on it. Linux Containers are installed on Ubuntu 18.04 LTS.

System Components	Server
OS	Ubuntu 18.04 LTS
Kernel version	4.15.0-48-generic and 4.15.0-51-lowlatency
RAM	8GB
CPU	4 Cores

Table 5.4: System components specifications in LXC Environment

The tests are made run using a NTAS web page where the source parameters are mentioned in the command and a measurement streams are selected. The command is made run which performs the tests by serving the mentioned packets with an inter-frame gap and packet length as mentioned in the command from a client to a server. The trace files obtained from the experimentation are viewed using one-way delay application where we obtain the latency times. The same process is repeated for all the experiments performed.

## 5.5 Analysis

After the experimentation is done, the empirical cumulative distribution (ECDF) function and the standard deviations are calculated from the data obtained from experimentation. ECDF is a non-parametric estimator of the underlying CDF of a random variable. It assigns a probability of  $1/n$  to each datum orders the data from smallest to largest in value and calculates the sum of assigned probabilities up to and including each datum. ECDF is simply the probability distribution that we would get when the samples are used to sampling instead of the population.

The empirical cumulative distribution function is usually denoted as  $F_n(t)$  where  $F(t)$  is cumulative distribution function  $I$ =indicator function

$I=1$  when  $Z_j \leq t$

$I=0$  when  $Z_j > t$

$n$ =number of samples(For further details refer to[24])

$$F_n(t) = \frac{1}{n} \sum_{j=1}^n \mathbf{1}_{\{Z_j \leq t\}},$$

Figure 5.3: Empirical cumulative distribution function

This section describes in detail the results and values obtained after experimentation. The graphical representation of the experimentation is shown in this section. Here the analysis of the empirical cumulative distribution of the experimental data obtained by deploying optimized and non-optimized service on Bare metal and Linux containers (virtualized environment) is done.

This section is divided into three parts for the analysis.

- Evaluate system performance by deploying optimized and non-optimized service in Bare metal Ubuntu 18.04.
- Evaluate system performance by deploying optimized service in Ubuntu 16.04 and Ubuntu 18.04.
- Evaluate system performance by deploying the optimized service in Bare metal Ubuntu 18.04 and Linux containers by varying generic kernel and low latency kernel.

### 6.1 Optimized Vs Non-Optimized

**Performance Comparison between an Optimized and non-Optimized service when deployed in Bm using Ubuntu 18.04.**

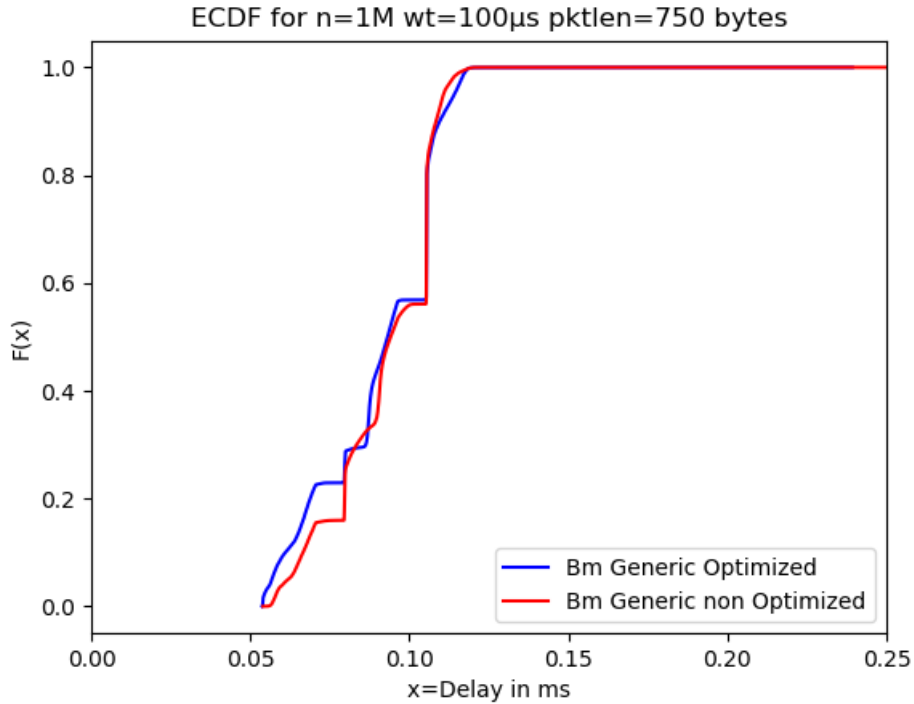


Figure 6.1: Latency w.r.t wait time 0 and 1,000,000 packets

Statistical analysis of Experimental data:

Environment	Bm 18.04 Generic with Optimized application (ms)	Bm 18.04 Generic with Non Optimized Application (ms)
Average	0.090	0.092
Standard Deviation	0.0183	0.0158
Min	0.053	0.0540
Max	0.23	0.26

Table 6.1: stat-1

In the Figure 6.1, the empirical cumulative distribution function is calculated and plotted for a service deployed in Bare metal, and the service is optimized and tested for the same environment. The 50% average of Latency time gives 0.093ms for the service in Bm when Optimized, and for Non-Optimized service, it is 0.094ms. Hence, in this case, Bm Optimized gives better performance.(refer to appendix for ECDF plots)

## 6.2 Optimized Ubuntu 16.04 Vs Ubuntu 18.04

Performance Comparison between an Optimized service when deployed in Bm using ubuntu 16.04 and Ubuntu 18.04

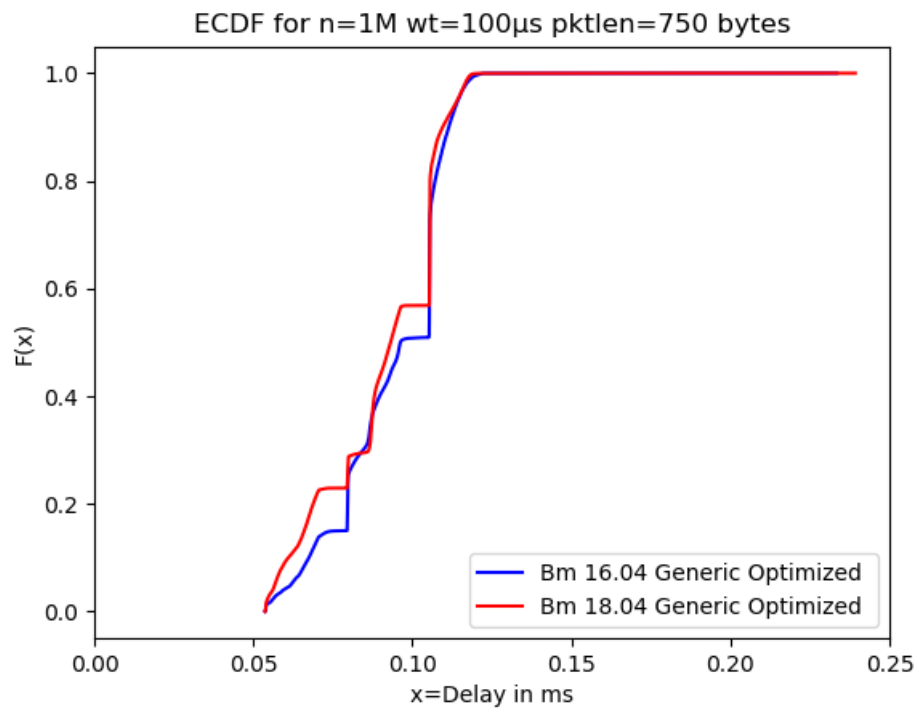


Figure 6.2: Latency w.r.t wait time 0 and 1,000,000 packets

Statistical analysis of fig:

Environment	Bm 16.04 Generic with Optimized application (ms)	Bm 18.04 Generic with Optimized application(ms)
Average	0.093	0.090
Standard Deviation	0.0167	0.0183
Min	0.053	0.0530
Max	0.23	0.23

Table 6.2: stat-2

In the figure 6.2, the empirical cumulative distribution function is calculated and plotted for a service deployed in Bare metal when the service is optimized. The 50% average of Latency time gives 0.096ms for the service in Bm using

Ubuntu 16.04, and for Ubuntu 18.04 it is 0.093ms. Hence Ubuntu 18.04 gives better performance.

## 6.3 Bare metal Vs Linux Container using different kernels

Performance Comparison between Bare metal and Linux containers by using different kernels

### 6.3.1 Performance comparison for optimized service in two different generic kernel environments

In this section, optimized service is deployed in two different environments with the generic kernel.

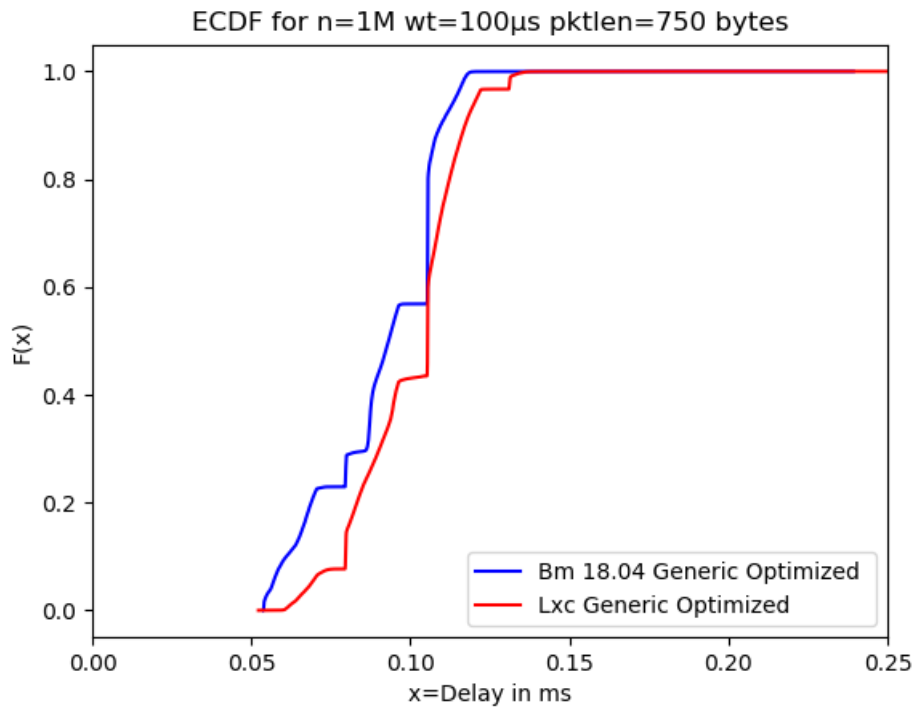


Figure 6.3: Latency w.r.t wait time 0 and 1,000,000 packets

Statistical analysis of above experimental data

In the figure 6.3, the empirical cumulative distribution function is calculated and plotted for a service deployed in Bare metal and Linux Containers with Generic Kernel. The 50% average of Latency time gives 0.093ms for the service

Environment	Bm 18.04 Generic with Optimized application (ms)	Lxc Generic with Optimized application (ms)
Average	0.090	0.099
Standard Deviation	0.0183	0.0161
Min	0.053	0.0520
Max	0.23	0.84

Table 6.3: stat-3

in Bm and Linux Containers; it is 0.10ms. Hence, in this case, Bm with generic Kernel gives the best performance.

### 6.3.2 Performance Comparison of Optimized service in Bm and Bm Low Latency Kernel

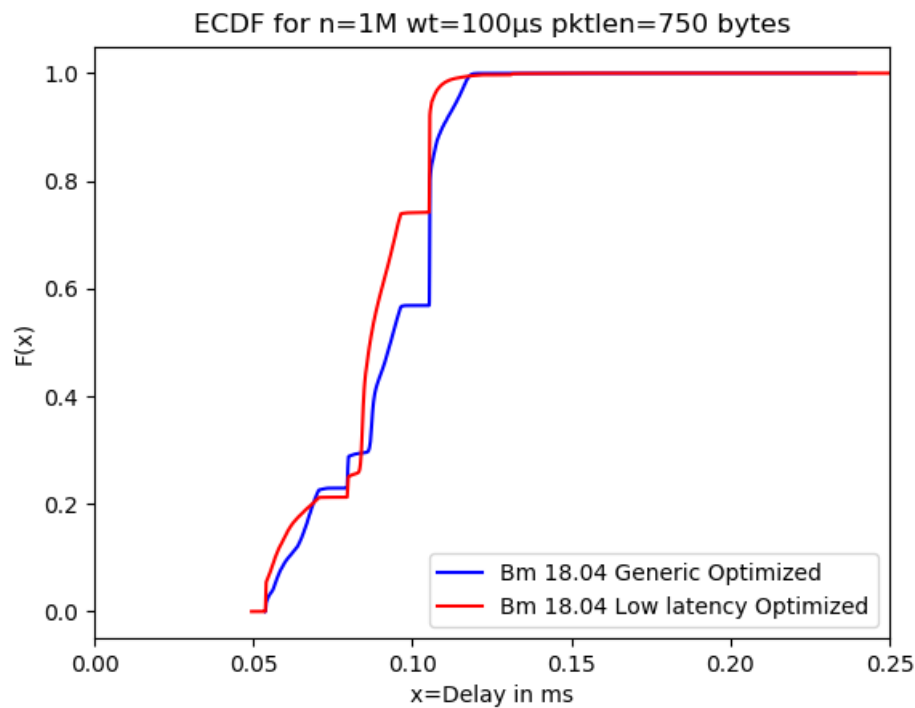


Figure 6.4: Latency w.r.t wait time 0 and 1,000,000 packets

Statistical analysis of Experimental data:

<b>Environment</b>	<b>Bm 18.04 Generic with Optimized application(ms)</b>	<b>Bm 18.04 Low latency with Optimized application(ms)</b>
<b>Average</b>	0.090	0.086
<b>Standard Deviation</b>	0.0183	0.0169
<b>Min</b>	0.053	0.049
<b>Max</b>	0.23	0.40

Table 6.4: stat-4

In the figure 6.4, the empirical cumulative distribution function is calculated and plotted for an Optimized service deployed in Bare metal using generic and low latency kernels. The 50% average of Latency time gives 0.093ms for the service in Bm when Optimized using the generic kernel, and for Low Latency, it is 0.086ms. Hence, in this case, Bm Optimized Low Latency Kernel gives better performance.

### 6.3.3 Performance Comparison between Optimized services deployed in LXC using generic and Low Latency Kernels

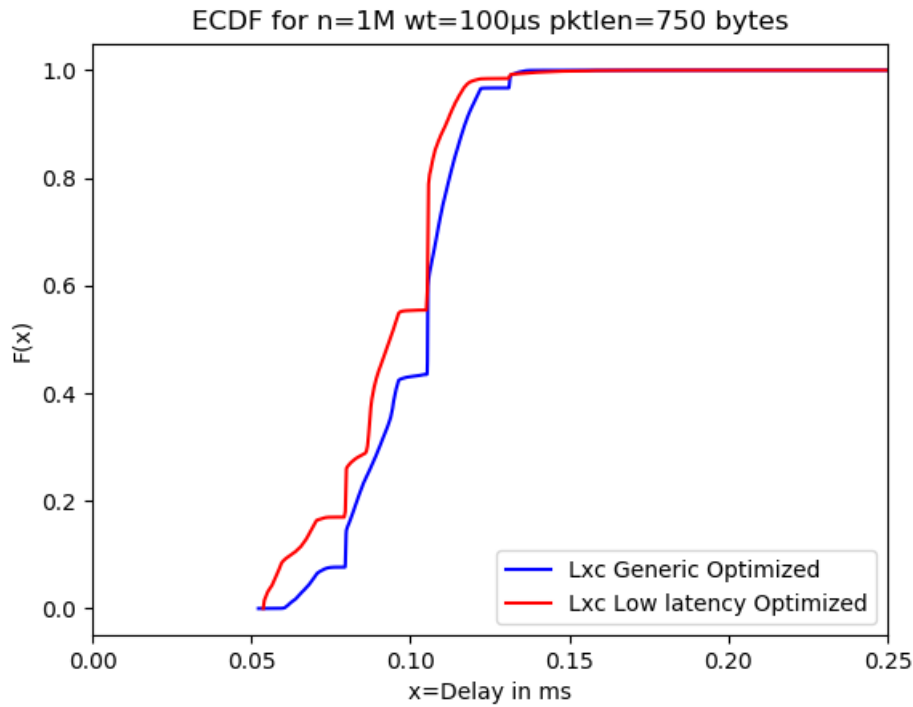


Figure 6.5: Latency w.r.t wait time 0 and 1,000,000 packets

Statistical analysis of experimental data:

Environment	Lxc Generic with Optimized application(ms)	Lxc Low latency with Optimized application (ms)
Average	0.099	0.092
Standard Deviation	0.0161	0.0238
Min	0.0520	0.0530
Max	0.84	4.06

Table 6.5: stat-5

In the figure 6.5, the empirical cumulative distribution function is calculated and plotted for an optimized service deployed in Linux Containers using generic and Low Latency Kernel. The 50% average of Latency time gives 0.10ms for the

service in LXC when Optimized, and for LXC Low Latency it is 0.093ms. Hence, in this case, LXC Low Latency gives better performance.

### 6.3.4 Performance Comparison between BmLL VS LXCLL

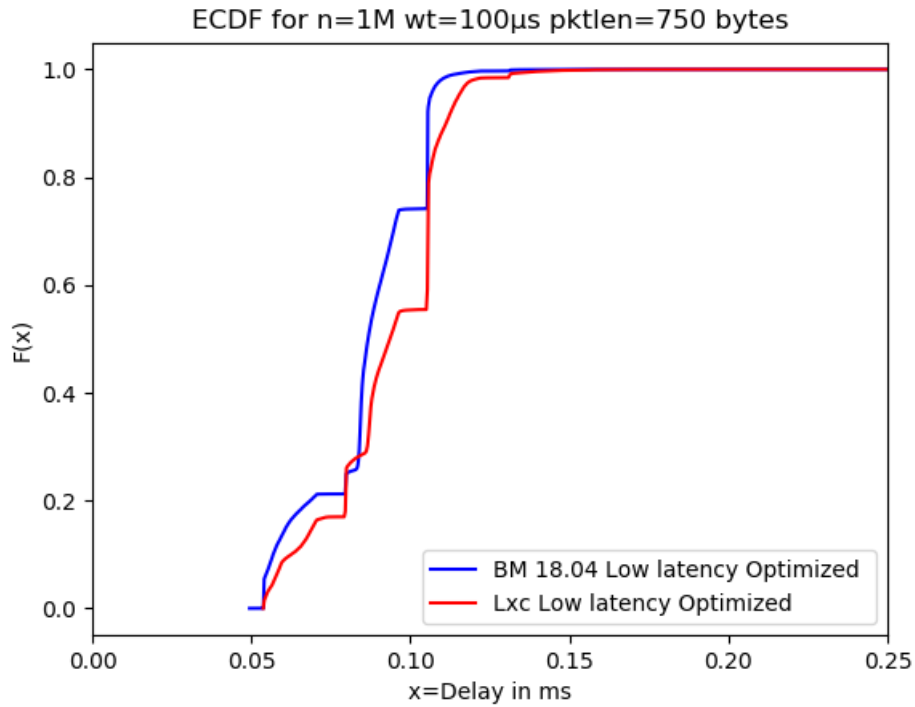


Figure 6.6: Latency w.r.t wait time 0 and 1,000,000 packets

Statistical analysis of experimental data:

Environment	Bm 18.04 Low latency with Optimized application(ms)	Lxc Low latency with Optimized application(ms)
Average	0.086	0.092
Standard Deviation	0.0169	0.0238
Min	0.049	0.0530
Max	0.40	4.06

Table 6.6: stat-6

In the figure 6.6, the empirical cumulative distribution function is calculated and plotted for a service deployed in Bare metal and the Linux Containers. The service is optimized, and Low Latency Kernel is made use in both cases. The 50% average of Latency time gives 0.086 ms for the service in Bm when Optimized, and for Linux Containers, it is 0.093ms. Hence, in this case, Bm Low Latency gives better performance.

## Chapter 7

---

### Conclusion and future work

In this Paper Request response time is calculated by sending 1 million UDP packets with 100microseconds and 750 Packet length to the Optimized service deployed in different environments such as Bare metal and Linux Containers by varying generic Kernel to Low Latency Kernels. These request response time are analyzed by using ECDF.In the Fig 7.1 50% average request response time is calculated for each case. By this analysis, the conclusion made is Bare metal Low Latency kernel has better performance over other environments.

This study is to evaluate the request response time of a service when a service is deployed in bare metal and Linux Containers.

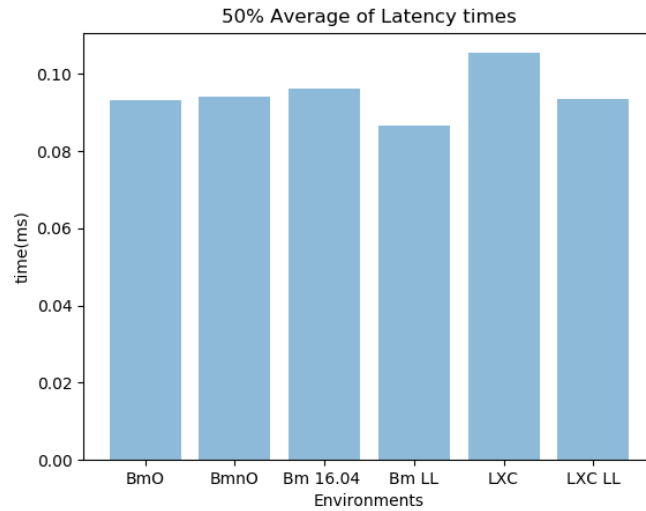


Figure 7.1: Comparison of 50% average values

## 7.1 Research Question and Answers

**RQ 1: If we use a personalized kernel for customized platforms such as Bare metal and Containers. Do we optimize the performance of a server?**

**A:** Yes, we can optimize the performance of a server by using the personalized kernel for customized platforms. In this study using low latency kernel in Bare metal and Lxc scenario, there is a 50% average gain in performance from 0.094ms to 0.086ms in the later case from 0.10ms to 0.093ms.

**RQ 2: When can we achieve the gain in the performance of a service by doing Optimization?**

**A:** By choosing personalized kernel called low latency kernel, and by using compiler optimization flag `-O3`, there is a gain in the performance from 0.093ms to 0.086ms.

## 7.2 Future work

This study is to evaluate the performance of a service in a virtual and non-virtual environment by considering the request response time as performance metric.

For future work, it would be exciting to experiment and note the request response time values in various environments like Hypervisors Using Xen, and also ICC Compiler can be used further optimize.

---

## References

- [1] O. Khedher and M. Jarraya, "Performance evaluation and improvement in cloud computing environment," in *2015 International Conference on High Performance Computing Simulation (HPCS)*, Jul. 2015, pp. 650–652.
- [2] a. X. Liu and K. C. and, "Evolution pattern for Service Evolution in Clouds," in *2012 International Conference for Internet Technology and Secured Transactions*, Dec. 2012, pp. 704–709.
- [3] S. Zhang, S. Zhang, X. Chen, and X. Huo, "Cloud Computing Research and Development Trend," in *2010 Second International Conference on Future Networks*, Jan. 2010, pp. 93–97.
- [4] S. Kamboj and N. S. Ghumman, "A survey on cloud computing and its types," in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, Mar. 2016, pp. 2971–2974.
- [5] F. F. Moghaddam, M. B. Rohani, M. Ahmadi, T. Khodadadi, and K. Madadipouya, "Cloud computing: Vision, architecture and Characteristics," in *2015 IEEE 6th Control and System Graduate Research Colloquium (ICSGRC)*, Aug. 2015, pp. 1–6.
- [6] S. Hassan, A. A. kamboh, and F. Azam, "Analysis of Cloud Computing Performance, Scalability, Availability, Security," in *2014 International Conference on Information Science Applications (ICISA)*, May 2014, pp. 1–5.
- [7] N. Jain and S. Choudhary, "Overview of virtualization in cloud computing," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, Mar. 2016, pp. 1–4.
- [8] A. B. S, H. M.J, J. P. Martin, S. Cherian, and Y. Sastri, "System Performance Evaluation of Para Virtualization, Container Virtualization, and Full Virtualization Using Xen, OpenVZ, and XenServer," in *2014 Fourth International Conference on Advances in Computing and Communications*, Aug. 2014, pp. 247–250.
- [9] "What is Bare Metal? - Definition from Techopedia." [Online]. Available: <https://www.techopedia.com/definition/2153/bare-metal>

- [10] S. Singh and N. Singh, “Containers amp; Docker: Emerging roles amp; future of Cloud technology,” in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, Jul. 2016, pp. 804–807.
- [11] Kovács, “Comparison of different Linux containers,” in *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, Jul. 2017, pp. 47–51.
- [12] V. Singh and S. K. Peddojuf, “Container-based microservice architecture for cloud applications,” in *2017 International Conference on Computing, Communication and Automation (ICCCA)*. Greater Noida: IEEE, May 2017, pp. 847–852. [Online]. Available: <http://ieeexplore.ieee.org/document/8229914/>
- [13] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in *2015 10th Computing Colombian Conference (10CCC)*, Sep. 2015, pp. 583–590.
- [14] “Low latency and real-time kernels for telco and NFV | Ubuntu.” [Online]. Available: [https://www.ubuntu.com/engage/kernel-telco-nfv?utm\\_source=facebook\\_social&utm\\_medium=social&utm\\_campaign=FY19\\_Cloud\\_Kernel\\_Whitepaper](https://www.ubuntu.com/engage/kernel-telco-nfv?utm_source=facebook_social&utm_medium=social&utm_campaign=FY19_Cloud_Kernel_Whitepaper)
- [15] S. Zhong, Y. Shen, and F. Hao, “Tuning Compiler Optimization Options via Simulated Annealing,” in *2009 Second International Conference on Future Information Technology and Management Engineering*, Dec. 2009, pp. 305–308.
- [16] Lei Wang, Boying Lu, and Li Zhang, “The study and implementation of architecture-dependent optimization in GCC,” in *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, vol. 1, May 2000, pp. 253–255 vol.1.
- [17] M. Al-Mulhem and R. Al-Shaikh, “Performance Evaluation of Intel and Portland Compilers Using Intel Westmere Processor,” in *Modelling and Simulation 2011 Second International Conference on Intelligent Systems*, Jan. 2011, pp. 261–266.
- [18] “Everything You Need to Know about Linux Containers, Part II: Working with Linux Containers (LXC) | Linux Journal.” [Online]. Available: <https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-ii-working-linux-containers-lxc>

- [19] N. P. Desai, “A Novel Technique for Orchestration of Compiler Optimization Functions Using Branch and Bound Strategy,” in *2009 IEEE International Advance Computing Conference*, Mar. 2009, pp. 467–472.
- [20] R. S. Machado, R. B. Almeida, A. D. Jardim, A. M. Pernas, A. C. Yamin, and G. G. H. Cavalheiro, “Comparing Performance of C Compilers Optimizations on Different Multicore Architectures,” in *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Oct. 2017, pp. 25–30.
- [21] J. Sahoo, S. Mohapatra, and R. Lath, “Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues,” in *2010 Second International Conference on Computer and Network Technology*, Apr. 2010, pp. 222–226.
- [22] D. Serain, “Client/server: Why? What? How?” in *International Seminar on Client/Server Computing. Seminar Proceedings (Digest No. 1995/184)*, vol. 1, Oct. 1995, pp. 1/1–111 vol.1.
- [23] Kai-Seung Siu and Hong-Yi Tzeng, “On the latency in client/server networks,” in *Proceedings of Fourth International Conference on Computer Communications and Networks - IC3N'95*, Sep. 1995, pp. 88–91.
- [24] Stephanie, “Empirical Distribution Function / Empirical CDF,” Mar. 2017. [Online]. Available: <https://www.statisticshowto.datasciencecentral.com/empirical-distribution-function/>

## Appendix A

---

# To plot empirical cumulative distribution function

```
1
2 import numpy as np
3 import matplotlib.pyplot as plt
4 data = np.loadtxt(r'''C:\Users\Mohit\Desktop\2019\lxcv''')
5 data1 = np.loadtxt(r'''C:\Users\Mohit\Desktop\2019\newlxc1''')
6
7 data =data*1000
8 data1=data1*1000
9
10
11 y_values =[]
12 y_values1 =[]
13
14 raw_data = np.array(data)
15 new_data = np.unique(data)
16 cdfx = np.sort(new_data)
17 x_values = np.linspace(start=min(cdfx),stop=max(cdfx),num=len(cdfx))
18 size_data = raw_data.size
19
20 def get_values():
21     for i in x_values:
22         temp = raw_data[raw_data <= i]
23         value = temp.size / size_data
24         y_values.append(value)
25     return y_values
26
27
28
29
30 raw_data1 = np.array(data1)
31 new_data1 = np.unique(data1)
32 cdfx1 = np.sort(new_data1)
33 x_values1 = np.linspace(start=min(cdfx1),stop=max(cdfx1),num=len(
    cdfx1))
34 size_data1 = raw_data1.size
35
36 def get_values1():
```

```
37     for i in x_values1:
38         temp = raw_data1[raw_data1 <= i]
39         value = temp.size / size_data1
40         y_values1.append(value)
41     return y_values1
42
43
44 get_values()
45 get_values1()
46
47
48
49 color = 'tab:red'
50 plt.xlabel('x=Delay in ms')
51 plt.ylabel('F(x)')
52 plt.title('ECDF for n=1M wt=100 s pktlen=750')
53 plt.plot(x_values, y_values, 'b', label='lxc generic ')
54 plt.plot(x_values1, y_values1, 'r', label='lxc low Latency')
55 plt.legend(loc='lower right')
56 plt.show()
```