Experimental evaluation of a CPU Live Migration on ARM based Bare metal Instances

Ilias Avramidis

A thesis submitted in partial fulfilment of the requirements of Liverpool John Moores University for the degree of Doctor of Philosophy

June 2022

Table Of Contents

ACKNOWLEDGEMENTS

I would like to sincerely thank my remarkable supervision team, Dr Michael Mackay, Dr Posco Tso and Dr Rubem Pereira. Their patience, support and belief in me and my abilities throughout all those years was remarkable. I would like to personally thank Dr Michael Mackay for the valuable guidance and encouragement to undertake a research degree with his expertise, knowledge and support during the studies. Especially during the difficult period of the lockdown due to the pandemic, his support to encourage me to not give up was very important to me. It was an honour for me to be one of his students and he was a mentor for me all those four years. Also, I would like to thank Dr. Takahiro Shinagawa for his contribution as advisor.

Furthermore, I would like to express my appreciation to the Xvisor Development Community for the cooperation, helping me to solve and overcome experimental obstacles.

Additionally, I would like to thank the Department of Computer Science for the lab facilities and all the hardware equipment offered to me in order to perform and complete my experiments.

Finally, I wish to express special thanks to my friends and my family for their support during all those years.

ABSTRACT

The advent of 5G and the adoption of digitalization in all areas of industry has resulted in the exponential growth of the Internet of Things (IoTs) devices, increasing the flow of data that travels back and forth to a centralized Cloud data centre for storage, processing, and analysis. This in turn puts pressure on the intermediate edge and core network infrastructure as traditional Cloud Computing is not ready to support this massive amount and diversity of devices and data. This need for faster processing, low latency and higher network consistency makes a case for Edge Computing solutions.

However, applying Edge Computing as a solution to overcome the network performance limitations that exist on an “IoT to Cloud” architecture while continuing to use Virtualization technology for system utilization is a bit of an oxymoron. Virtualization increases performance overheads, while sharing network resources among users and applications creates further bandwidth limitations and latency since communications are still served through the same physical network interfaces. The demand for network and system consistency, finer security and privacy has led to the deployment of Bare metal instances. Bare metal instances are nothing more than traditional servers that lack the virtualization layer offering native performance to the user. Furthermore, the rise of the ARM processors and the introduction of cheap low power architectures targeted to the Edge introduce a compelling new candidate platform especially on Bare metal instances.

Live migration is a valuable tool for increasing applications and users’ mobility, service availability offering workload balancing and fault tolerance. However, live migration is tied to the existence of a virtualization layer therefore implementing a live migration process on Bare metal instances is very challenging. To the best of our knowledge, there is no existing proposal for a Bare metal live migration scheme on ARM based systems. Therefore, this thesis presents a novel design, implementation, and evaluation of an ARM based live migration scheme for Bare metal instances suitable for modern Edge Computing Micro Data Centres. Our experimental evaluation confirms the effectiveness of our novel design as well as highlighting the importance on identifying the number of registers that describe and are critical for the reconstruction of the CPU state at the destination.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AaaS	Analytics as a Service
ARM	Acorn RISC Machine
AI	Artificial Intelligence
APIPA	Automatic Private Internet Protocol Addressing
CCA	Confidential Compute Architecture
CDCs	Cloud Data Centres
CDNs	Content distribution networks
CPU	Central Processing Unit
CSPs	Cloud Service Providers
DCs	Data Centres
DTB	Device Tree Blob (Binary)
DMA	Direct Memory Access
DMZ	Demilitarised Zones
ENs	Edge Nodes
GDPR	General Data Protection Regulation
IaaS	Infrastructure as a Service
ID	Identification Number
IoT	Internet of Things
IIoT	Industrial Internet of Things
IT	Information Technology
IoE	Internet of Everything

ISA	Instruction set Architecture
IANA	Internet Assigned Numbers Authority
IOMMU	Input Output Memory Management Unit
KVM	Kernel-based Virtual Machine
LCD	Liquid-Crystal Display
LTE	Long Term Evolution
LwIP	Lightweight TCP/IP stack
MaaS	Metal as a Service
MB	MegaByte
MEC	Mobile Edge Computing
ML	Machine Learning
MMIO	Memory Mapped Input Output
NAS	Network Array Storage
NIC	Network Interface Card
OS	Operating System
PaaS	Platform as a Service
PIC	Programmable Interrupt Controller
RDP	Remote Desktop Protocol
RPI	Raspberry Pi
RTT	Round-trip Time
SaaS	Software as a Service
SAN	Storage Array Network
SD	Secure Digital

SSH	Secure Shell
SLAs	Service Level Agreements
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
VMs	Virtual Machines
VMCS	Virtual Machine Control Structure
VMM	Virtual Machine Manager
VMX	Virtual Machine Extensions
VPN	Virtual Private Network
VT	Virtualization Technology
QoS	Quality of Service
WAN	Wide Area Network

LIST OF PUBLICATIONS

CONFERENCE PAPER

I. Avramidis, M. Mackay, F. P. Tso, T. Fukai and T. Shinagawa, "Live migration on ARM-based micro-datacentres," 2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, 2018, pp. 1-6.
URL:
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8319241&isnumber=8319155

The paper gives a high-level overview of the components that take place during a Bare metal live migration as well as an overview of our proposed approach on implementing that. The information on this paper is linked and related to the literature review as covered in the thesis (Chapters 2 to 5).

II. Avramidis, M. Mackay, F. P. Tso, R. Pereira, "CPU Live Migration on ARM Based Bare metal Edge Nodes", 2021 (Pending submission)
The paper gives a high-level design overview of our CPU state live migration implementation as well as presents the experimental results and evaluation of the experiment that we conducted. The information on the paper covers mostly the Chapters 6 to 8 of the thesis.

POSTER

III. Avramidis, M. Mackay, F. P. Tso, T. Fukai and T. Shinagawa, "Live migration on ARM-based Micro-datacenres", Department of Computer Science, Liverpool John Moores University, Byrom Street, Liverpool, L3 3AF, UK

1 Introduction

1.1 Motivation

The Cloud is undeniably now the most preferred hosting environment for business workload and applications, delivering a wide range of IT resources on demand. Recent surveys have shown that 90 per cent of businesses prefer to use Cloud services, hosting approximately 60% of their workload in 2019 while this trend is expected to reach up to 94% in 2021 [223, 225, 227]. Nowadays, more and more businesses are being converted to data businesses; this transformation has increased the data volume by up to 63% of the overall business workload [223, 228]. The handling and management of that amount of data is a challenging and demanding task, which requires huge amounts of computing and processing power. Therefore, companies are adopting one or a combination of the Cloud computing deployment methods to support this. Furthermore, the rapid growth of the Internet of Things (IoT) and Industrial Internet of Things (IIoT) devices, as well as a wide range of smart technologies, remarkably increases the amount of data produced on a network. Based on current surveys, the number of IoT and other smart devices will also increase from 17 billion this year, up to 29 billion by 2022 while the world population was only 7.5 billion in 2019 and is predicted to reach approximately 8.5 billion by 2030 [227, 228, 229]. This further highlight how the number of Internet-connected devices per person that require cloud processing is likely to increase.

Although until now the Cloud has been ubiquitous, promising durability, high rates of availability, and redundancy, but the rapid growth of these internet-connected and interconnected devices in a network producing a massive variety, volume and velocity of data may expose some of the weaknesses in Cloud Computing. These mainly focus on the degradation of the intermediate network performance like poor bandwidth performance and latency that leads to high round-trip delays and network bottlenecks. Therefore, current centralised cloud architectures cannot scale to support and efficiently cover these upcoming future trends. The answer and solution to this impending issue is the adoption of modern decentralised IT architectures, shifting from a centralised plan to a distributed scheme which replaces the all-to-Cloud model with multi-locations and Micro Data Centres. There is where Edge and Fog computing architecture solutions find their ultimate expression. Although Edge and Fog computing principles are not new, with the rise of IoT technologies, such design schemes may see massive implementation in future network environments.

Both Edge and Fog computing technologies provide an enhanced architectural solution that enables fast and efficient processing, computation, and analysis of the data produced by IoT and Smart technologies [51]. Figure 1.1 illustrates a high-level overview of a communication scheme and the coexistence of Cloud, Fog and Edge technologies in a layered architecture. In this model, Fog computing acts like an intermediate layer that connects Edge Computing devices to the Cloud. Fog computing also works as a management mediator between the Cloud infrastructure and the Edge devices in a network infrastructure. Edge Computing describes a decentralised, distributed IT processing architecture that enables mobile computing, IoT, and a variety of smart services to coexist and interact in an efficient manner. As the name Edge implies, data processing, computation and storage is happening at the Edge of a network, near to the source where the data are produced. In that way, Big Data and latency sensitive services achieve faster, efficient, real-time processing and analysis, saving network bandwidth and round-trip delay times.

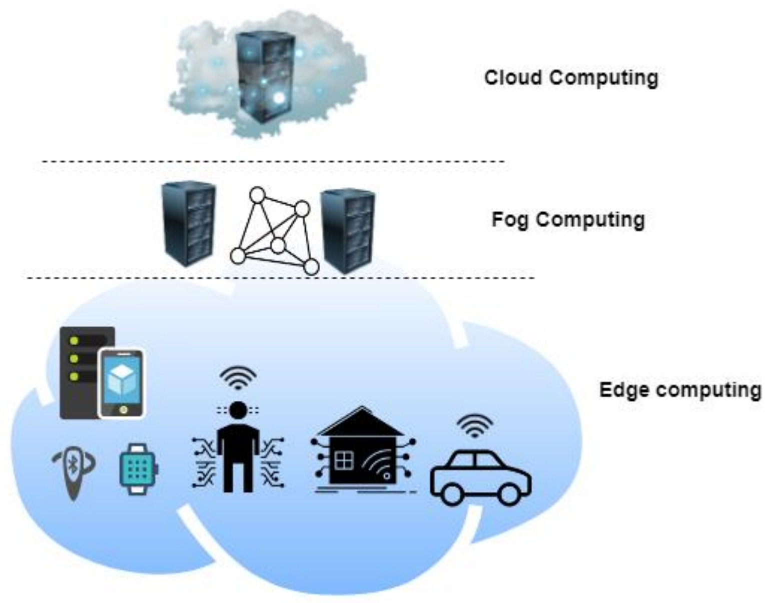


Figure 1-1 Data processing layered architecture from Edge to Cloud

However, some data produced by multi-edge networks where multi-edge networks is considered several independent networks managed by individuals or CSPs, may still need to be gathered on a centralised Cloud backend location for further data processing, storage, and analysis which does not work efficiently especially when it relates to Big Data, real-time, or mission critical sensitive applications that demand computation in a timely manner. However, these applications are very sensitive to system and network performance degradation like latency where tasks such as processing, analysis and computation need to be achieved in a timely manner.

The emergence of Micro DCs has made Edge Computing implementation much easier and more attractive than ever before [17]. Modern Edge network schemes will utilise Micro DCs as the core infrastructure of processing, computation and storage of the data. Micro DCs are effectively miniature versions of traditional data centre infrastructures that are implemented as a modular, containerised, grid of small racks. They can therefore be deployed in strategic locations as near as possible to the source of the data on a variety of distributed locations, collecting, processing and analysing data, thereby eliminating the need for transmission to a centralized Cloud data centre.

Moreover, this data often contains sensitive or private information that is important to the user, so data integrity and privacy are also of high importance. Due to these factors multi-tenant infrastructures such as virtualised Cloud data centres, or even virtualised Micro DCs, may be exposed to security threats and are vulnerable to security holes putting not only customer's information but the availability of services at risk. Therefore, Cloud vendors are introducing a new hardware platform as a service, named Bare metal instances. More and more Cloud providers are including Bare metal servers in the Cloud infrastructure service suite, delivering it as a service to customers. These Bare metal servers, or Metal as a Service (MaaS) as it is also called, are nothing more than traditional, powerful hosting server towers with the only difference being that they are dedicated to a single customer, eliminating the demand for a virtualization layer, providing native, high computing performance, higher levels of security and privacy.

Although the deployment of Micro DCs, through the implementation of Edge Computing architectures in combination with Bare metal instances meets the core IoT requirements of low latency, fast convergence and data processing with enhanced data privacy, there are some parameters that still need to be considered like redundancy and high availability. Bare metal instances do not utilise virtualization, which has an impact on the redundancy, failover and fault tolerance attributes offered by live migration techniques which are tightly bonded with the existence of virtualisation technology. Live migration in particular is a powerful tool that finds extensive implementation in virtualized environments among servers, offering a high degree of workload redundancy and availability, quality of service and load balancing. Therefore, recent research work has been done on the development of a live migration scheme on Bare metal instances. Fukai's [37, 38] research introduces a novel live migration model taking place on Bare metal instances. Based on that novelty, a live migration scheme is feasible even with the lack of a virtualisation layer on systems running an x86 processor architecture.

Moreover, modern ARM architecture processors, focusing on Edge-to-Cloud data centres, are attracting a lot of interest in this arena. The characteristics of small-footprint design, power-efficiency and low price make them an ideal alternative implementation for Bare metal Micro Data Centre infrastructures, achieving both performance and lower overheads. The implementation of ARM in data centre infrastructures is an emerging concept with a lot of interest from the academic and research community. However, there is a lack of support for performing live migration on ARM systems, especially when it comes on Bare metal instances where the virtualization layer is eliminated. To the best of our knowledge, live migration is possible to support only among x86 Bare metal instances but there is currently no implementation under development to perform such a task on the ARM architecture.

In this research, we explore a novel experimental approach for how to develop and perform a live migration process on ARM based Bare metal instances, which could be used to support Edge Computing through Micro DC deployments.

1.2 Aims and Objectives

Key aim of this research is the design and implementation of a CPU live migration process as this takes place among ARM based Bare metal instances where CPU live migration is considered the group of registers that composes and describes the CPU state at a specific time where the migration is performed.

Current Cloud Computing solutions, created to support and provide efficient, on-demand processing and high availability, are limited in providing improved responsiveness and low latency as the growth of smart and electronic devices increases rapidly. Therefore, modern Edge Computing solutions through the utilization and deployment of Micro Data Centres can better interact and serve the needs of the Internet of Things technologies. This thesis aims to provide a better understanding of how the implementation of Bare metal Micro Data Centres at the edge can be embedded within current structures; explore the parameters required for their successful adoption and use; design and develop a model to provide higher availability, consistency, and fault-tolerance through a live migration mechanism. Additionally, modern ARM processors targets at the edge, offering high compute performance and power efficiency in affordable prices. Therefore, in this thesis for the first time, we aim to investigate and integrate an ARM based live migration scheme evaluating the migration time in comparison to the required state as well as its impact on the wider system performance.

To fulfil these aims, the following list of objectives have been carried out.

1. To review current Cloud Computing architectures looking at the limitations and drawbacks of the adoption of an ‘all-to-Cloud’ architecture model which lead us to the adoption of Edge Computing
2. To assess the value and the role of virtualization in modern IoT environments as well as the performance limitations that modern real-time, high-performance applications face caused by virtualization overhead.
3. To identify computational architecture solutions such as Bare metal instances along with their contribution to delivering an efficient architecture at the edge improving the network and system performance.
4. To analyse ARM architecture virtualization extensions and ARM based hypervisors and the lack of delivering a high available, consistent fault tolerance infrastructure.
5. To design an ARM based CPU live migration scheme applicable to Bare metal instances in order to cover the existing gap for provide a reliable infrastructure.
6. To identify the essential group of registers to implement a CPU live migration.
7. To develop a CPU live migration model between ARM based instances.
8. To conduct computing simulations, testing host CPU load performance and migration system metrics.
9. To point out the lack of definition as Guest state on an ARM system in relation to the required CPU registers in order to perform optimum performance during a CPU state live migration.

1.3 Novel Contributions

Live migration is a valuable tool, especially on the edge where all the intense processes such as real time data processing and computation take place. However, live migration on Bare metal instances is a very challenging task, especially on ARM based systems with few available solutions. This research aims to deliver the following novel contributions:

- In Chapter 6, is given a novel design and approach to discover and capture ARM processor state on an ARM based host system. Through this novel contribution we cover our 5th aim and objective as explained above.
- In Chapter 7, a novel approach and code implementation to preserve, store and migrate processor state at the source host and receive the migrated registers at the destination host. That novel contribution covers our both 6th and 7th aims and objectives.

- Also, in Chapter 7, a novel approach and code implementation to adapt the migrated registers at the destination host. That novel contribution focuses on performing the 7th aim and objective.
- In Chapter 8, is given a novel point for research and evaluation as far as concern the definition of Guest state in relation to the required group of migrated registers in order to successfully re-assemble the Guest state on a destination system. Through that novel contribution we cover our 8th and 9th aims and objectives as described above.

1.4 Thesis Structure

The following is the structure of the remaining chapters of this thesis:

Chapter 2 presents the challenges that emerge due to the rapid growth of the IoT and digitalization in the business industry as well as the adoption of Edge Computing architecture solutions through the emergence and installation of Micro Data Centres at the edge. Chapter 3 then describes the performance limitations of Virtualization in the data centre infrastructure as well as the rise of Bare metal instances delivering higher end-user performance, while Chapter 4 discusses the importance of live migration on virtualized systems, highlighting the existing lack of support on Bare metal instances. Chapter 5 discusses the rapid shift in adoption of ARM processors in data centre infrastructure, the benefits, and the differences from the x86 architecture. In Chapter 6 we introduce our high-level design for the support of an ARM based CPU live migration scheme for Bare metal instances as well as the components and phases which are part of it, while in Chapter 7 we provide a thorough explanation of the source code for our novel proof-of-concept implementation. Chapter 8 provides an illustration of our implementation, presenting our lab environment experimental results and outcomes. Lastly, Chapter 9 gives a conclusion of the thesis and identifies areas for future work.

2 The revolution of IoT and IIoT in the industry

2.1 The Rise of Smart Technologies drive to Business Digitalization.

The advent of 5G technology is driving an increase in digitalization and digital transformation, forming an increasingly connected society with ultra-fast connection speeds, low latency and high bandwidth, radically changing the flow, process and utilization of data. The International Data Corporation (IDC) as many other official research institutes, according to recent estimates, predict that the number of connected electronic devices and objects to the internet forming the Internet of Things (IoT) will reach approximately 40 billion in 2020 while it is expected to increase up to 72 billion devices in the next few years, ten times more than the global population [28, 34, 248].

The Internet of Things is a consequence and outcome of digitalization. While digital transformation is all about data, IoT is the link between the devices that generate and produce those data. The IoT is a huge ecosystem of connected and interconnected "things" where the word "things" refers to a wide range of digital devices, physical objects, and people that communicate through the Internet. Nowadays, digital sensors and micro-chips transform objects and devices to a smart version of them, making them active members of the IoT ecosystem. Smart devices are considered to be any electronic device or object capable of getting assigned an IP address and connecting to the Internet sharing data with each other. We can simply visualise the IoT as a giant network of advanced digital technologies that share and exchange digital data.

The IoT completely transforms day to day operations, affecting the way that people access resources and services as well as the way that they communicate and interact with modern, smart digital technologies. It is estimated that people now spend almost 80% of the day in interaction with IoT devices like smartphones, tablets, and laptops [28, 34]. However, the IoT is not just about smartphones, tablets and TV sets anymore. The range of the IoT is enormous covering most modern technologies. The emergence of innovative digital technologies like Smart Cities, Smart Homes and Home Automation makes IoT an integral part of our life.

Smart homes are becoming the revolution in the field of commercialization where in cooperation with Artificial Intelligence (AI) and Machine Learning (ML) they give life and voice to home appliances like home assistance and smart housekeeping like smart hoovers. People are able through wearable and embedded devices like smartwatches and smartphones to control, manage, monitor and even schedule housekeeping tasks and other daily jobs. Consumers communicate and control those smart devices remotely using an app or by accessing a web-based interface. Additionally, consumers can schedule to turn on/off the lights, or the thermostat, they can monitor and protect their properties through smart security cameras and even control the doors with smart video doorbells. Moreover, a variety of wearable health devices like fitness bracelets help on monitoring, measuring and reporting of the health status of the consumer.

The emergence of IoT technologies has completely transformed cities' infrastructure converting them to smart cities to significantly improve the quality of citizens' and visitors' life. A wide range of digital technologies covering every layer from the air to the street to the underground, affecting cities' infrastructure through many aspects like environmental, social and financial. This urbanization trend is a significant factor that plays a core role in the digitalization process of cities. Surveys have shown that more than half of the global population is becoming more urban while this is expected to be increased to 68% of the global population in the next two decades [230, 231]. Urbanization increases the investments in the adoption of smart digital technologies which are expected to grow up to 135 billion US dollars on investments by 2021 according to reports from the International Data Corporation institute (IDC) [248].

Smart cities are taking advantage of the digital and information technologies in such a manner that objects and people connected at the public network can communicate, having as their primary scope the improvement of the urban life. The core attribute in the success of a smart city ecosystem is ubiquitous Internet connectivity, known as hotspots. Cities hotspots work via Access Points that offer connectivity and enable access to the Internet almost everywhere. Cities are transformed into huge hotspots where inside that "dome" of connectivity people and objects are sophisticated enough to communicate, exchanging important digital information making cities a better place to live where citizens gain access to the Internet by using a variety of connectivity mediums.

As more and more businesses follow the digitization evolution, business digitalization becomes a priority in most industrial sectors like manufacturing, healthcare, agriculture, farming etc in order to stay competitive in the business industry. Each sector needs to think and act differently, to be adjusted in this new digitalization era. Digitization is the key to the door leading to business digitalization as well the driving force in this rapid spread of IoT digital technologies that have been massively adopted both by industrial and corporate sectors as well as by individuals, improving our daily life in many aspects.

Digitization is the migration process of transforming all analogue, physical assets in business to a digital format [28]. Through that conversion process, several digital data are produced, requiring appropriate treatment for the business to benefit. There is where digitalization finds application, through the adoption of smart digital technologies used to improve the business operations and frameworks in order to increase efficiency, business productivity, revenue and the quality of the end-user product.

The Industrial Internet of Things (IIoT) helps to manage that rapid shift, offering a link between the physical and the modern, digital world, making smooth the transition to business digitalization. Surveys have shown that the IIoT digital technologies will be adopted by 80% of the industrial businesses, particularly in manufacturing, healthcare and agriculture [230, 231].

As Figure 2.1 illustrates, the IIoT is a subset of the broader IoT ecosystem. Consumer IoT devices aim to improve the quality of a person's everyday life such as the smart homes and smart cities digital infrastructures, while IIoT on the other hand, focuses on industrial sectors' productivity and internal operations that take place providing increased visibility on how users interact with products, services, or applications like Figure 2.2 illustrates.

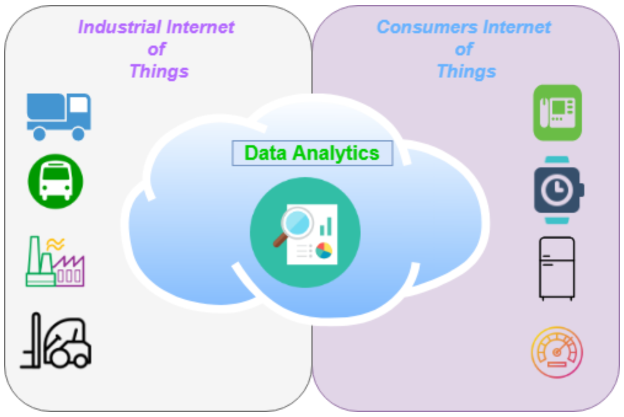


Figure 2-1 IoT and IIoT

Under the scope of an IIoT ecosystem, machines, devices and people communicate, intelligently connected through the network, sharing actionable information optimizing strategies across several departments. On production, digital sensors and many other digital objects become the source of meaningful data where advanced, sophisticated data analytics formulas take place giving valuable insights into internal operations of the business. Sensors do self-monitoring, reporting in real-time progress of the product state, identification of potential errors as well as stock capacity of warehouses. In that way, smart factoring helps on proactive maintenance, avoiding downtimes or production outages, automating, and speeding up the supplies actions increasing production rates and avoiding human errors. Smart factoring limits the need for human presence while enabling the ability of remote monitoring and management of production systems and processes through smart IoT technologies.

Furthermore, the application of IoT in agriculture allows farmers to get precious insights on the growing status of the fields, greenhouses or the vehicles. Digital sensors installed throughout the fields collect data about the growing rates, or weather-related information, allow farmers to make fast adjustments based on given conditions.

Figure 2-2 Digitalization in the Business Industry

2.2 Limitations in the IoT to Cloud model

Nowadays, the IoT is spanning everywhere we look, from devices that people own and carry with them, home supplies, to being embedded in factory equipment and even part of a city's infrastructure.

As the volume of IoT devices connected to the Internet increases rapidly, the amount of raw data generated by those devices is also growing remarkably fast. In a recent report of IDC [248], is estimated that the number of the connected “things” will be 42 billion generating approximately 80 zettabytes of data by 2025. Some of these can be small chunks of data, indicating real-time measures gathered by sensors while others can be large data generated by streaming sources like voice and images.

The world of IoT devices is composed of a diversity of device platforms where hardware and system specifications vary. In most of the cases, IoT devices are embedded systems with a microchip installed offering limited performance capabilities where most can perform some basic on-device processing tasks such as to execute compression and encryption mechanisms. They are not enhanced with enough compute power, storage, memory or network capacity in order to accomplish intensive, real-time processing tasks like type conversions, sorting mechanisms, execution of advanced queries or the support of advanced analytic mechanisms. IoT devices deal with an immense amount of real-time digital information that needs to be stored, analysed, processed and be accessible anywhere from other digital devices and services, tasks that the IoT devices themselves are not capable of performing. Moreover, it is logical and consequent that individuals and businesses cannot afford an investment to build and form a local, on-premise data centre, in order to manage, maintain, and handle the processing of that massive amount of collected information. On-premises processing and maintenance tasks of both underlying infrastructures as well the hosted workload is very demanding, costly, time- and high energy-consuming procedures. So, the real question when it comes to the IoT is, where data will be stored and how fast it will be processed.

There is where the Cloud-IoT model finds application where IoT serves as the source of data and the Cloud serves as a centralised information hosting centre. During the last decade, Cloud computing has proved to be the most preferred choice as a hosting environment, dominating in the IT industry, delivering an efficient, flexible, durable and scalable hosting environment suitable for both application development and deployment. Following that IoT growth, CSPs offers a broad range of services and powerful tools, designed to collect, store, process and manage data of any type, size and kind. Recent surveys indicate that 60% of businesses utilise a Cloud data centre hosting up to 90% of workloads while approximately 90% of the business industry trust and utilise a Cloud service deployment method that Cloud Service Providers support [224, 225, 230].

Below is an overview of the reasons that make the cloud indispensable to the success of the IoT:

- Diversity of hardware.
- Diversity on Operating system and software requirements.
- Storage capacity.
- Remote processing and compute power.
- Cloud-based development.
- Big Data analytics.
- Mobility.

As we have already explained, the IoT is a massive, complex network of diverse systems that expand beyond traditional devices like laptops and smartphones, including a broad range of daily objects such as vehicles, TV sets, thermostats, smoke detectors, surveillance cameras and more. Each of them supports a different type of operating system, software and data types which increases the complexity of the required underlying management infrastructure. Online users through mobile and web applications are connected to distinct Cloud-based backend platforms each of which is configured to support and cover a different kind of request and need. On the other hand, businesses start adjusting their workload, processes and services based on Cloud principles, by developing Cloud-based applications. The Cloud, thanks to abstraction (virtualization layer), offers a pallet of cross-platform development tools while supporting a wide range of storage options satisfying all kinds of tastes and needs.

Big data uses them to apply advanced analytics which is referred to as the Artificial Intelligence implementation. Big data analytics are advanced sophisticated models applied on an ocean of data in transit meaning real-time ingested data or data at rest ingested by storages and warehouse locations. Big data analytics is a science in high demand delivering valuable information to the business industry and society. Businesses try to convert data into actionable insights that will in turn help to discover and unlock business potential; and understanding more about the business context makes the use of big data analytic tools help in making better decisions about internal operations which leads to a more efficient system in the future. Companies adjust, improve and optimize business operations, day-to-day processes and actions in order to increase business revenue and quality of the end product. Big data helps in identifying or predicting potential faults, preventing future failures in production while completely transforming business frameworks and working methods.

An essential problem when dealing with Big Data is the resource issue. Businesses need to be very careful on architectural choices when dealing with Big Data, advanced analytics, and artificial Intelligence on the design of the underlying network and server infrastructure, database architecture options. The higher the volume of the data, the more resources are required like memory, processors, and disk performance to analyse them. The need for a durable infrastructure, that can handle an enormous amount of real-time data led businesses and organizations to turn to the Cloud. Most known CSPs (Amazon, Microsoft, Google) due to increased demand of Big Data analytics, now offer their own Big Data solutions (Dataflow, BigQuery) which deliver Analytics as a Service (AaaS).

Making the most of Big Data requires not just having access to the right big data analytic tools but also to maintain a scalable, resilient, high-performance infrastructure capable of fulfilling the following elements: data collection, data storage, data analysis and data visualization as Figure 2.3 illustrates. Ideally, such an infrastructure must cover the following performance characteristics:

- Maximum usage and high available memory capacity
- Storage capacity (petabytes)
- Fast I/O disk performance
- Solid network performance
- Parallel processing

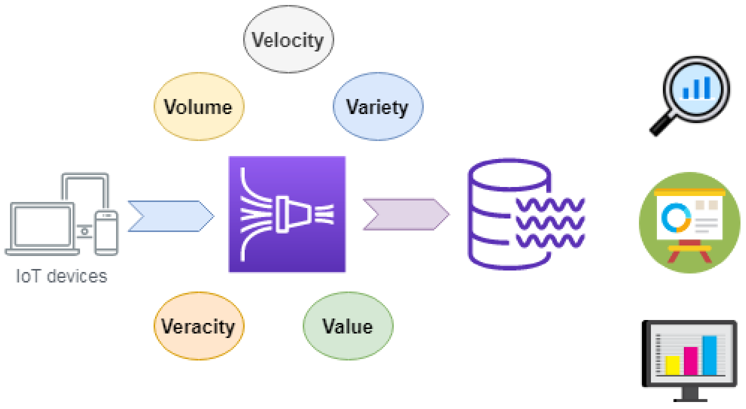


Figure 2-3 Big Data characteristics

The rising volume, velocity and variety of data generated by IoT technologies, as well as the need for data-intensive processing applications such as big data analytics, AI and ML exert pressure and test the limits and strengths of the current IoT to Cloud infrastructure to the fullest from processing, memory, and storage to networking data access. Modernized infrastructures need to be capable of handling intensive processing and big data in real-time analytical tasks particularly as more and more businesses tend to combine these advanced technologies and workload with the Cloud.

The IoT to Cloud architecture has established a centralised client-server model where most of the traffic ends up in a Cloud Service Provider datacentre backend. Although the Cloud is self-sufficient on resources offering an advanced palette of big data analytical tools as well as integrated artificial intelligent environments for machine learning developers, however, that centralised, "all in one" architecture model does not work efficiently and beneficially for all kinds of workload and preferences in the digital age that govern us. Such an architecture suffers from vulnerabilities that affect and have a major impact on the networking and processing performance.

A number and variety of hardware digital devices and sensors produce an enormous amount of data. Data that represent information, requests or multimedia access the public network (internet) via different types of medium such as wireless, Ethernet, fibre, or a cellular network that serve on different speeds and bandwidth. A packet that accesses the Internet is routed through several networks known as "hops" until reaching the destination, routing and switching mechanisms taking place and performed on that packet during that journey.

As data reaches a Cloud data centre premises, the trip does not end there. Data need to traverse through the internal CSP network. In practice, a Cloud data centre consists of a number of internal regions, distributed around the world in well-designed locations. Each region communicates with each other through a private network owned by the Cloud Service Provider. That means that the traffic inside that network remains private and does not mesh with other public, widely used networks. However, data needs additionally to traverse through that network in order to find the right region and server where the required backend service is running and hosted. A backend refers to any service, application, website, or big data analytic tool that consumers try to access for personal use.

A Cloud data centre is composed of several racks of servers shared among multiple users as Figure 2.4 illustrates. This task is performed by the adoption and use of Virtualization Technology and with a virtualization layer called Hypervisor. The hypervisor is responsible for managing all the inbound and outbound traffic. So, once the data reaches that physical resource, the physical server, then a number of virtualization mechanisms are applied to that data to get it routed to the right destination virtual environment. Virtual switching, routing, packet and schedule processes and many other mechanisms are applied to packets on the way in and out of the physical server.

In order to evaluate and discuss the performance limitations of a Cloud model architecture, we first need to define some factors based on affecting the performance such as:

- Network capacity
- Number of I/o request
- Average response time
- Workload
- Throughput
- Number of connections
- Multi-tenancy

So, through the process as described we can also identify the following major limitations strictly related to performance degradation:

- Network latency
- Bandwidth limitations
- Packet switching
- Process Scheduling

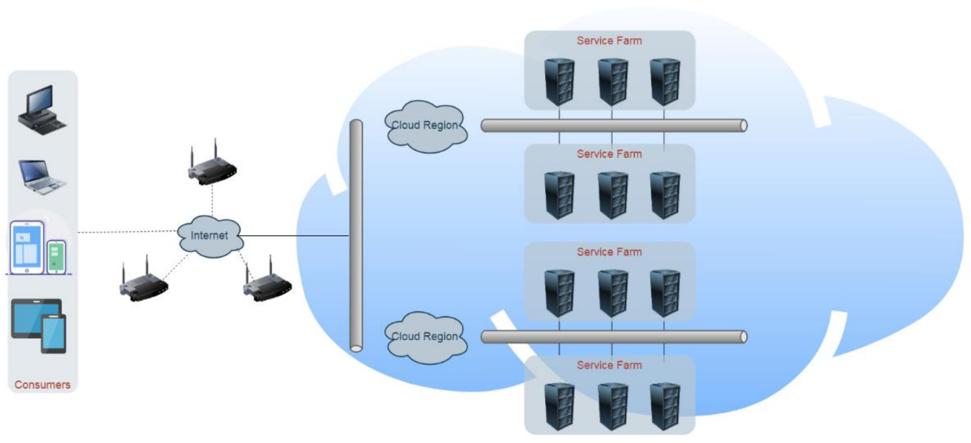


Figure 2-4 Data road path from Edge to Cloud backend

Network performance is based on three factors, latency, bandwidth, and throughput where all three are related and affect each other. Latency is depending on the distance and bandwidth of the transmission channel. Latency is another way to describe delay on a network which is measured as the time it takes for data to reach the destination from the source. This is usually called round-trip time (RTT) that determines the time taken for data to reach their destination and back to the source. RTT measurement is very important to TCP/IP applications where frequent acknowledgement packets must be exchanged between the source and destination peers.

As we mentioned, data on an edge network accesses the Internet through a variety of ways, where medium bandwidth speeds vary. Moreover, the distance from an edge network to the entry point of a CSP data centre infrastructure vary. Any traffic that passes through the Internet there is no guarantee of successful and fast communication. Based on those facts, latency-sensitive applications should be aware and take into consideration those limitations. Network latency can be noticed in both public and private networks of CSPs premises. Imagine that data on average passes through up to five intermediate public networks, making use of at least two medium types (cellular, fibre) where different packet conversions are applied on the data to match and meet the requirements of each medium. Those processes produce quite a delay on packet transmission. Furthermore, bandwidth speed plays an important role in the simultaneous transmission of data allowing the initiation of multiple user connections. Common enterprise networks support from 25Mbps up to 10GB of speeds. Low speed affects the total migration time of the systems' state especially during the memory migration from source to destination where the amount of memory can reach up to 32Gbs [265].

Modern networking architectures take advantage of network-based storage solutions adopting either a storage area network (SAN) or network-attached storage (NAS) implementation. Both network-based storage architectures can be implemented locally on-premises or on Cloud. Low network speed in relation to data large in size affects the data storage and disk I/O requests causing bottlenecks and delays in data processing. Network performance degradation issues are not the only limitations on an IoT to Cloud architecture. Multi-tenant Cloud infrastructures also suffer by significant performance vulnerabilities.

2.3 From the Cloud back to Edge Computing

We are living in the digital age of the Internet of Everything (IoE) as first defined by Cisco, [34] where people, data, things, and processes communicate in a sophisticated manner and come into interaction through the Internet, which has transformed information into actionable, useful insights more than ever before. In a recent forecast from IDC the number of generated data produced by an enormous number of IoT and IIoT devices is estimated to reach the volume of 79.4 zettabytes until 2025 [230, 231]. In addition, Cisco predicts that by 2021, 90% of the Internet traffic and workload will be Cloud-based and only 10% will be hosted on-premises [230, 231]. As the volume of data sources and the data generated from those sources increasing rapidly, a centralised IoE to Cloud model faces lots of limitations especially in terms of network performance degradation and slow convergence of the information. Adopting such a centralised Cloud model, all data from a variety of sources flow to a Cloud data centre through the Internet taking advantage of the great benefits that Cloud computing provides [2], offering a pool of unlimited resources with high-performance computing power and storage capacities. However, in the case of IoT, that extreme amount of digital information puts tremendous pressure on the network and on the Cloud backend infrastructure while processing and storing all data on the Cloud increases the cost. In order to tackle the performance limitations from which an IoT to Cloud model suffers, businesses and vendors need to adopt durable, sustained, robust solutions, capable of handling and processing that massive volume of information. Those attributes create a case for the Edge Computing approach.

Although Edge Computing is not a new concept, it has started becoming popular again as an architecture implementation in modern networks due to the rapid and exponential growth of the IoT. Sending data back and forth to a centralised Cloud data centre puts pressure on the Internet and cloud infrastructure. As the number of connected devices increases the amount of data is rising extremely fast. Latency and higher roundtrip times become major problems for administrators. In that scale of digital information adopting a Cloud model is not any more efficient. The need for faster processing, lower latency and higher network consistency makes a case for Edge Computing solutions.

Edge Computing brings data processing, computation, analysis and storage closer to the source where data are being generated and gathered instead of relying on a centralised Cloud computing model where data need to be transmitted thousands of miles away [8, 12, 17]. Edge Computing solutions work efficiently for many smart technologies and modern applications which are latency-sensitive and based on real-time processing.

In general, Edge Computing is defined in the literature as "*the application of a distributed computing architecture model where information processing takes place closer to the edge of a network where things, people, objects and services access and consume that information*" [29, 42].

2.3.1 Fog and Edge Computing deployment architecture models

Although Edge and Fog computing are used interchangeably in industry there are some major implementation differences [51]. Both technologies are governed by the same principles in terms of processing data and information locally, closer to where the data originated from rather sending them to a centralised Cloud data centre. By maintaining data locally, we leverage the compute power of a local network to carry out a variety of administrative tasks that traditionally would be handled by a Cloud Service Provider. However, the key difference between the two is mainly focused on the location where those tasks find implementation.

In order to understand more about the differences between Edge and Fog computing, we need to describe what it is that we call "edge" in computing. The word "edge" describes a logical layer rather than a specific physical location of where the edge devices and end users take place and exist. It is the logical layer where all kinds of devices and things are connected to the same network. In industry, the location of the edge depends on where data processing is by design to take place in order to deliver and fulfil business requirements. In Edge Computing, data processing takes place either on devices itself or on edge nodes (ENs). On the other hand, Fog computing handles the data as passing in and out through network devices with sufficient processing power such as switches, firewalls, and routers of the network. For convenience, we treat those two technologies as one logical layer, called the edge layer as illustrated in the following Figure 2.5.

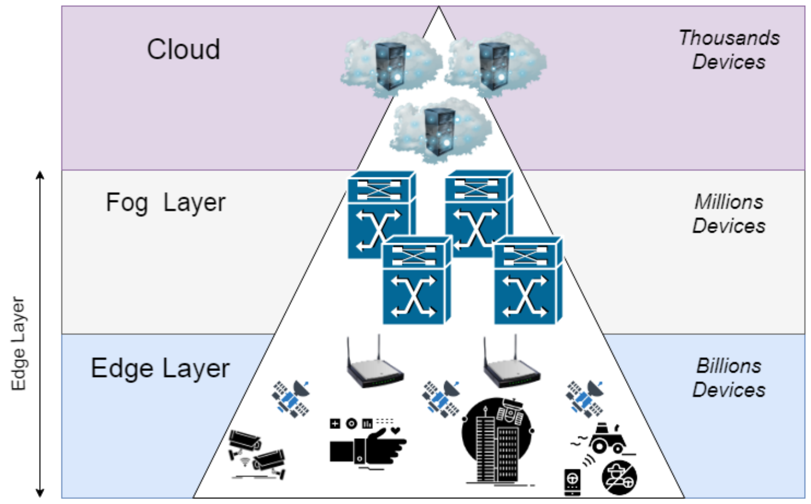


Figure 2-5 Edge to cloud networking layers

A Fog computing architecture works as a mediator tier that extends the Cloud computing operation closer to the edge of a network, closer to the source of data. Fog computing maintains Cloud features like networking, virtualisation, storage, and redundancy while meeting the requirements of the sensitive applications that reside in an edge network offering Quality of Service (QoS), higher service level agreements (SLAs) and low latency. A Fog architecture is described as a decentralisation model where resources take place in distributed locations between the edge network and the Cloud. The goal of such implementation is to improve the overall network performance bringing computing power closer to where it is needed, reducing latency and network delays which occur during the long transmissions back and forth to a centralised Cloud data centre backend. Additionally, a fog infrastructure allows us to apply a higher level of security by implementing several demilitarized zones (DMZs), filtering the configuration of grouping patterns of the same type of data and deploy firewall rules for finer security.

Edge and Fog computing are interconnected. The first one introduces a management layer which efficiently handles data generated by edge devices forwarding results and core operations to the fog layer, while the second one is responsible for the transmission of data to the Cloud. The adoption and deployment of both technologies has radically changed and transformed the traditional IoT to cloud model opening the road for modern IoT technologies to benefit from low latency, higher bandwidth capacity, and faster processing and improved network consistency.

As we mentioned, the term Edge Computing describes a logical layer that brings data computation closer to the user rather than a specific location where Edge Computing should be applied and deployed. Deploying an Edge Computing architecture depends on several factors and varies from infrastructure to infrastructure. In the case of large scale IoT networks a variety of devices are connected using Wi-Fi, 4G or the latest 5G technologies producing data that varies in processing and transmission requirements and needs, something that makes Edge Computing architecture deployment very challenging in order to provide full network coverage.

Gopika et al. [251, 252, 253] classify Edge Computing architecture designs in three implementation models however, from our point of view and understanding of the term edge we focus on two of them as illustrated in Figure 2.6. The first deployment model is by utilizing resource rich edge nodes where user-applications connect while the second model is by utilizing a heterogeneous group of edge devices. On the left side the figure describes a model consisting of a number of resource rich edge nodes installed in a distributed way. Those nodes could be powerful, high-performance servers capable of hosting, processing and interact with end-user applications while on the right side, a variety of computing and networking resources act as processing and computation point such as routers, switches, access points and embedded devices capable of processing data.

Based on the experiments that Gopika et al. [251, 252, 253, 254] were conducting, taking as use cases gaming and other resource intensive applications, the necessity of the Edge Computing paradigm is shown in order to meet the latency and delay requirements of modern real-time applications providing great performance to the end user. Even in cases where Edge Computing is deployed through limited computing resources, it remarkably improves the overall system and network performance experience giving positive feedback on adopting Edge Computing architectures on several application scenarios.

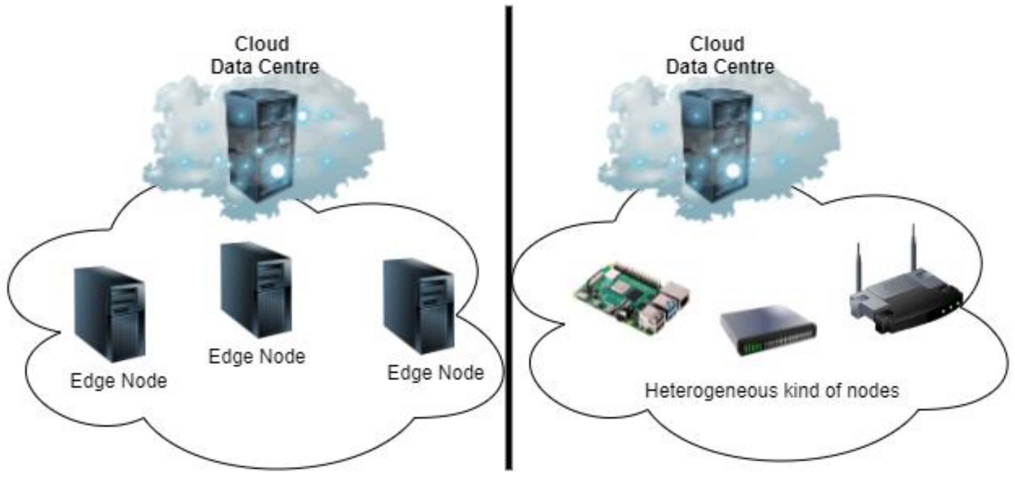


Figure 2-6 Edge Computing architecture deployment models

2.3.2 What are the benefits of Edge Computing?

Edge Computing completely transforms the way of handling, processing and analysing data coming from a variety of IoT devices and a diversity of IoT technologies. It is letting us achieve workload decentralisation helping on offloading network traffic. In that way, even with the rapid growth of IoT technologies that require real time processing, no impact should be noticed on the network performance. By adopting such a distributed architecture model, computational tasks and needs can be delivered at the edge as data are collected and where they are produced, eliminating the need for sending data long distances to a centralised Cloud data centre location. Some of the great benefits that Edge Computing implementation is offering are mainly focused around the following areas:

- Improved performance
- Data privacy and security
- Reduction of operational expenses

2.3.3 Improved Performance

Edge Computing enables the ability of accurate, faster processing by improving the network performance while remarkably reducing the latency. Reduction of latency is one of the driven, key benefits of Edge Computing, making it attractive to many businesses. Latency is an important attribute for most latency-sensitive, real-time applications and modern IoT technologies with strict network performance requirements such as video streaming, online gaming, autonomous vehicles, and artificial intelligent and big data analytics. The edge is located closer to the end-user and IoT devices rather than a hundred miles away like a Cloud data centre. The Edge Computing layer reduces the distance between edge devices and the Cloud. It achieves that through the relocation of where processing takes place on a network. Placing computational tasks at the edge of a network, closer to where data initially was generated, helps to prevent data from travelling long distances, crossing a variety of unknown networks until they reach a centralised Cloud data centre. Shorter distance means lower roundtrip times, lower latency, speeding up data processing and system communications.

On a traditional IoT to Cloud model, data follows two-way processing streaming paths, an upstream and downstream. As Figure 2.7 points out, upstream is the stream where data flow from IoT devices to the cloud, while downstream is defined as the data flows from the Cloud to end devices. IoT devices not only consume data and information by making requests on Cloud-based services and applications but at the same time become an active source of information, producing data. Users when accessing web-based applications or login on a mobile application, send identification data back to the application backend while after data pass processing, a reply with access permitting or denying goes back to the user. Instead, a well-designed Edge Computing architecture solution can perform a variety of administration tasks, data storage, processing, analysis, caching and computation, offloading the traffic and requests heading to the Cloud.

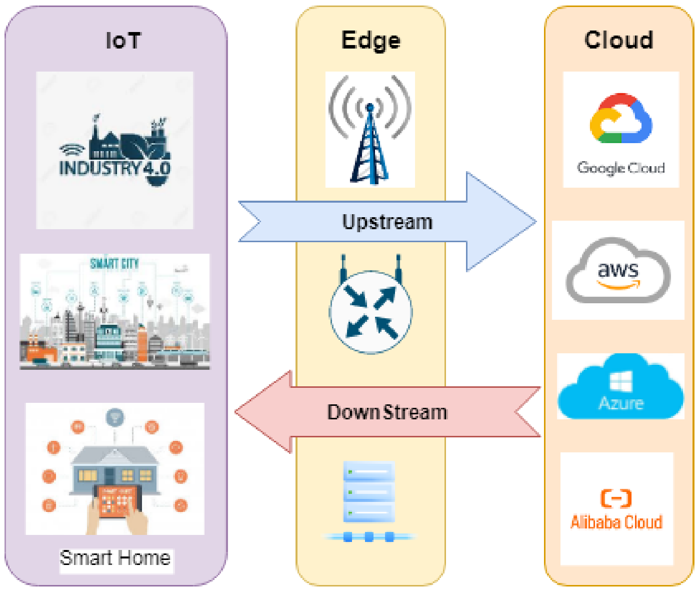


Figure 2-7 IoT to Cloud model

Latency increases according to the distance. The longer the distance that data need to traverse to get processed, the higher the latency of the network. We measure latency in relation to the round-trip time

(RTT) which is the time it takes for data from a source to reach a destination and back again. RTT is one of the most widely used measurements in order to determine or diagnose the reliability and efficiency of a network connection. In some cases, network latency can prove a threat not only for the smooth operation of applications but also to human life. In case of autonomous vehicle technology, vehicles like cars and public transportations exchange information constantly with a centralised system that controls and manages vehicle's route through a satellite. Network Latency or any delay of a command reaching to a vehicle can be proved vital putting human lives in danger.

In addition, low latency helps in achieving higher throughput, freeing faster the available bandwidth. In that way, more open connections can be established in a shorter period and higher throughput which is a significant factor and attribute of applications that make use of TCP/IP connections, which means longer battery life for IoT devices. Most IoT devices are embedded systems, with low hardware specifications and limited power capabilities. Intensive processes in combination with long term connections and the effort to maintain those connections for a long period consumes a lot of energy, exhausting the device's battery life in a short time. Therefore, by implementing connections for shorter periods of time due to short distances, we achieve longer battery life which is a very important aspect especially in agriculture where digital sensors monitoring the field need to get removed in order to recharge, causing downtimes.

Below are some more real-life examples where the Edge Computing architecture solution makes a difference compared with a traditional Cloud-based implementation.

Autonomous vehicles and self-driving transportation are a huge part of modern Smart Cities infrastructures. Automation combines different technologies like image and video processing, GPS navigation control systems, live tracking and traffic management (ATM) systems that demand sub second response times. Vehicles send real-time information like video and image footage feeds to a backend service hosted on a centralised Cloud data centre. Once data gets processed and decisions made then instructions return to the vehicle. If autonomous vehicles must exchange information with a Cloud data centre each time a movement is made, vitally important seconds are added to the overall computation process. A slight delay of the information can prove vital to both passengers and pedestrian life. As we know, it only needs a few seconds of a vehicle deviating from its course to put people's lives in danger.

Reduced latency is not the only benefit of an Edge Computing infrastructure. Increased capacity and availability of network infrastructure and higher bandwidth speeds are some other aspects of the

Edge Computing model. Consider the case where building surveillance cameras transmit live footage of high analysis that needs to be stored and analysed for future use. Streaming that continuously raw data of video, sound data and motion signal to a cloud server puts a significant strain on the Internet and Cloud data centre infrastructure where thousands of cameras are doing the same thing. Gigabytes of data, from a number of sources consume network bandwidth, creating bottlenecks and latency on their way to the Cloud. By relocating storage and processing of image and motion analysis workload to the edge of the network, we achieve offload and distribution of the massive workload, and significant reduction of bandwidth.

Furthermore, Edge Computing works beneficially in cases like video and sound streaming, online gaming platforms, real-time image processing, video surveillance etc. Prior to Edge Computing, face recognition applications were running advanced algorithms on Cloud-based services which would take a lot of time. With the adoption of an Edge Computing model, those algorithms could run locally on the edge saving a lot of power, cost, and bandwidth.

Completing some administrative tasks on data such as conversions, filtering, prioritising help us to manage the information efficiently maintaining a high quality of data at the edge avoiding transferring unnecessary data to the Cloud for further computation and analysis tasks. Edge Computing provides a great fit with data analytics and artificial intelligence processes that demand fast response times, and low latency real-time processing, tasks would be inefficient if data should be transmitted to a centralised cloud data centre.

Edge Computing can also be proved efficient to CDN implementations where reduced RTT and latency are the primary goals. Caching data closer to end-users but also with data crossing lower distances achieves less congestion and fewer bottlenecks that lead to more core bandwidth.

2.3.4 Data privacy and security

In the case of an IoT to Cloud model, data flows through several unknown networks, crossing many networking devices. Once data reaches a Cloud backend server, the virtualization layer is in place. From a security standpoint the virtualization layer hides a lot of security threats. Storing and maintaining data and other sensitive information in a shared environment like a Cloud data centre, exposes data to several security threats [82].

Now more than ever before, with the massive growth of IoT devices and the adoption of Cloud solutions, virtualization on the Cloud will face a lot of challenges around data protection and data rights. Furthermore, with the updated rules as defined by the European Union’s General Data Protection

Regulation (GDPR) finding implementation on everything, data integrity and privacy becomes a major concern for businesses and organizations.

By leveraging the computing power of a local network, under specific region boundaries, we can more easily achieve a finer security definition and control the policies, security rules and regulations that need to be followed up, offering better privacy in order to keep data safe and consistent. In that way, we minimize the potential of network attacks since individual edge networks can be monitored more easily while data and information are exposed on fewer shared users and devices.

2.3.5 Reduction of operational expenses

Another important driving factor that leads to Edge Computing is the operational costs. As the size of a network grows due to the increased demand and growth of IoT devices, the need for higher bandwidth speed and high availability, remarkably increases business expenses. Although the bandwidth is sufficient, ubiquitous and easily accessible, it comes with a cost. Furthermore, storing, retrieving, and accessing data frequently to a cloud storage service increases the costs more than many businesses expected.

Edge Computing reduces the bandwidth costs by applying data processing within the LAN network before data reach the WAN layer. Taking as an example, video surveillance cameras that produce data of HD quality that can reach 1,000GB in a month, streaming that amount of data over an LTE network or a fibre network remarkably increases the cost.

Implementing that intermediate layer of Edge Computing we can distribute the load, performing most processing and storage tasks at the edge and forward the core information or an aggregation of the data.

2.4 The utilization of Micro DCs at the Edge

The demand for instant access to data and information at anytime from anywhere, as well as the need for faster, latency-free processing, leads more and more businesses onto the adoption of the Edge Computing paradigm. Edge Computing describes a topology in which data and information processing, storage and analysis are placed closer to the source of the information, reducing the latency while increasing the available bandwidth of a network. The goal of such an architecture is to avoid long-distance data transmissions back and forth to a centralised cloud data centre. IT administrators have the flexibility to choose and decide whether applications should be more efficient to be hosted on the Cloud or on the edge of the network.

As we described in Section 2.3.1, one of the edge deployment architectures is based on the utilisation of resource rich edge nodes (ENs). In such implementation, edge nodes are expected to be high performance servers fully equipped with all the required resources which are necessary in order to cover network needs such as compute power, storage, and networking infrastructure. The number of ENs varies and depends on several factors such as the number of IoT devices that are connected to a network, the communication medium as well as the type of the IoT devices. The emergence of Micro Data Centres (Micro DCs) could meet the challenges and demands of such an implementation. Micro Data Centres can be a core part of an Edge Computing implementation, sitting along the route between the IoT edge devices and the Cloud. Micro DCs installed in various distributed locations can be configured to gather, store and process data of a single site or branch while they can be a great fit serving as content delivery network (CDN) servers helping in caching data and content at the network edge like Figure 2.8 illustrates.

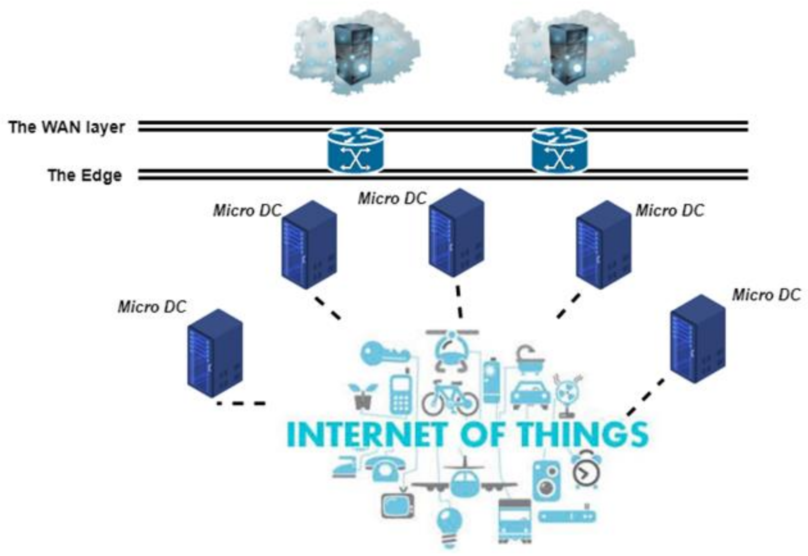


Figure 2-8 Adoption of a Mirco DCs architecture diagram

CSPs install data centre facilities in various locations around the globe, named regions, where each region is sub-divided by two or more availability and redundancy zones. A zone is a distinct, separate, fully equipped data centre having its own power supply and cooling systems, computing power, and storage [263]. Inspired by and adopting that architecture as Figure 2.9 illustrates, Micro DCs could play the role of a zone, formatting a Micro DC triangle on the edge providing high redundancy and availability of services.

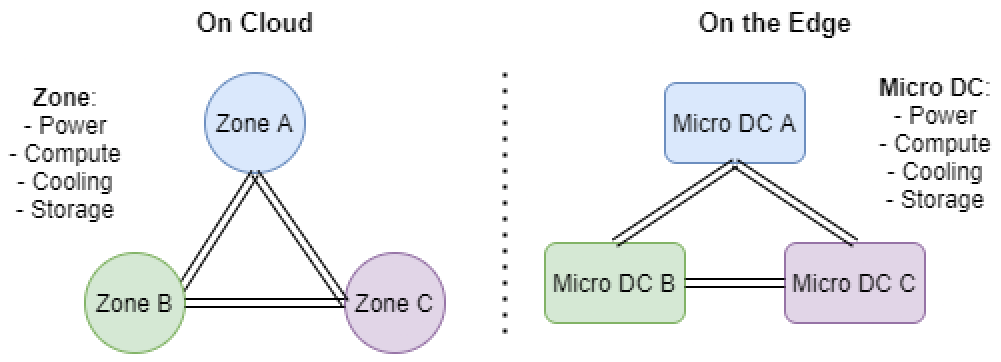


Figure 2-9 Adopting Zone triangle architecture on the edge

A Micro Data Centre as the name implies, is a small footprint of a traditional data centre, typically having the size of a rack, capable of support up to 10 servers and 100 VMs. It is a pre-designed, pre-built, pre-configured, self-contained containerized data centre that aims to provide solutions on a different set of challenges which traditional data centre infrastructures are unable to handle. As a standalone, self-contained system, is fully equipped with all the features that a traditional data centre contains, including networking connectivity, security, cooling, environmental monitoring, power protection and distribution supply, Micro Data Centres play a key role in the Edge Computing implementation. We can think of the Micro Data Centre as a tooling or “medium” of “how” Edge Computing is intended to work and succeed. Through the adoption of a Micro DC it is easier to understand how Edge Computing serves and supports all those benefits that it is promising to provide.

The driving factor that brought the development and rise of Micro Data Centres was the desire for workload distribution to several locations through the adoption of an Edge Computing architecture where processing and a variety of administration tasks would be handled locally instead of transferring data and information thousands of miles away to a centralised Cloud data centre.

Maintaining a data centre on premises is a costly, time-consuming process that requires the need for specialised IT staff and a support department as well the need for a well-designed, secure hosting space for all the server infrastructure. The emergence of Micro DCs came to change the view of hosting a data centre on premises.

Micro DCs are designed to eliminate the limitations and restrictions that a traditional data centre deals with, offering the benefits of reducing of upfront capital investments and utilization costs, reduced footprint and energy consumption rates, while increasing the speed of deployment of an additional data centre offering flexibility and scalability. Micro DCs can be easily deployed and get “repeated” on a site when and where needed in a cost-efficient, flexible manner. They have the capability to perform intense data processing functions such as big data analytics, machine learning and artificial intelligence. Once one is fully utilized; another Micro Data Centre can be added either on the same facilities or at a different site based on the current requirements. Micro Data Centres are shipped fully equipped with all the necessary infrastructure resources, offering a plug and play solution, ready to go as powered on. The standardized model and the compact, containerized style help with fast deployment, increasing scalability, while it can fit anywhere, easily deployed in various locations and situations where a traditional data centre would be impractical or even impossible to fit. The speed of deployment of a Micro Data Centre varies. The more standardized model the more likely to be available ready to be delivered.

Micro DCs offer the following key benefits:

- Low latency: The communication distance of people and devices with a Micro Data Centre edge is a significantly shorter path than communication with the Cloud. Less distance means lower round-trip delay time and low latency. Consequently, achieving faster data transmission and processing times.
- Communication channels: Data makes use of and pass through different type of networks on their way to reach a Cloud edge cluster. Mobile networks make use of satellite communications and infrastructure which significantly increase the latency which causes remarkable degradation performance. Utilizing a Micro DC's architecture, people and devices connected to the Internet by using wireless and local network connections spanning a smaller number of networks through optical fibre infrastructures, make faster and more reliable the interconnection among computing and digital nodes to a server edge cluster.
- Data governance: Most parts of today's business world depend on digital data. Businesses depend on the availability, durability, security, quality, and fast processing of data. Big data analytics models require fast response times, like the stock exchange and any trader company.
- Security: Each Micro Data Centre has a limited amount of computing resources, although enough to perform big data processing and analysis functions. However, the amount of the interconnected computing resources is enough to achieve malicious attacks like a DDoS attack.

2.5 Summary

The increasing pace of IoT devices connected to the Cloud reveals some of its architectural limitations adding extreme pressure on the network and backend infrastructure of the data centres. Network bottlenecks, latency and delays are some of the degradation performance issues that nowadays the centralised all-to-Cloud model must deal with. The demand for low-latency, fast responsiveness and access to data processing, computation and storage brings the need for Edge Computing architecture solutions. Relocating data operations closer to the user, we achieve offload of the global network while offering higher stability, consistency, better data management and finer security.

The emergence of Micro Data Centres, fully equipped, modular, small in scale, easy to be installed and deployed almost everywhere, make them the best solution in such use cases. Micro DCs allow the adoption of a decentralised architecture solving the network performance issues in a cost- and power-efficient way.

In the next chapter we will discuss the performance issues that virtualization technology introduces in computing and the emergence of Bare metal instances as hosting platforms offering higher performance than virtualized.

3 Virtualization Technology

3.1 Overview of Virtualization

Virtualization Technology (VT) plays a key role in the operation and implementation of Cloud computing offering higher utilization performance of the underlying infrastructure on data centre facilities through the performance of server and workload consolidation techniques while remarkably reducing the need for capital investments on hardware resources. Although both VT and Cloud are two independent, distinct technologies we could say that they are strictly bonded together in order to deliver a Cloud-based functionality in an economic, efficient, flexible and reliable way. In its broader sense, Virtualization is the art of science that enables the ability to create multiple virtual entities of an IT object or resource like storage, network and server resources. A single piece of an IT object operates and acts as if it were multiple instances of it [3, 7, 9, 40]. As we described in a previous section, Cloud operation is delivered through the public network (Internet) to the end user, through the form of the three main services, IaaS, PaaS and SaaS. One of the core components in the architecture of all those services is the virtualization layer. Although VT is not visible to Cloud consumers, it is a fundamental part of the Cloud architecture finding implementation through a suite of forms covering a broad range of the IT resource at the backend of a Cloud datacentre infrastructure. Some of those forms which we can discern, and mention are, Storage virtualization, Network virtualization, Data virtualization and Hardware virtualization while Server virtualization has become the most well-known form of deployment not only in cloud data centre infrastructures but in the IT industry in general.

Server virtualization is the biggest trend of all the forms of virtualisation in the IT industry finding extensive implementation in any kind and size of business providing great benefits. One could say that is a combination of forms, merging characteristics of both operating system and hardware virtualization methods. It is the concept of the partitioning of a physical computing system (desktop, server, and laptop) into several distinct, individual virtual environments, simulated computing entities called Virtual Machines (VMs), capable of running and hosting their own operating systems. Each VM is unaware of the existing virtualized infrastructure or the existence of the rest of the VMs. Server virtualization permits the simultaneous installation and execution of multiple OSs and any kind of workloads on the same physical computing system. The most vital role on a virtualized architecture is played by the hypervisor, also known in literature as the VMM (Virtual Machine Monitor). A virtualized server system is divided into four major components following a layered structure where the layers each depend on one another. Starting from the bottom up, the physical hardware called host environment, is the collection of the actual amount of physical hardware resources of the server, like CPU cores, memory, storage, and I/O peripheral devices. On top of that, based on the type of hypervisor that finds implementation on each topology depending on current needs each time, stands either the operating system where the hypervisor layer resides and is equipped with a number of VMs, called host OS (Type 2 - Hosted hypervisor), or directly the hypervisor layer (Type 1 – Bare metal hypervisor) [33]. Finally, at the top stands the operating system running inside a VM called Guest OS.

From a technical perspective, taking a closer look inside a CSP data centre infrastructure as the Figure 3.1 shows, each physical server system is hosting several Virtual Machines that share the same physical hardware resources. Each VM is part of a virtualised private network which is able to communicate with other VMs members of the same virtual network as well with the external world, through the Internet. On each VM is assigned a virtual port attached to a virtual switch where that virtual switch is part of a wider virtual network composed of additional virtual switches and virtual routers that are responsible for passing and routing the traffic from the virtualised infrastructure to the physical networking infrastructure. Furthermore, a cluster of storage devices is available to users, in order to store system information as well as any kind of structured and unstructured data they want.

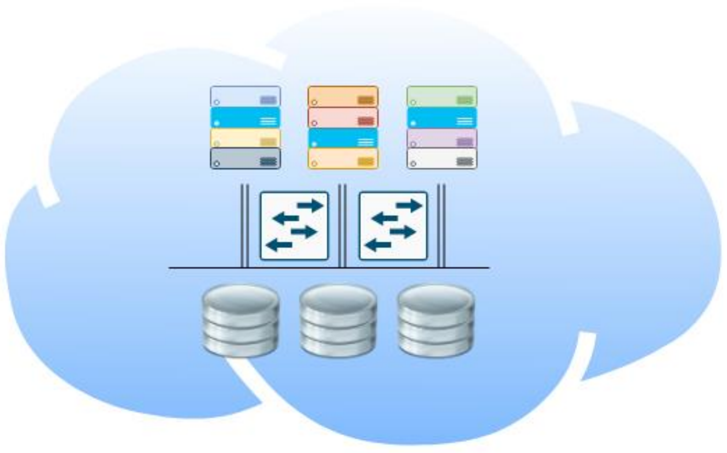


Figure 3-1 Cloud data centre

3.1.1 Types of Hypervisors

The most essential role on a virtualized system is played by the hypervisor, also known as virtual machine monitor (VMM). A hypervisor is a software-based, abstraction layer that has multidimensional roles. The primary scope of a hypervisor is to abstract and isolate the hardware resources from the upper operating system and applications running on it. In that way, a hypervisor allows the simultaneous execution of multiple operating systems on a single physical server by creating virtual machines named Guest machines or Guest OSs while the physical server that a hypervisor is running on is called the Host machine. Each Guest machine runs its own workload. A hypervisor works as an orchestrator tool which manages, controls and monitors access to the VM and the underlying hardware resources. Through the hypervisor, system administrators can perform several actions like to create, stop, delete, and modify a virtual machine or even migrate to another host system, all the VMs are hosted upon the same host machine, meaning that all are sharing the same physical resources. The hypervisor is responsible for the proper allocation of those resources to each Guest machine in order to provide smooth operation and support. A hypervisor is also used as an extra security layer that protects from unauthorised access of the hardware resources from rogue applications or software running on a system.

A hypervisor has four main functionalities, emulation, isolation, allocation, and encapsulation. The hypervisor offers an emulated environment of a fully operational computing system. Each Guest operating system can't tell the difference between a virtual machine and a physical machine, nor can applications or other computers on a network. They have traditionally no idea or sense of the existence of other VMs or the virtualized and shared hardware infrastructure. From a VM point of view, it behaves exactly like a physical computer able to execute and host any kind of workload. A high importance functionality is isolation. The hypervisor must create and keep each VM's environment and processes running alongside them in isolation. Operations and activities executed by a VM must not interfere with or affect the operation of the rest of the VM's or host's operating system functionality. Furthermore, the hypervisor is a management tool that is able to control and allocate the amount of the virtual resources like, processing, memory, storage and networking resources.

There are two supported types of hypervisors, Type 1 also known as a Bare metal hypervisor and Type 2 also known as a hosted hypervisor. The former stands directly upon hardware having full control of the hardware infrastructure and resources while the latter is installed upon a hosted operating system running on a host machine as Figure 3.2 illustrates. The main differences between those two types of hypervisor are mainly noticed around the performance and security. A Bare metal type of hypervisor typically is faster offering higher efficiency because it has direct access to the hardware resources and does not need to pass through an operating system layer. Since there is not an OS layer or host applications to compete for access to hardware resources, which gives freedom for the hypervisor to take control of all the available hardware resources and properly allocate them to the VMs as needed. Furthermore, Type 1 hypervisors offer higher security since the absence of an operating system layer limits the surface for potential attacks and compromises. The most well-known and common implementations of Type 1 hypervisors are VMWare ESXi, Xen, Microsoft Hyper-V while major interest has been raised on ARM based hypervisors like Xvisor and BitVisor, which will be explained further in Section 6.3 and 6.4.

Before choosing a Cloud Service Provider for hosting their services and applications cloud consumers need to take under consideration which hypervisor each provider uses and pick wisely depending on their requirements. Amazon holds the most shares in the Cloud IaaS market, and recently announced the shift from a Type 2 hypervisor (KVM) to a customized version based on the open-source Xen. Google Cloud Platform, although new in the IaaS market has managed to gain an important share in the last two years. Google makes use of a customized version of the KVM hypervisor while Microsoft with the Azure cloud platform utilizes Hyper-V hypervisor [261, 262]. Although each of them has developed their own custom version of a hypervisor, the performance penalty that the virtualization layer introduces on a server system is unavoidable. Therefore, Cloud consumers should also consider Bare metal instances for intensive workloads that need dedicated performance which neither Type 1 nor 2 of hypervisors can offer.

3.1.2 KVM and Xen Overview

Xen is a member of the Type 1 hypervisor family also known as a Bare metal hypervisor that allows the creation of several Guest machines running on the same physical host machine. The Xen architecture is composed of a hypervisor layer, a specialised Guest operating system named Domain-0 or Dom0 and several Guest machines called Domain-U or DomU. Once it is booted up it creates the specialised environment Dom0 which is executed on the most privileged layer having direct access to the underlying hardware resources and the rest of the infrastructure. Dom0 like each DomU Guest machine has a dedicated vCPU and virtual memory while containing device drivers to I/O devices.

Xen offers two operation modes, para-virtualized (PV) and hardware-assisted virtualization (HVM) modes. PV mode requires the modification of the Guest operating system making it aware of running upon a virtualized environment. However, modern processors are enhanced with Virtualization Extensions (VT) like Intel VT and AMD-V offering the capability of executing a Guest operating system without any kernel modifications, achieving full virtualisation [57, 59, 75]. Xen uses QEMU to perform full hardware emulation including BIOS, IDE disk controller, VGA graphic adapter, USB controller, network adapter etc. for HVM Guests. CPU virtualization extensions are used to boost the performance of the emulation.

KVM (Kernel Virtual Machine), on the other hand, is a Type 2 hypervisor offering full virtualization meaning that Guest operating systems do not need to get modified in order to execute on a Guest machine environment. KVM like Xen is based on the same hardware-assisted virtualization extensions support (Intel-VT, AMD-V). KVM consists of a sophisticated, advanced kernel module, kvm.ko that provides the core functionality in addition to specific CPU modules like kvm-intel.ko or kvm-amd.ko. KVM does use a customized version of QEMU to perform hardware emulation while making use of para-virtualization to access I/O devices through VIRTIO. Figure 3.2 illustrates the architecture operational differences between Xen and KVM hypervisors as well as the flow of access to and from the hardware resources.

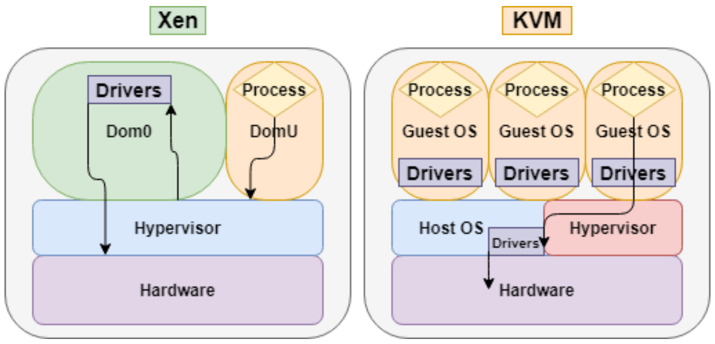


Figure 3-2 Xen and KVM hypervisor architecture

3.1.3 Challenges of Virtualization

Virtualization technology no doubt has changed the system architecture a lot and even more a data centre infrastructure. Through server and workload consolidation techniques, we can achieve higher CPU utilization and reduction of the overall operational and maintenance costs. However, virtualization introduces significant challenges related to delivering consistent performance, security and privacy among virtual machines and applications running on top of them. Although VT helps in maximizing utilization of hardware resources, a lot of research evaluations have shown that virtualization comes with shortcomings on the performance of a system that significantly affects the operation of the hosted applications [7, 259, 260].

Although in the last decade advanced enhancements on the hardware and software layer have reduced the performance overhead associated with virtualization, the performance gap with a native, non-virtualized system remains. Some of the major problems of virtualization right now are performance guarantees and consistency while privacy and security concerns that can appear based on the configuration. If you deal with requirements of milliseconds processing it is difficult to fulfil that in a virtualised environment, more if we talk about centralised Cloud facilities hosted thousands of miles away where the latency factor is added. Based on experimental measurements, the overhead in virtualized environments can reach up to 20% [7, 259].

Several performance evaluations have been conducted by many researchers looking at identifying the performance metrics that most affect the performance of a virtual system. Although conducting an evaluation of performance on cloud infrastructures is often not possible, many experimental studies have proven that the virtualization layer adds a performance barrier no matter what the hardware or software it runs upon. There are many aspects to take into consideration in order to address this performance degradation, like networking I/O requests, network throughput, CPU utilization, I/O disk speed, I/O peripheral requests etc. [260].

The degradation performance on a virtualized infrastructure can be noticed in different areas. As the schematic diagram illustrated in Figure 3.3 shows, three of the most important areas are the Hypervisor layer, the management of Guest resources and the network. Let us give a brief description of each area and how it actually affects the system's performance.

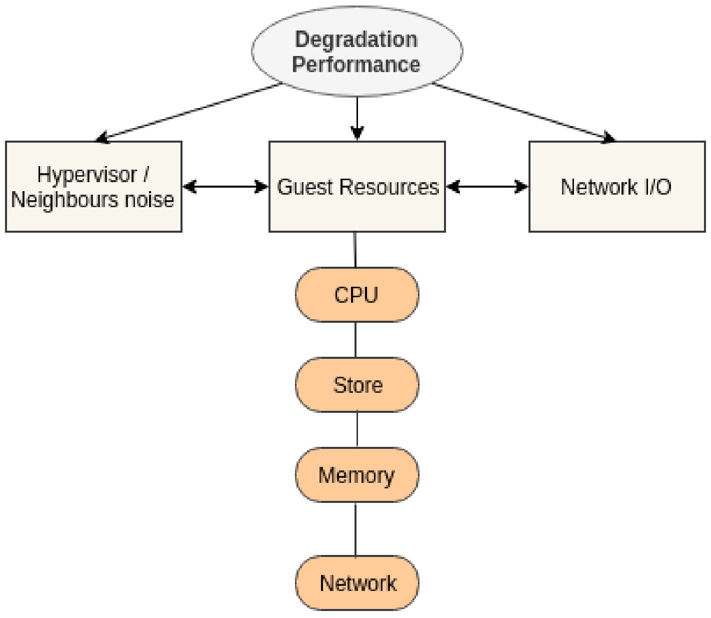


Figure 3-3 Virtualization performance degradation areas

A hypervisor plays the role of the orchestrator; a management software that controls the operation of the virtual machines. It is responsible for the proper allocation of the hardware resources while performing a variety of administrative tasks like creation, suspension, and deletion of a virtual machine. The hypervisor is responsible for performing CPU scheduling of the processes running on each virtual machine in order to provide a parallel execution, causing degradation of performance while caching and storing the state of a VM in memory or storage based on the infrastructure. Physical hardware resources are dynamically shared among Guest users by the hypervisor when and as is needed. In that way, virtualized shared environments suffer by the phenomenon called "noisy neighbour" where virtual machines compete with each other. A neighbour creates "noise" when a VM monopolizes most of the portion of the available hardware resources. In that way, resources are shared unequally, while it exhausts the hardware infrastructure at the expense of the other VMs resident on the same host system.

Multiple types of Kernel may be running on the same host server, sharing the same physical hardware resources as processors and memory. Each time that a process raises an exception request in order to access or perform a more privileged function where the user's privileges are not enough to do that, the hypervisor needs to save, preserve and restore back the state of the VM in/from a portion in memory. This is a very intensive and CPU consuming process.

However, the network performance degradation is considered as the most significant factor in virtualized systems. On a virtualised host, all the Guests share and make use of the same physical network interfaces sharing the available bandwidth and medium. Each VM has a dedicated link called a virtual port attached to an internal logical virtual switch which the hypervisor manages. The physical network interface is attached and is a member of that virtual switch. Through that virtual switch and after the performance of the switching and routing procedures that take place as if it is happening on actual hardware switch, the data finds the way out to the external world. The packet switching process executed by the hypervisor becomes a time-consuming process especially when the number of hosted VMs increases or the flow and rate of data is increased. Applications demand the establishment of a communication channel before the data exchange starts. However, multiple opened communication channels for a long period of time consume a significant portion of available bandwidth, increasing the network overhead and bottlenecks.

In computing, operating systems make use of the hardware-based hierarchical protection mechanism to protect data and applications from faults and unauthorised access to the hardware. That mechanism is implemented by CPU architectures providing different CPU execution modes. Intel architecture processors implement that mechanism through four protection rings, Ring 0 to 3 where Ring 0 is the level with the most privileges that is being used by the kernel while Ring 3 is being used by the user. On the other hand, on ARM processors these privilege levels are implemented through an exception level mechanism which we will analyse further in Section 5.4.2.Although the modern CPU processors are enforced with advanced virtualisation extensions which allow a Guest operating system to be executed on Ring 0, still privileged instructions need to be trapped on hardware and emulated, significantly reducing the processing performance while during that time a sequence of actions need to be performed by the hypervisor layer in order to keep track of those commands and virtual machine state, slowing down the overall system performance.

Modern virtualised topologies make use of network-based storage which means that each VM in order to perform a read/write request needs to raise a request to the hypervisor. Although hardware assisted virtualisation extensions improve the networking virtualisation performance, accessing multiple virtual machines on the same network infrastructure exhausts the available bandwidth while slowing the network throughput. The CPU sharing in combination with packet delay and packet loss on a virtualised shared infrastructure, although typically low, still can slow the network performance which can prove critical to transactional database connections.

Although virtualization offers centralized management, by merging all the services into a single system or fewer physical systems, this makes your system architecture vulnerable increasing the chances of a single point of failure. Server consolidation plans must be delivered followed by strong backup, failover, redundant systems. On Cloud data centres (CDCs), cloud providers offer auto-recovery and auto-failover mechanisms due to system failures, keeping data backups in different areas and regions, globally.

However, the most important consideration and challenge on virtualized environments is to secure and protect the virtualization infrastructure from security threats. Virtualised environments are exposed to many vulnerabilities and risks which an intruder can exploit, threatening the integrity and privacy of a system. Those vulnerabilities are at higher risk when it comes to a Cloud infrastructure scale.

The heart of a virtualized system is the hypervisor; it not only controls and manages the operation and state of each VM hosted on a host machine, but also handles the inter-communication between VMs and VMs to the physical hardware. On modern CPU architectures, a hypervisor is loaded on a dedicated mode at the highest privileges. Therefore, hypervisor-based attacks are considered the most dangerous and must be eliminated. A hypervisor is a massive software package, configured to perform many sophisticated tasks. The main responsibility of a hypervisor in this context is therefore to keep each VM isolated from the others and from unauthorised access to the host operating system or hardware direct. A compromised, affected hypervisor can perform several rogue actions. A malicious intruder is able to bypass the VMM layer and gain direct access to the underlying hardware or the operating system that is running on the host in the case of Type 2 hypervisor architecture. Poor isolation can also cause a series of malicious attacks like VM escape and VM hyper-jacking [77, 82]. Administrators must be very careful with offline or dormant VMs which remain provisioned offline. Those VMs may contain a security loophole and if those VMs return to an online state after a while, this creates a point of vulnerability until their patches and software updates are brought up to date.

Furthermore, VM-based attacks are another category where a rootkit and malware installed on a VM tries to perform and execute penetration attacks either to another VM or on external targets. Figure 3.4 names some common cases and type of attacks that administrators need to be aware of on virtualised systems [77, 82].

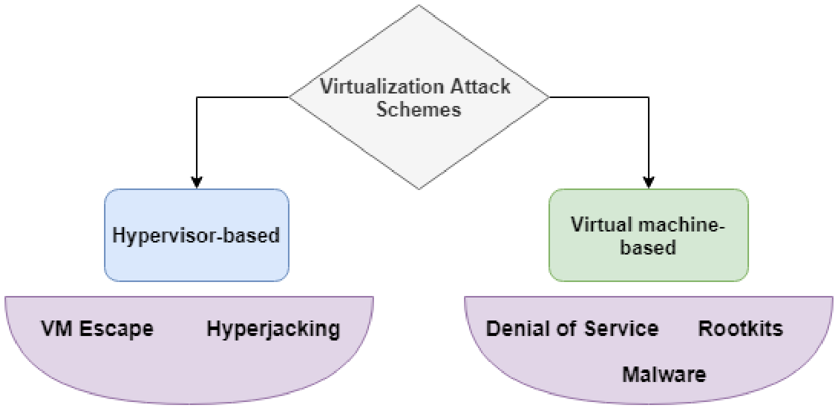


Figure 3-4 Virtualization threats considerations

3.2 The Advent of Bare metal Instances

Prior to the emergence of Cloud computing, businesses used to manage and administer their own data centre infrastructure hosted on their own facilities. The development of server virtualization remarkably reduced businesses' upfront costs and capital investments on space and hardware resources. However, hardware and infrastructure maintenance as well as administrative tasks were too costly and time-consuming processes, that were required a lot of effort from administrators while deadlines and time limiters putting a lot of pressure on them. With the advent of Cloud computing, many businesses decided to relocate and shift their workload to the Cloud by adopting a more flexible rental model.

The cloud is a multi-tenant ecosystem. Due to the existence of a virtualization layer, the physical hardware resources are shared among all users causing a lot of limitations mainly noted on the system and network performance like latency and inconsistency, while due to the shared hosting environment, a lot of challenges around security and privacy are raised. Therefore, virtualized cloud infrastructures are not intended, and do not work efficiently, for all kinds and types of customers and workloads. There are specialised workloads that are restricted by strict privacy and security agreements while demanding powerful computing performance. Even the use of containerization, an alternative form of operating system virtualization where multiple applications isolated from each other, sharing the same OS kernel, cannot host and provide a consistent, high-performance environment to intensive, specialised workloads [3]. Although containers are considered lightweight environments, easily scalable and highly portable solving a number of problems for software developers, performance and security issues and remain. Containers share the same OS, so isolation is considered lighter than VMs while resource allocation most of times is poor and limited. To be modular and easily portable, containers allocate some MB of memory and cores of a CPU [266]. Therefore, that kind of form of virtualization is not preferred for intensive, heavily resource-consuming applications.

The need to tackle those limitations and offer to customers a stable, consistent, durable, high performance and secure infrastructure led CSPs to include as part of their service a suite of Bare metal instances, also named as Metal as a Service (MaaS).

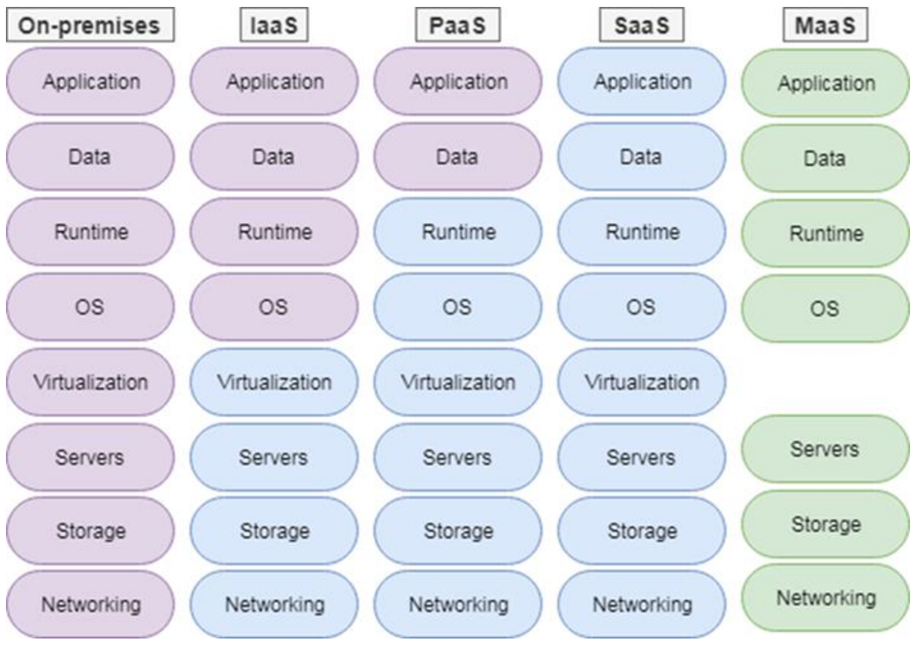


Figure 3-5 Metal as a Service part of the Cloud service suite

3.2.1 What are Bare metal instances

A Bare metal instance is nothing more than an actual physical server machine, a piece of hardware that lacks the virtualization layer as shown in Figure 3.5 above. It is specifically designed to eliminate the virtualization overhead delivering a native, pure system performance to the end-user.

Although the virtualization technology plays a core role in the cloud computing infrastructure, it is not necessary or a prerequisite of the normal operation of the Cloud. A statement of N.I.S.T [95], supported the idea that the operation of Cloud is not based on virtualization but undoubtedly enables and helps in the delivering of most of the Cloud computing benefits and characteristics. Cloud computing can stand by itself without the contribution of the VT layer following the same principles, maintaining the same characteristics offering the same and more benefits even through Bare metal instances.

Bare metal instances are servers dedicated to a single tenant meaning that the hardware resources belong and are consumed by a single and only customer. Three of the top Cloud Service Providers, Google, Amazon, and Microsoft as well many other IT vendors like IBM, Rackspace, Oracle etc. provide Bare metal instances as a service on demand. While more and more players join the Bare metal market. Surveys have shown that the global Bare metal market is expected to increase up to 11.40 billion US dollars during the period 2021-25; Bare metal absorbs and attracts various sectors such as IT and telecommunications, healthcare, government etc.

Cloud data centres facilities are composed by both virtualized and Bare metal servers. The core difference between VMs and Bare metal instances is the existence of the intermediate layer, known as hypervisor, which is installed on top of the hardware. The hypervisor manages, controls, monitors the sharing of physical resources among the VMs. In contrast, on Bare metal servers, users control and manage the entire system and networking infrastructure.

As with any Cloud service, similarly with Metal as a Service, users can register for a physical server machine on demand through a web-based portal with no need for human interaction. Then a new system joins the cluster of the data centre. Bare metal servers are highly customizable, and users can choose the level of resources and hardware specifications from a pool of resources in a similar way like on IaaS. MaaS follows the same pricing schema of a utility style billing model, where users pay for only what they use.

3.2.2 Bare metal instances as a hosting platform

Metal as a Service is a collection of raw hardware and resources. It’s a collection of nodes that customers can configure as desired, based on their needs, while they are free to install even a customized operating system or any similar software, unlike Cloud servers in which users must choose from a pre-configured list of operating systems that is supported by each Cloud Service Provider.

As with virtualized systems, so Bare metal instances are not the best choice for every business workload. For business operations, applications and services that do not need performance and security requirements, it's easier and often better to deploy a solution for a virtualized environment like IaaS. For business sectors like finance, healthcare, manufacturing and retail where they do make use of big data analytics and need to meet the strict requirements of privacy, security and high performance, Bare metal instances fit better. When it comes to performance, clearly Bare metal instances are the right choice. Bare metal is developed for specialised, sophisticated software and applications that due to specific and strict requirements are not Cloud friendly and not intended to be hosted and running on a virtualized server infrastructure.

There are many workloads that are not suitable or are intended to run on the Cloud and do not work efficiently on virtualized infrastructures. In such cases, MaaS is the best choice as an infrastructure hosting solution providing privacy and high performance while it retains Cloud computing characteristics. In some cases, software, and applications demand root access, so they can directly talk and communicate to the hardware. On virtualised systems, users can manage and see only the logical resources of the Guest machine. When software requires access to the hardware in order to operate and I/O call, the hypervisor is responsible for receiving and executing the command on behalf of the Guest user, increasing the latency and execution time of a process.

There are sophisticated software and applications that require root access or direct access to the hardware, those are not able to work with a hypervisor layer in the middle. Bare metal instances are however accessible through SSH and VPN connections while users can interact with devices via remote desktop access (RDP).

Additionally, the updated rules of GDPR raised a lot of challenges and concerns about the virtualized, shared Cloud environments and brought a lot of challenges about the privacy and policies followed by shared, multi-tenant virtualization environments such as public hosting Clouds. In the case of Bare metal instances, as single tenant systems, they offer flexibility on following any security and policy regulations that business defines.

Based on customer demands and requirements Bare metal instances can spin up in a couple of minutes to several hours if specific customizations need to take place. Although the provisioning and deployment process of a Bare metal instance takes longer than virtualized environments, it worth sacrificing that time to get a high quality, reliable and consistent system for your business's needs. However, recent studies work on boot time and resource reservation improvements are making a lot of progress [220,222].

G. Kominos and et al [119] performed an extensive performance evaluation on the three most usable hosting platforms of, Bare metal, virtual machines and containers, which are available to the end-user as a service on a Cloud environment. Using as performance metrics the CPU, memory, networking, and disk I/O show that Bare metal instances achieve and deliver higher performance overall compared to the other two platforms while VMs running upon a Type 1 hypervisor introduce a significant overhead. A Bare metal host delivers the best CPU performance while VM was worst since the scheduler needs to manage Guests' processes efficiency, increasing the overhead. Another significant observation was related to read and write tasks to and from memory. VMs show a significant degradation in performance during writing to memory slots while similar results were noticed on I/O requests to the drive. However, due to the lack of a virtualization layer and single-tenant structure, Bare metal instances lack resource utilization and better manageability.

3.2.3 Bare metal Instances for Micro Data Centres

In Chapter 2 we mentioned some Edge Computing deployment methods where Micro Data Centres could play an important role in their implementation. Gopika et al. [252] and Christian et al. [254] performed an investigation on several Edge Computing architecture models while they looked into additional technologies that could contribute and support such solutions, mostly focusing on virtualized technologies like virtual machines and containers that could boost the efficiency and scalability of systems plus the added advantage of a live migration process. However, applying Edge Computing as a solution to overcome the network performance limitations that exist between an IoT device and Cloud architecture while still utilizing virtualization technologies for system operation is a bit of an oxymoron. The virtualization layer increases the performance overhead, while sharing hardware and network resources among users and applications creates bandwidth limitations and latency since communications are still served through the same physical network interfaces. Therefore, we suggest and propose the utilization of Bare metal instances rather than virtualised systems. Combining Micro Data Centres with Bare metal instances could achieve the highest performance taking advantage of the easily distributed, multi-deployed, low latency architecture that Micro Data Centres provide and the benefit of native compute performance that Bare metal instances deliver.

As we explained in Section 2.4, a CSPs availability model is accomplished through the adoption of distinct zones composed of regions. Virtualization plays a key role in that, giving the flexibility of moving VMs around different racks and data centres. Taking a closer look as Figure 3.6 illustrates, let us visualize a zone as a rack where several VMs are sharing power and networking. Any failure on those resources would affect VMs' operations. Therefore, for high availability a service is hosted on zone A and a replica of that is hosted on zone B on a different rack [264]. Similarly, adopting the same principle and strategy on Edge Computing with Micro DCs and Bare metal instances, a service can be hosted on a Micro Data Centre on a dedicated server while keeping a replica of that service on a separate Micro Data Centre on a different Bare metal instance. We could even increase high availability by utilizing Bare metal instances of Embedded Development Boards like Raspberry Pi boards (RPIs) and BeagleBone by keeping and maintaining several replicas of services on the same Micro Data Centre as well as on a distinct one.

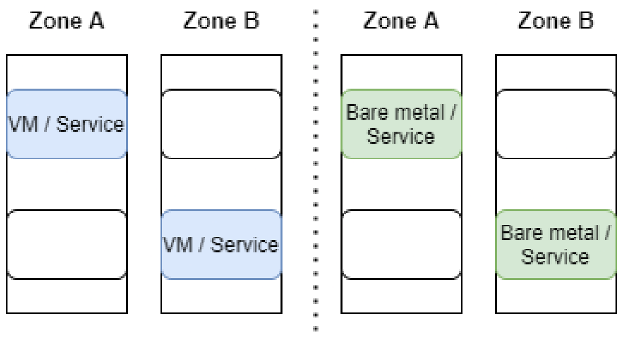


Figure 3-6 Availability sets of VMs and Bare metal instances

Currently there are no similar proposals in the literature looking into Bare metal solutions to enhance Micro Data Centre deployments on the edge. Figure 3.7 illustrates our proposed architecture and point of interest of an Edge Computing architecture implementation, where Micro Data Centres utilize Bare metal instances free of virtualization rather than virtualised systems. Specialised applications like vehicle automation, online gaming, big data analytics and more depend on fast responses and real-time processing. Bare metal instances could be the best hosting solution to meet the demands and requirements of such applications. The role of a Bare metal instance plays either a resource rich server or an embedded device like a Raspberry Pi board. Raspberry PIs have become very popular platforms on the edge due to low cost, small size, and power efficient characteristics. Several experiments show RPIs in the role of an edge node or edge gateway [258]. In cases like that, a single or a cluster of RPI boards can deliver a valuable, high-performance infrastructure. Motivated by that we explore RPIs playing the role of Bare metal hosting platforms in order to provision and enforce a Micro Data Centre infrastructure. Each PI member of a cluster can support and deliver a service, or an entire cluster of PIs can be allocated to provide a dedicated service to users. Scalability issues can easily be solved by deploying another cluster of PIs while during a faulty hardware replacement between PIs, it's an affordable and handy process.

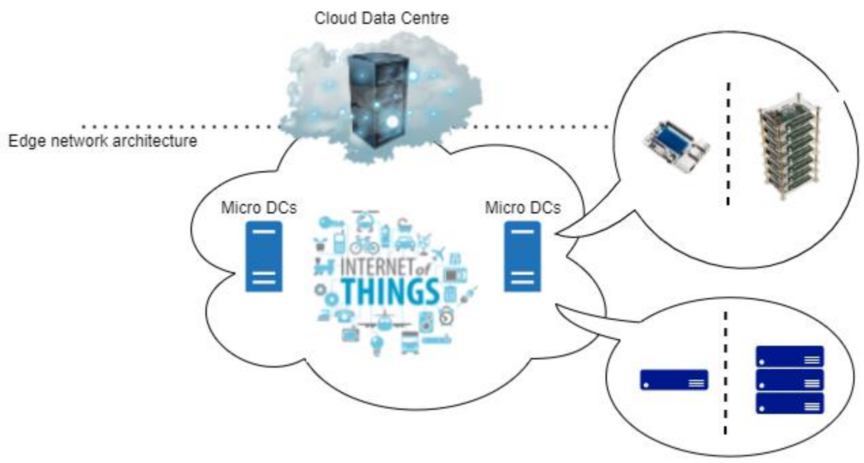


Figure 3-7 Bare metal Micro DCs edge architecture implementation

3.3 Summary

Virtualization technology has become an integral part of Cloud computing offering benefits such as scalability, system reliability and enforced fault tolerance while maximizing hardware utilization. However, all those benefits have an impact on the performance and system consistency. Several research papers show that the virtualization layer affects and slows down up to 20% of a system's performance. Therefore, Cloud Service Providers have included Bare metal instances in their service suite, free of virtualization.

Bare metal instances or Metal as a Service are dedicated servers to a single host, free of virtualization. However, utilization of Bare metal instances instead of virtual machines comes with some challenges on achieving workload balancing, availability, and fault tolerance, since live migration techniques are not available due to lack of a virtualization layer.

In the next chapter we will introduce the importance of live migration on both virtualized and Bare metal systems. We will discuss and analyse the implementation of live migration on VMs as well as the challenges and limitations on implementing live migration on Bare metal instances.

4 Live Migration

Modern Edge Computing architecture implementations allow installing small modular data centre infrastructure at the edge of a network known as Micro DCs. Edge Micro Data Centres enable the benefit of faster, real-time processing closer to the edge user through a distributed network infrastructure. Bringing computing and processing power closer to the IoT devices requires mechanisms that will offer high availability, redundancy, and fault tolerance, maintaining that attribute of mobility and modularity of a service avoiding any disruption or downtime. Live migration techniques can provide us with that insurance.

As Figure 4.1 demonstrates, edge Micro Data Centres are installed on multiple distributed locations on a network, gathering an enormous amount of digital information from a variety of resources that send and request data and other vital information.

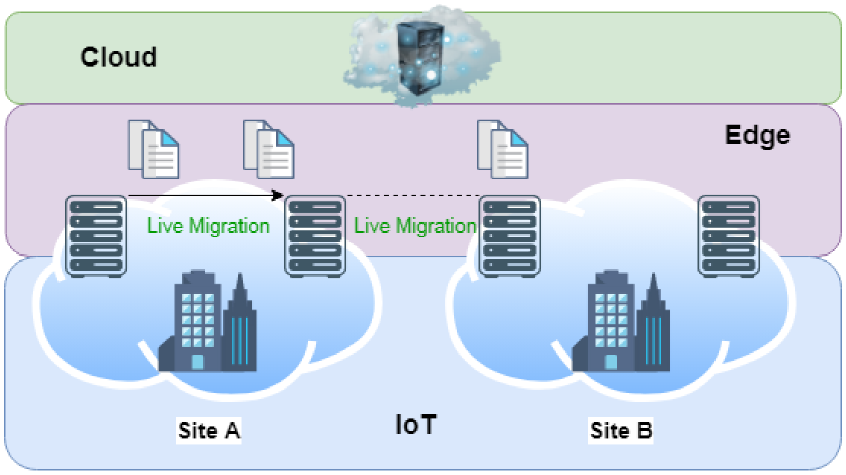


Figure 4-1 Live migration among micro DCs on an Edge network

Live migration is a vital tool in Edge Computing for many real-time, time-sensitive applications that have strict uptime service requirements like video streaming, online gaming, health care, e-commerce, smart traffic control systems and vehicle automation. Unexpected hardware and system failures or scheduled maintenance can cause disruption of service which in some cases can be catastrophic. Migration tools offer the great flexibility of transferring the state of a system to a backup system in a short time, avoiding long downtimes and service disruptions. They can either be used to migrate system data and state to another Micro Data Centre or even to a server located on a different physical location. In this way, we achieve high availability, redundancy, and fault tolerance. Smart traffic and vehicle automation applications interact in real-time with digital sensors installed in various locations collecting several measurements about objects, speed, distances and more. That information needs to get processed in order to prevent potential accidents. Disruption of a service like that can cause accidents, threatening human lives. Migration applications can help in factory automation by performing workload relocation offering higher Quality of Service (QoS) and location-awareness on devices and vehicles that need to be moving around.

4.1 Overview of Virtual Machine Live Migration

Virtual machine migration is a mandatory tool both to enterprise and Cloud infrastructures, permitting the relocation of a single or cluster of virtual machines from one physical host system to another, ideally with no service downtime. There are two techniques to perform a VM migration, the live and cold migration techniques. We focus only on live migration since cold migration demands the suspension of the Guest OS which leads to downtime of the upper services, something that is not desirable or acceptable on modern smart networks. Live migration on the other hand is referred to as a downtime-free technique where the Guest OS is relocated with no need for restarting the hosted applications. The Guest OS and users connected to it, do not need to be aware of the process that is taking place and no service interruption occurs.

Continuity of service, high availability and accessibility are critical aspects especially when referring to Cloud infrastructures. Proactive maintenance, fault tolerance, server and resource consolidation, Quality of Service and redundancy are some of the great benefits that a virtual machine migration process offers. Virtual machine live migration techniques remain one of the most interesting topics in the research community looking at performance and security improvements. Most of the virtualization vendors like VMware, Oracle, Microsoft, and Linux have integrated a VM migration tool as part of their virtualization suite.

On a virtualized system the hypervisor is the orchestration layer that manages, controls, and monitors a virtual machine live migration operation. The primary goal of live migration is to be completed in the shortest possible time, eliminating a service disruption or failure of service. The performance of a VM migration is measured in relation to the completion time of the following events:

- **Preparation time:** Describes the time that it takes to allocate the required resources at the destination system in order to ensure the availability of the resources.

- **Memory pages migration time:** The time that it takes to fully migrate all the memory pages from the source host to the destination.
- **Downtime:** The time that a VM is paused in order to complete and transfer the final portion of the state.
- **Total migration time:** Describes the overall time that it takes for the completion of a VM migration process from source to destination host. The time starts being measured by the initiation state until the point where the VM state is up and running at the destination VM.

Although during the last decades different technological approaches have been developed and suggested offering process migration and operating system migration methods [20, 250], however, none of them was fully able to offer an efficient, secure, and stable way of migration between hosts. Therefore, migration of virtual machine instances on virtualized systems has been adopted as the most popular way of migration that has been dominant over the last few years in the enterprise industry which still attracts the interest of the research community on suggesting improved performance techniques and implementations.

4.1.1 Virtual Machine Live Migration Components and Phases

A VM is a representation of a physical system having the same hardware resources such as processing power, memory, storage, networking interfaces and access to the available peripheral devices equivalent to a physical system. During a live migration process, the state of all those resources and open connections must be relocated at the destination system in order for a Guest OS to remain functional and resume at the same state as before the migration. The hypervisor layer is responsible for the orchestration and management of the process.

As has been well documented by many researchers in those years, during a VM live migration the following components need to get migrated from source to destination host [268]:

- CPU state
- Memory pages
- Motherboard settings
- Network TCP/IP connections
- Peripheral status

The CPU state describes the running state of the cores which have been allocated to the virtual machine. Memory migration is one of the most crucial parts and the second highest in size of data that need to be migrated. Memory size varies based on the needs and demands of each virtual machine typically being between 2GB and 64GB. Furthermore, during a VM live migration process, all the TCP/IP network connections must remain open and live before and after the completion of the process. In order to achieve something like that, both physical machines must be connected to the same logical networking subnet. The network migration processes rely on the ARP mechanism's unsolicited reply that announces the new location of the migrated VM. The VM should keep the initial IP address in order to achieve network redirection. Recent IP addressing solutions as discussed and suggested from Zap's modelling, [241] a dynamic DNS mechanism, name to IP address mapping translation can also be applied so VMs should be accessed using the same hostname after migration.

Modern virtualized architecture schemes utilize shared data storage techniques that keep storage data and VM states centralized making them easily and quickly accessible from the rest of the network. Shared storage techniques like SAN (Storage Array Network) and NAS (Network Attached Storage) also provide the advantage to avoid storage migration.

Planning and preparing a live VM migration before it takes place is an important task in order to perform a higher performance migration. The following steps must be performed:

- Selection of a requested VM or group of VMs that want to migrate.
- Allocate the required physical level resources at the destination host machine.
- Pre-copy most of memory pages to destination while the VM still running at the source machine
- Send CPU state.
- Transferring control to the destination system.

There are two strategies to fulfil a VM live migration process, the pre-copy and post-copy methods [1, 20]. The difference between those approaches is mainly focused on the way that they manage and treat memory pages transformation from source to the destination system.

- Pre-copy

Handles the transformation of memory pages from source to destination in an iterative way without affecting the operation of running the VM on the source host. The number of memory pages that have been modified by the source user during that process, called dirty pages, are transferred to the destination machine with the next iteration. While the rate of the dirty pages remains too high then the total migration time is increased. The pre-copy technique includes a short stop and copy phase where the operation of the VM is suspended on the source host, and the rest of the memory pages as well as the critical CPU state is transferred to the destination host where the VM continues to run.

- Post-copy

At the beginning there is a short time period of stop and copy, migrating the essential kernel state in order for the VM be able to be executed and start running at the destination host. When the VM accesses a memory page portion that is not available to the destination host, it raises a page fault signal, and the memory page is pulled by the source host on demand. The post-copy approach achieves a remarkably short downtime of service but longer total migration time.

Memory pages can be transferred by using three main techniques, push phase, stop-and-copy and pull phase. It's very common for a combination of two of them to be used in order to achieve the best performance in the shortest overall migration time.

- Push phase: As its name declares, while the VM is still running on the source host, memory pages are pushed to the destination machine. Dirtied pages must be resent in order to keep consistency.
- Stop and copy phase: The running VM at the source host needs to be suspended in order to transfer all the memory pages to the destination host where resumes its operation. Although this technique is straightforward and simple the main drawback is that VMs suffer and are affected by an unacceptable downtime that cause interruption of the service hosted inside a VM.
- Pull phase: The VM starts the operation on the destination machine and when there is a demand to access a memory page, that page is pulled from source to destination.

The following Figure 4.2 gives a comparison between the stages that a system passes through during a pull and push phase respectively.

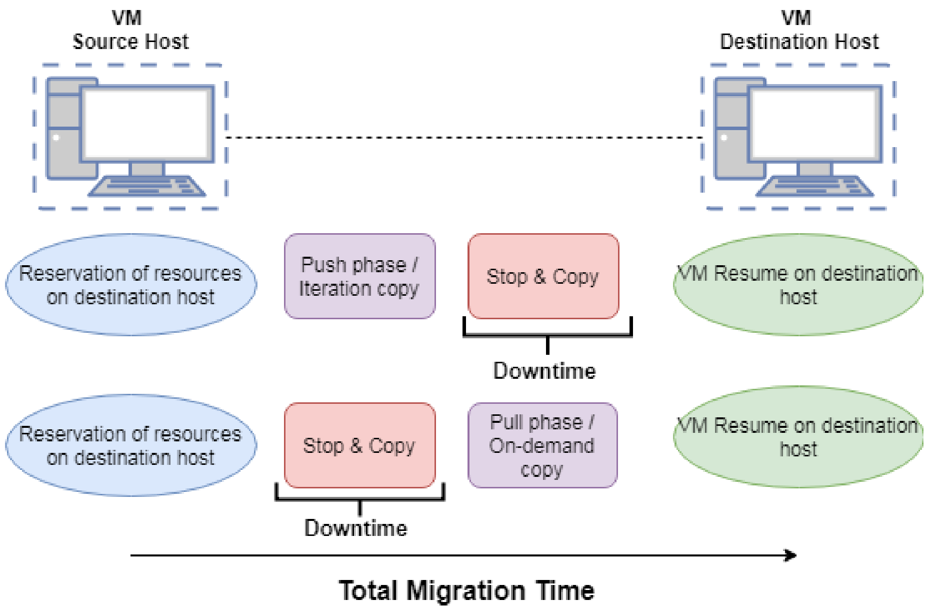

VM
Source Host
VM
Destination Host
Reservation of resources on destination host
Push phase / Iteration copy
Stop & Copy
Downtime
VM Resume on destination host
Reservation of resources on destination host
Stop & Copy
Downtime
Pull phase / On-demand copy
VM Resume on destination host
Total Migration Time

Figure 4-2 Pre-copy vs Post-copy memory migration process

Migration of memory contents is one of the most crucial and highly important parts during a VM live migration process. Nowadays, particularly on CDCs, memory size has capacity starting from 4GB reaching up to 128 GB or even higher, in some cases dealing with sensitive, volatile data which can be lost in case of a failure.

There are different memory modules that need to be migrated from source to destination listed as following:

- VM configured memory: A VM like a physical system needs access to physical memory for loading and storing process's essential code. This amount of memory is allocated by the hypervisor and works as a physical memory to a VM.
- VM used memory: A portion of frequently accessed and modified memory pages, part of the configured memory module is the VM used memory module.
- Application requested memory: Each process is an application that demands access to a specific amount of memory VM for their smooth operation and functionality. However, it is not necessary that this memory is hosted entirely in physical memory as may reside on a disk if the available memory space is not sufficient. Swapping methods takes effect and operation in this case.
- Application actively dirtied memory refers to the memory space where active applications and processes are loaded into; a VM frequently accessing those memory pages, modifying them regularly,

minimizing in that way the use of swapping disks techniques which extends the operation of the migration process.

Performing a live migration in an efficient manner based on the ability to transfer the entire state of a source system to a destination over a network is almost impossible or challenging to be performed in traditional computing environments. Early studies and approaches to that area were related to process migration techniques. However, process migration has several restrictions and many dependencies on the OS while demanding the creation of dedicated user spaces or domain groups in order to achieve a level of isolation from the rest of the system.

However, in virtualized infrastructures thanks to the abstraction layer, migration of a group or single entity of a VM from one host to another is a simple and easy task. Major advantages of such an implementation are that VMs can be easily relocated, VM instances can scale up or down independently to storage state. There is a centralised control and monitor management solution that requires backup and redundant architecture designs. Network bandwidth and latency are major factors that network administrators need to consider in order to not impact or affect VMs' performance.

4.2 Live Migration of Bare metal instances on the Edge

Bare metal Cloud preserves the cloud-based characteristics of flexibility and scalability offering fast and automated, on-demand server provisioning and pay-as-you-go billing model while offering higher computing and processing performance with finer security. Since there is no resource sharing among users, Bare metal cloud eliminates the effect of "noisy neighbours" where users compete for access to a greater share of resources. Furthermore, with no virtualization layer, no hypervisor overhead and sharing of underlying resources are in place.

Bare metal cloud is a great fit to mission-critical applications that require optimum computational power where virtualisation overhead is not acceptable and governed by strict security and privacy policies. Many applications require access and modifications on the hardware or specific attachments in order to configure a high-performance SAN or NAS integration.

Big data is one of the main areas in IT in which Bare metal instances could be a perfect fit. With that massive growth of IoT devices, an enormous amount of data is collected and stored to the Cloud for further processing and analysis. Big data analytics formulas demand the high compute performance that

Bare metal can provide as well as the flexibility to spin up an additional server or shut it down when needed while paying only for the time of use. E-commerce is another field where Bare metal server provisioning can be valuable. There are times in the year when retail websites have the need for more resources. That sporadic demand of resources leads to the "noisy neighbour" issue that virtualised systems deal with, however, on Bare metal systems, users can fully utilize the maximum hardware resources when requested without dealing with the issue of potential inadequacy of resources.

Micro Data Centres improve network performance while Bare metal clouds offer dedicated, native hardware performance, free of virtualization overhead. A potential combination of both technologies could deliver an ideal hosting environment for all kinds of workloads, from those that need network consistency to those which require pure, native processing power. Although Micro Data Centres can host Bare metal instances, redundancy, fault tolerance and high availability on those become extremely challenging.

Designing and deploying an Edge Computing architecture in order to fit to a specific workload becomes challenging. Several factors and aspects affect the choice of the right model and infrastructure components that form your edge network. Relocating computing and processing power closer to the edge demands careful planning of the number of edge nodes needed in order to cover the needs of the varied IoT connected devices. QoS and service continuity become high priority so devices can remain connected even on the move. Workload distribution, service load balancing and fault tolerance become the key for the maintenance of a healthy edge network. Live migration schemes and strategies becomes an interesting topic to the research community.

Rohit et al. [255] proposed a container-based live migration algorithm upon Linux systems called LIMOCE which can find applications on ARM based edge clusters. The algorithm is based on the Checkpoint/restore in user space functionality that Linux provide exposed via API and managed by a centralized management tool and hidden from users. However, in order to make it work Linux kernel modifications are required. Paolo et al. [256] introduces a mobile Edge Computing (MEC) architecture consisting of three-layers for the support of mobile devices that lack computing and processing power on the edge. Edge devices that cannot perform intense computation tasks, are connected to an MEC middleware layer which performs those tasks on their behalf. That middleware layer support consists of two primary components, the Elijah platform and the Server Manager that manages the connection and communication of edge devices with the Elijah platform. The MEC layer provides a virtualised function migration in order to provide high availability, and service continuity based on a VM / container architecture. Working in the same field, Shangguang et al. [257] conducted a survey on the available service migration strategies and solutions that could find implementation on MEC architectures mentioning once again the three-layer architecture as introduced by Paolo et al. [256].

Performing a VM live migration has been a straightforward process. Essentially, VM's state resides in the memory of the host machine. Once that memory portion has been copied ahead to the destination host, the host can access its virtual disk files through shared storage. In contrast, on Bare metal instances, the state exists in the physical hardware components. Extraction and insertion of the hardware state through a software-based layer is very challenging. Specialised software is needed with access to both hardware and application layer. Software developers need to develop their own program to accomplish a task like that. Performing a Bare metal live migration without the assistance of a hypervisor management layer is extremely challenging.

A Bare metal live migration framework needs to meet some requirements and several assumptions. At first, a live migration framework on Bare metal instances needs to be independent of the operating system because users should be free to select the OS of their choice without limitations. Furthermore, users should not be responsible for taking actions in cases of unexpected failures. That should be an administrator's responsibility whether this is subject to industrial or Cloud environments. Additionally, one of the greatest advantages of a Bare metal instance is that it is fully customizable, meaning that users can install and configure entirely based on their preferences. So, changes or modifications at the OS level could cause conflicts to a software-based developed live migration scheme. Additionally, source and target Bare metal instances should have the same hardware components and characteristics like motherboard, networking interfaces, memory capacity and CPU architecture. It's very common, on data centre infrastructure to find clusters of racks of servers with identical hardware specifications. So, finding an available Bare metal instance is not difficult.

4.3 Related Work on Bare metal Live Migration Schemes

During the years different approaches have been proposed on a live migration such as OS live migration [20] and process migration [250] schemes. Although OS live migration allows the migration without the need for source and destination to have identical hardware specifications or the existence of a virtualization layer, modifications are still required on the operating system. Additionally, process migration as introduced by the Zap model [241], allows the migration of a process among source and destination OSs. However, this kind of migration model is not preferred due to a series of dependencies that a process maintains with other processes running on the same system and connections with the OS itself.

Bare metal live migration is a fresh topic in the academic and research community with few practical implementation schemes in production. Recent research experiments like Fukai's et.al. [37, 38] introduce an advanced, thin hypervisor layer able to perform a live migration on Bare metal instances on single and multi-core processors based on x86 architecture called BitVisor [37] and BLMVisor [38] respectively. Both hypervisor schemes utilize the same core hypervisor governed by the same underlying operations. The great advantage of that approach is that it does not emulate or virtualize the hardware resources, instead, it exposes them directly to the Guest OS, as Figure 4.3 illustrates. Furthermore, the hypervisor's footprint is remarkably smaller than traditional hypervisor layers. The hypervisor supports the execution of a single and only Guest OS which simplifies its functionalities like CPU scheduling and memory mappings operations and it only acts during the live migration process while remaining almost inactive during the normal execution of the system therefore the hypervisor overhead remains at negligible levels. Moreover, in order to avoid any performance degradation when the live migration process takes place, a dedicated network connection is used between the source and destination transferring the Guest OS state across the network.

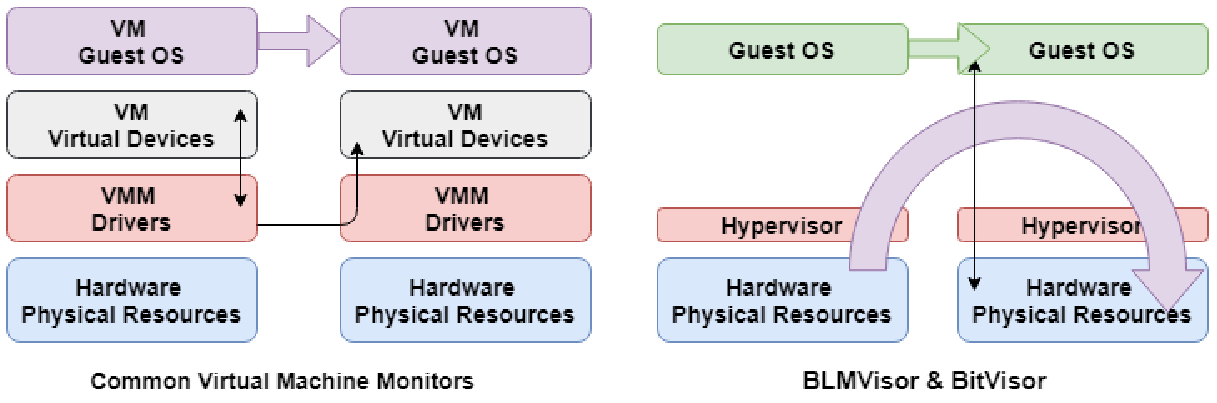


Figure 4-3 BLMVisor and BitVisor architecture

Modern x86 processors are enforced with hardware assisted virtualization extensions offering a full virtualization solution of an unmodified Guest OS. Both AMD and Intel vendors have introduced their own suites of hardware assisted virtualization extensions under the brand names of AMD-V and Intel-VT respectively. Currently implementations of BitVisor and BLMVisor support only Intel processors taking advantage of the hardware-assisted virtualization functions. Intel processors use a specialized data structure called virtual machine control structure (VMCS) capable of storing, maintaining, and restoring CPU state. VMCS mechanisms save both host and Guest environment state in memory. A hypervisor can then easily read VMCS structure on a source machine and write the states into VMCS at a destination machine performing a CPU state migration process automatically. Most of the CPU state resides inside a VMCS structure however some general-purpose registers and more system specific registers can be obtained through the software.

Memory migration is a research field that has been studied very well many years ago. During the migration process the hypervisor starts transferring all the available memory pages in the background, known as "pre-copy" process as explained above in Section 4.1.1 while those most recently accessible by the operating system pages, which have undergone some modifications, are sent following an iterative model. During the migration process, the operating system accesses some of the memory pages. Modified pages need to be resent from source to destination until a small portion of them remain at the source system. Then the hypervisor stops the Guest OS operation at source and transfers the remaining number of dirty pages at the destination known as "stop-and-copy" process.

Another challenging part of a live migration process is the preservation of internal and I/O devices state. The migration of internal state of the various physical components such as network cards, system's timers and the state of the supported interrupt handlers cannot be performed through a software layer but a more privileged application is required with higher permissions and access to the underlying hardware. In order to capture and set those internal states from source to destination system, the hypervisor monitors and reconstruct the physical state based on device specifications.

Modern platform architectures allow programmed input-output (PIO) or memory mapped input-output (MMIO) functions access to device registers through memory pages. In general, there are two approaches for a processor to communicate with a peripheral device: Port-mapped I/O and Memory-mapped I/O. The PIO method makes use of special CPU instructions to talk to an I/O device. In contrast, the MMIO method does not require special CPU instructions. Each device register is assigned to a memory address space specific to each platform which can be access via simple CPU load/store instructions for access to I/O devices such as GPIO, timers and so on. Each time a Guest OS requires access to those specific memory addresses, then a paged fault request is raised to the hypervisor which can handle and monitor the access seamlessly. The device registers are classified into three categories as readable, write-only, and internal registers. Device state can be described by two sub-states, the configuration state and status state. The configuration state contains operational configuration values of the device which are modified by the OS while the status state describes the running state of the device at each time. To give an example, NIC configuration state refers to the transmission and reception bandwidth while the status state can be the actual status of the interface during a device reset operation.

There are also two categories of unreadable states. One refers to write only registers that software can modify or set but cannot read them, and the internal registers which are updated by the device itself. No existing software can write or read those states. The way that BLMVisor and BitVisor preserve those states is by continuously monitoring the accessing on write I/O registers that are mapped to I/O memory. When the Guest OS sends a request to access an I/O address, the hypervisor intercepts those values and stores them into memory locations. During the migration, the hypervisor captures and transmits the most recently updated value to the destination.

Like write-only registers, there are a group of read-only registers. In order to set unwritable states at the destination, the hypervisor also needs to reproduce and reconstruct the state, through dummy data until the state changes to the desired state.

However, both the BLMVisor and BitVisor implementations face some limitations not only on hardware specifications of the devices but also on the supported CPU architecture. Currently, BitVisor and BLMVisor supports the x86 Intel architecture while supporting device specific migration. The rise of ARM processor architectures in embedded technologies and the introduction of chips targeted to data centre infrastructures introduce a new candidate as a hosting platform especially on Bare metal edge instances. The small CPU footprint in relation to the low energy consumption and low price on the market make it an alternative solution for Micro Data Centres. Due to high customizability, chip vendors can adjust ARM processors based on workload needs. Therefore, we aim to investigate a practical implementation for a Bare metal live migration on ARM based instances that can take place on a Micro Data Centre infrastructure.

4.4 Summary

Live migration is a valuable tool that helps in the reliability of an environment. Due to live migration, Cloud environments are able to offer and provide service continuity and high availability. Moreover, live migration allows the implementation of workload consolidation techniques in order to perform higher Quality of Service, redundancy, fault tolerance and load balancing among systems.

However, live migration is mostly available on virtualized systems. With the increasing use of Bare metal instances, maintaining these live migration benefits becomes extremely challenging since the virtualization layer is eliminated. Working on that area, Fukai first introduced a lightweight hypervisor scheme capable of performing a live migration of a system's state among Bare metal instances on x86 architecture. However, Bare metal live migration is based on a lot of dependencies that makes it difficult to be portable for a wide range of platforms and architectures. Recently, ARM chips have focused on Edge to Cloud datacentre infrastructures and have attracted a lot of interest from many vendors. Currently, there is no such implementation on ARM based systems.

In the next chapter we discuss the adoption of ARM architecture in the data centre infrastructure and the benefits of ARM processors utilization on Edge Computing.

5 ARM Architecture for Virtualisation and Edge Computing

5.1 Overview of the ARM Architecture

In computing, there are two architectural designs of processors, one is CISC (Complex Instruction Set Computing) and the other is RISC (Reduced Instruction Set Computing). The key difference between the two architectures is focused on the instruction set architecture (ISA) which describes a set of commands that a processor can handle and execute such as addressing modes, data types, registers, interrupts, and exceptions handling. The CISC approach, as its name implies consists of more complex instructions that a processor needs to execute. A single instruction may be composed of several additional operations that need to be performed. Although the CISC architecture minimizes the number of instructions per program, it spends a higher number of cycles per instruction. An example of the CISC architecture is the x86 instruction set. Both Intel and AMD make use of the x86 processor' architecture. On the other hand, RISC does the opposite, reduces the number of cycles per instruction using a pipeline instruction scheme. Several numbers of instructions are executed in a pipelined manner, where each instruction is executed within a single clock cycle. ARM instruction set architecture which stands for Advanced RISC Machine, is a member of the RISC family. ARM instruction set architecture design completes a task in a few lines of code by using simple commands. Simple instructions result in the need for fewer transistors, resulting in more chip space, meaning ARM processors have a smaller chip footprint. Power consumption rates and efficiency can be one of the most important criteria and factors on embedded devices And ARM processors offer a low-power design since fewer transistors are being used and relatively lower processing speed is achieved.

The introduction and adoption of ARM processors has great economic impacts on organizations, significantly reducing on-premise and cloud infrastructure and operation costs while delivering great efficiency. In embedded and mobile systems, power efficiency is by far the most important aspect compared to performance. Heat and power consumption become the key factors on a mobile product design. X86 architecture CPUs are well-known for their high performance making them suitable on large desktop and laptop processors. ARM processors in contrast achieve high energy efficiency due to the RISC instruction set architecture since the internal architecture is much simpler with fewer types of instructions. That leads to energy savings and less overhead.

ARM is the dominant RISC microprocessor architecture finding massive implementation in almost all the world's devices covering a wide range of technologies. During the last decade, ARM architecture finds utilization on 95 per cent of the current mobile and consumer devices such as smartphones, tablets and wearables landscape, while it keeps growing in the market networking systems infrastructure like routers, switches and firewalls and on modern embedded technologies such as automotive and a wide range of smart devices in industrial and utilities, holding a sizeable market share. Given of the range of the available interactive wearables devices in the market, ARM provides a wide range and kind of processors covering market's wide range of applications. For example, Moto 360 smartwatches are Cortex-A7-base while fitness tracking Gear Fit is ARM Cortex – M4 CPU based.

What makes the ARM architecture stand out from the x86 architecture is that it offers chip-makers the option to design their own processors. ARM does not actually make or produce processors, instead, it designs a CPU's architecture which becomes available through a licensing-based model. In simple terms, it sells licenses for its "instruction sets" architecture which determines how processors handle commands and internal instructions, how the input and output data should be formatted, how the processor interacts with RAM and other peripheral components and much more. An instruction set architecture is a blueprint for how all the parts of a CPU will operate. Through that license-based model, chip vendors like Samsung, Qualcomm, AMD, Broadcom, Amazon, Huawei or Apple have the freedom to customize their own CPUs and systems-on-chips (SoCs).

Another great feature that ARM processors first introduced into the semiconductor industry was heterogeneous computing architecture. Heterogeneous computing is a system design that enables the ability to combine and host more than one kind of processor into a single chip covering graphics processing unit, application-specific integrated circuits as well as a modern type of neural processing unit that's specifically designed for machine learning. Although the heterogeneous chip architecture has been adopted for years on mobile devices, datacentres are mostly dominated by traditional complex chip architectures that focus on optimised performance. However, the rapid growth of IoT and Edge Computing will change those stereotypes in the data centre industry.

ARM refers to heterogeneous architecture as *big.LITTLE*, a feature that is available on the modern Cortex-A series processors. In the case of *a big.LITTLE* chip, one of the cores will be low power while the other is much more powerful core. Based on the chip utilization, if intensive tasks are taking place, then the compiler communicates with the chip to make use of the powerful core while during basic, low levels of processing the lower-power core will run and the more powerful turns off. In that way, up to 75% power-saving is achieved.

5.2 ARM into the Cloud to Edge server infrastructure

The emergence of the IoE era driven by a diversity of sophisticated, advanced smart technologies requires a flexible architecture tailored to meet modern applications needs and demands in process and computing infrastructure. Although for decades data centres have been dominated by complex instruction set chip architectures which perform tasks at high speed, during the last few years, ARM is working up to make its entrance in data centre infrastructure. In 2020, ARM has fulfilled this milestone, gaining some market shares on server infrastructure from Intel, the dominant force for years and now reaching up to 25%. ARM released its first chip designs dedicated to data centre infrastructures, introducing the Neoverse family line with E1 and N1 cores. ARM through the Neoverse architecture cores it is trying to build a new ecosystem, offering a high-efficiency architecture, giving a solution on customer's needs on modern Cloud to Edge infrastructures. Neoverse E1 and N1 CPU processors offer high performance while increasing the energy efficiency up to 30% higher than previous generation processors of the Cortex family. Those modern CPUs aim to address the requirements for specific applications in the cloud-to-edge infrastructure.

N1 aims to find implementation more at the edge or within a data centre. Those processors aim to offer higher computing performance, 60% and faster processing speed, expecting to boost by 60% over the previous generation processors. Meant for the data centre or the edge while it can scale from 4 up to 128 cores, it also promises 2.5x more performance on cloud-based workloads, along with 30 percent power efficiency improvements. On the other hand, E1 delivers higher throughput performance than N1 processors. N1 cores mainly target fast data processing while E1 cores target fast data transferring, a combination that could be a great fit for servers located at the edge of a network where billions of IoT devices are connected.

ARM Neoverse cores could also be great candidates for Bare metal Micro Data Centres since they combine both performance and high throughput factors, very important to modern Edge Computing solutions where data needs to be analysed quickly and passed through end devices over a network.

One of the reasons that ARM processors dominate in the mobile and embedded industry, having application in a wide range of embedded and wearable devices, is that the variety of ARM families cover most of the market's needs. Besides the Neoverse family, the ARM family palette contains Ethos and Cortex families.

We are going to provide a highlighted general overview of the Cortex family, areas of applications and main characteristics. The ARM cortex family is divided into three sub-families named Cortex-A, Cortex-R and Cortex-M. Through those three types of processors, ARM covers a range of applications' needs perfectly, giving the ability to chip designers to choose the core that best fits without setting restrictions and forcing them into a one-fits-all implementation.

ARM in 2011 announced the release of ARMv8 core series introducing the support of x64-bit operating systems. ARMv8 is the successor of ARMv7 enhanced with new features and extensions of the ISA and the hardware, like the ability for both 32-bit and 64-bit core execution states, embedded on hardware security and cryptographic extensions; also, the support of hardware-assisted virtualization enables the ability to run multiple unmodified Guest OS. The scope of ARMv8 was to insert into the server market and data centre infrastructures offering high performance low-cost small size chips comparable to the market leaders for years of the chip on server market, Intel and AMD.

In March 2021 ARM announced after 10 years a new architecture and successor of v8, the ARMv9 as a response to the high demand of specialized processing and computing. ARMv9 inherits the same ISA and exception levels from v8 having full backwards compatibility. The new add-ons and extensions on ARMv9 are mostly targeted around the pillars of AI, IoTs, 5G and security. ARM in collaboration with Fujitsu developed the second version of Scalable Vector Extension (SVEv2) technology, a technology that enhances the processing of 5G systems and the capability of running ML workloads while increasing the overall performance at 30% from the previous generation. In the field of security ARMv9 introduces the Confidential Compute Architecture (CCA) which protects sensitive data and portions of code from being modified and accessed on hardware-based protections even from other privileged applications.

In the last few years, both Google and AWS, two of the top Cloud Providers, announced the discussions of the adoption of ARM servers on data centre infrastructure. Already three of the biggest enterprise leaders in business and the IT world, Google, Amazon, and Facebook have announced their interest to adopt ARM based servers into data centre infrastructures, reducing the need and adhesion to x86 chip architectures that Intel and AMD dominate.

The ARM family of processors comes in different types covering a wide architecture of platforms. Application developers and programmers can choose from three main suites of processor profiles A, R, and M, based on the needs and demands. Due to rapid and aggressive development of ARM processors in the market, it's difficult to develop a generalised implementation covering all those kinds of architecture. Therefore, we selected to work on the latest ARM family to server infrastructures, Cortex-A series eight (ARMv8). Unfortunately, at the time that the thesis was being written ARMv9 was not available so we could not develop our design and implementation on the latest version.

An architecture profile is the one that we mainly focused on and used on server infrastructures, capable of supporting a full OS. The following list gives a brief description of each profile [7, 9]:

- Cortex-A family processors refers to high-performance application processors finding implementation in mobile devices, networking infrastructure, in automotive and a wide range of Linux and Android consumer devices such as tablets, raspberries etc., that contain memory and MMU Translation system. The Cortex-A series of processors cover this market's demands supporting both 32bit and 64bit architectures.
- Cortex-R family processors refers to real-time processors offering high-performance processing to embedded devices that depend on and demand reliability, fault-tolerance, and real-time processing such as autonomous systems.
- Cortex-M family processors refers to microcontrollers, finding application on smart home devices like smart lighting and motion sensors. The main benefits of M family processors are low power consumption, low-latency, and highly deterministic operations. The latest version of M-profile processors puts its emphasis on providing high machine-learning performance (Cortex-M55). Those types of platforms support a different exception handling design and only support a simpler, smaller instruction set named Thumb instead of an ARM instruction set. The key difference of that new core architecture is the enhancement with the neural processing unit (NPU) which delivers high ML performance, mission-critical to utilize the potentials of AI and IoT.

The technological trend and demand for adopting virtualization nowadays and making it available on smartphones as well technological vendors that aim to reduce financial investments, reduce space requirements and at the same time the electric and power consumption rates move their interest to on-premises ARM server implementation and moreover on CDCs. Google announced interest in introducing powerful ARM servers on its CDCs. Google and AWS have announced the relationship with Qualcomm for the adoption and replacement of x86 processors with ARM servers in data centre infrastructures[267].

ARM, following the technological needs and demands for multi-boot environments, introduced virtualization with the ARMv7 family where Cortex-A15 and Cortex-7 processors introduced support for hardware-assisted virtualization. Further enhancement of virtualization specifications and features were added later in ARMv8 and keep evolving until today. In production environments like CDCs servers make use of the Cortex-A family. ARM virtualization extensions are greatly different from virtualization on x86 architecture.

Additional studies try to explore the capabilities and advantages of ARM based Bare metal instances mostly on RPI single boards for the support of big data analytics and machine learning tasks in an effort to decongest the heavy workload from IoT products to the Cloud [221].

5.3 ARM TrustZone execution environment features

ARM Cortex-A family processors are enforced with TrustZone [6, 9] security extensions, a set of hardware-based technological security features that divides the hardware infrastructure into two separated, isolated worlds as illustrated in Figure 5.1, a secure and a normal world, also referred to as modes. TrustZone functionality has a similar concept to trusted platform-modules (TPM) on x86 platforms. On ARM TrustZone extensions create a trusted execution environment (TEE) where a software that is running in the secure mode has a completely different view of the hardware infrastructure from one running in non-secure mode. Secure world offers security, confidentiality and integrity allowing only trustful software and applications to run into it. TEE is a good solution for storage and maintenance of encryption keys or biometric credentials that need to be used for verification purposes by the operating system or other applications. This means that even when a malicious code affects the system, and an intruder obtains root privileges to the normal OS world, it cannot jump or access the secure world.

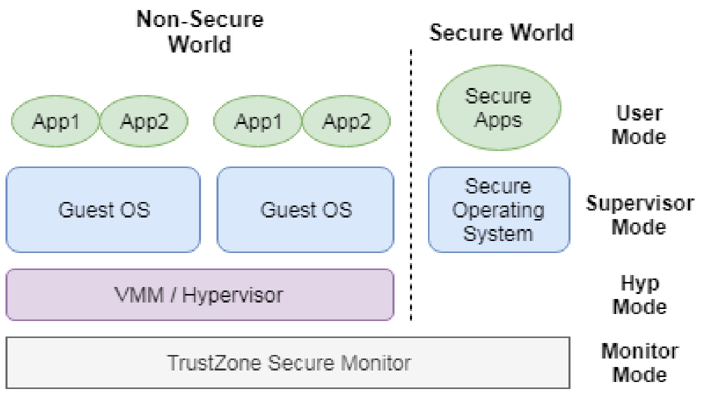


Figure 5-1 ARMv8 processors supported CPU modes

TrustZone is a hardware embedded, programmable chip that enables peripheral and memory protection. Memory is also divided into secure and non-secure regions. When software is running in a non-secure world, a non-secure memory flag is enabled that permits only access to normal world memory space. Only a trusted software runs on a TEE having access to the processor, peripherals and memory while it is completely isolated from applications and the operating system running outside of it. ARM Trustzone does this by creating a partition and isolation of the hardware components such as busses, peripherals, memory, interrupts so that the application code does not have access to that protected, restricted portion of resources.

On ARM processors with TrustZone Security extension features, each core can be executed on non-secure and secure world where the context switch between those two worlds is managed by the dedicated CPU mode, named Monitor mode as Figure 5.2 illustrates. The transition between the Secure and Non-Secure world happens through a dedicated instruction named Secure Monitor Call (SMC). Once this instruction is called, the CPU enters into the most privileged dedicated mode, the monitor mode that gives access to all hardware including the restricted peripherals and memory portions. In that way, a trusted OS, or application runs in the TEE. Trusted applications that reside in a TEE have access into the device's peripherals, interrupts and memory whereas hardware isolation keeps them secure and isolated from user-installed applications and the main operating system.

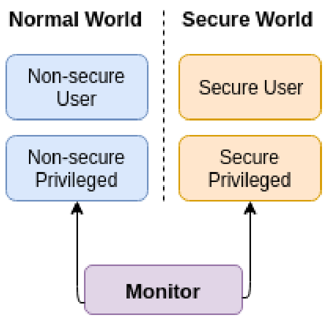


Figure 5-2 TrustZone Normal and Secure worlds

5.4 ARM Virtualization Extensions

The vision for the emergence of ARM servers as well the path to make ARM processors competitive and comparable to Intel's and AMD's x86 architecture processors in the datacentre could not be possible without the introduction of Virtualization extensions on ARM. Therefore, ARM adopted the same path as x86 architecture processors and introduced hardware-assisted virtualization extensions in its latest models of the ARMv7 architecture and improved them further on ARMv8 processors.

Taking into consideration the principles as documented and proposed by Popek and Goldberg [31], ARM introduced and implemented hardware-assisted virtualization extension features capable of supporting and providing an efficient full virtualization environment and equivalence to several Guests Oss.

5.4.1 Hardware-assisted Virtualization

Because of the need to overcome the burden of binary translation implementation achieving faster processing and improved performance on virtualized systems, CPU manufacturers of x86 architecture (e.g., Intel introduced a set of virtualization extensions as part of their processor architecture). Hardware-assisted virtualization technology introduces a new layer in the x86 CPU ring architecture, Ring -1 where the hypervisor can load, and Guest OS can run on Ring 0 just as on non-virtualized systems as is visualised in Figure 5.3. Processors with hardware-assisted virtualization technology enabled, support a set of new instructions called VMX (Virtual Machine Extensions). VMX operations run under two modes, root and non-root operation modes. This set of instructions permit a restricted number of operations and values that control what registers can accept.

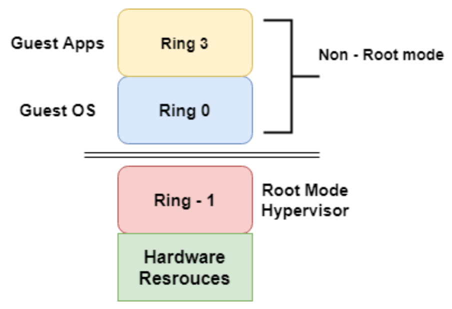


Figure 5-3 x86 Ring privileges architecture

In 2005 Intel [59, 240] introduced a suite of hardware-assisted virtualization extensions using the brand name VT-x. Intel's VT extensions utilize those executions modes where operations can run either on root-mode or non-root or else Guest-mode. A hypervisor transitions between those two modes through the execution of VM entries and VM exits as illustrated in the Figure 5.4. Through a VM Entry operation, a VMM transits from a root mode state to a non-root state and, vice versa, through a VM Exit operation return from a non-root state to a root.

In a VMX root mode state, a set of new processor instructions are available and the values that can be loaded to control registers are restricted. Because VMX operations permit and allow a specific number of actions to take place, a Guest software is able to run and be executed at the privilege level that was intended to be run by design. VMX operations define a specialized data structure that keeps track of VMX transitions through root and non-root operations as well as the processor state in each mode. A hypervisor makes use of a separate instance of a VMCS for each virtual machine per virtual CPU.

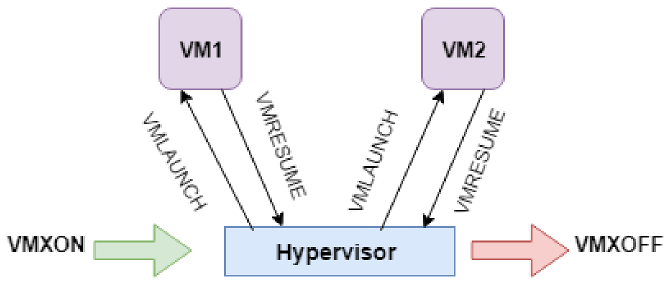


Figure 5-4 Intel VT-x CPU Operations

5.4.2 ARM Exception Levels

In order to understand better the new virtualization extensions as first introduced on ARMv7 and embedded on ARMv8, one first needs to explain and introduce the concept of privileges. Systems are enhanced with several protection domains, known as protection rings on the x86 architecture, that control, manage, and protect access to hardware resources like memory and CPU from unauthorised or malicious intentions. For example, the operating system's kernel has a higher level of access to system resources than a user application running upon it, which has limited permissions to performing modifications or configurations to the system. The ARM architecture implements a slightly different scheme of levels of privileges. Instead of rings, it adopts a horizontal privilege architecture known as Exception levels.

Code execution can have either an unprivileged or privileged access level. The unprivileged access level has limited visibility and access to only a specific number of system registers and to a protected portion of memory regions. When a user application tries to access restricted resources then the process generates a fault condition and therefore exceptions used to take over and handle those accesses. In contrast, the code that is executed on privileged access levels has access to all resources and memory regions with no restrictions. The exception level scheme defines who can execute, which complex list of instructions on what resources. The 'who' part refers to each of the available CPU modes that a processor is running.

ARMv7 processors follow a slightly different privilege level design from the ARMv8 processors, which define three distinct privilege levels that are defined by the *PLx* definer as Figure 5.5 presents. Each privilege level has an identification number starting from 0 to 2 and the higher the number means the higher the level. PL0 is the least privileged level where user's applications run. This software 'does not have permissions to access or implement any configuration settings and the only permissions that software has access to is the unprivileged portion of memory.

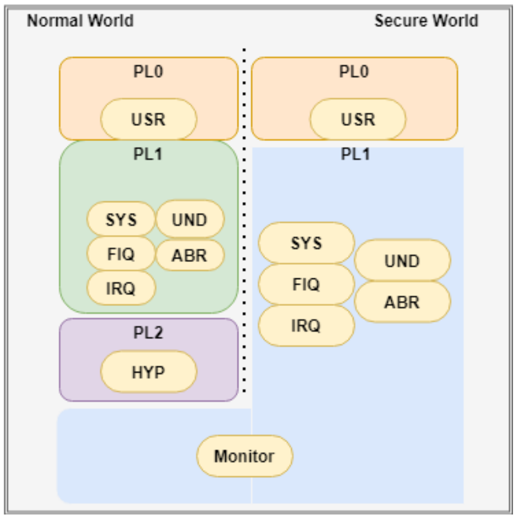


Figure 5-5 Available CPU modes on Normal and Secure world respectively

In contrast, on ARMv8 processors those privilege levels are called Exception levels that bear the initials of *Elx*. Furthermore, an additional exception level was introduced. So, the ARMv8 architecture is divided into four exception levels, EL0 to EL3 where, similar to ARMv7, EL0 level is referred to as the least privileged while EL3 the most privileged level. Several modes become available depending on the exception level. CPU can be executed in one of the four exception levels as demonstrated in Figure 5.6. Exception levels determine the privileged level as defined on ARMv7 architecture.

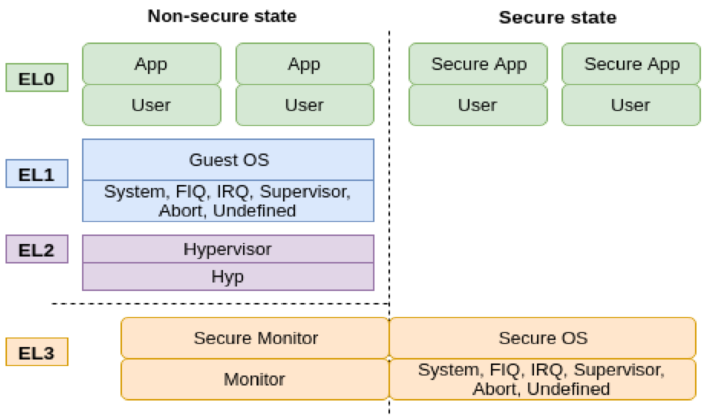


Figure 5-6 ARMv8 Exception model

ARM hardware-assisted virtualization extensions as introduced on ARMv7 (Cortex-A9) are focused on the introduction of a new processor execution mode called HYP mode. As with any system, on ARM an operating system kernel runs in SVC mode while user's applications run in the USR mode. The HYP mode has higher privileges from both SVC and USR mode and that's why HYP mode is running in the EL2 privilege level with dedicated access to registers and memory space. HYP mode has a separate set of instructions available as well as its own set of registers. Since HYP mode is running on a higher, distinct exception level than the SVC mode, there are no conflicts or sharing of code execution with the kernel on SVC mode, making it clear and independent. The HYP mode was developed to provide features to a candidate hypervisor and not for running an operating system like Linux kernel since that would require a lot of changes and source code modifications of the kernel. As we described in Section 5.1. ARMv7 and v8 architectures offer two worlds of execution and operation, secure and normal or non-secure world. Virtualization extensions are implemented only when a core is running in the non-secure world. System administrators have to ability to configure on which exception level or what instructions should be trapped on the hypervisor. They are able to choose if the code exceptions should be trapped always and only on the HYP mode, handled by the hypervisor or some of those should be treated by the kernel in SVC mode.

HYP mode by default is disabled. In order to access and enable the virtualization extensions, we have to enable it during the boot process of the system. As Figure 5.7 illustrates during the boot process the system needs to enter the secure world and change to the Monitor CPU mode. Through that it can activate and access the HYP mode under the non-secure world.

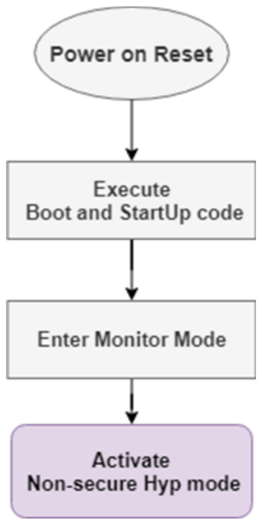


Figure 5-7 ARM normal boot process

Through the HYP mode, a hypervisor handles the content or world switch between virtual machines when a Type 1 hypervisor is being used or between the host and the virtual machines on Type 2 hypervisor. The processor follows the same procedure as it does when an exception is raised. The processor needs to enter the HYP mode with an EL2 exception level in order to save and store the current state of the processor.

In order to perform a complete content switch among VM and the host, a series of registers need to be stored and retrieved as listed below.

- General Purpose Registers (GP)
- Page Table pointer (PT)
- Floating – Point Registers (FPU)
- Banked Registers for all kernel modes
- System Coprocessor Register (MRS)
- Address Space ID (ASID)

Very frequently, during the normal code execution of a program, a user requires access to more restricted resources that requires higher privilege permissions than user's mode (USR). So, an exception is triggered that makes the CPU change the mode to a more privileged level. Which exception level is based on the type and kind of exception handler that deals with the request. A more privileged software called exception handler runs at a higher privilege level then handles this request and returns the result back to the software that raised it. There are different types and kinds of interrupts like FIQ, IRQ, Aborts, each of them has a linked interrupt handler.

The following procedure takes place during an exception:

- The CPSR is stored into the banked SPSR register of the particular mode where the exception is handled
- The current processor mode and exception level are set based on the type of the exception
- The interrupt bits are set on the CPSR register
- It then stores the return address into the banked LR register

Once the exception has been processed the following steps are executed in order for the system to continue the code execution before the exception.

- The stored value from the SPSR register is copied back to the CPSR in order to restore the state of the status register.

- That automatically changes and restores the processor's mode and exception level.
- The banked LR register is copied back to the PC.

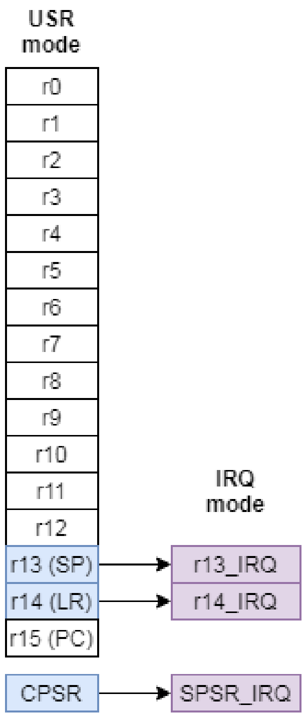


Figure 5-8 Example of ARM Banked registers mechanism

5.4.3 ARM Registers

Latest versions of ARMv8 processors support up to nine execution modes, also called processor modes, as listed in Figure 5.9. The processor mode can be modified and manipulated in two ways, either by a program that runs on one of the privileged modes or through a hardware interrupt.

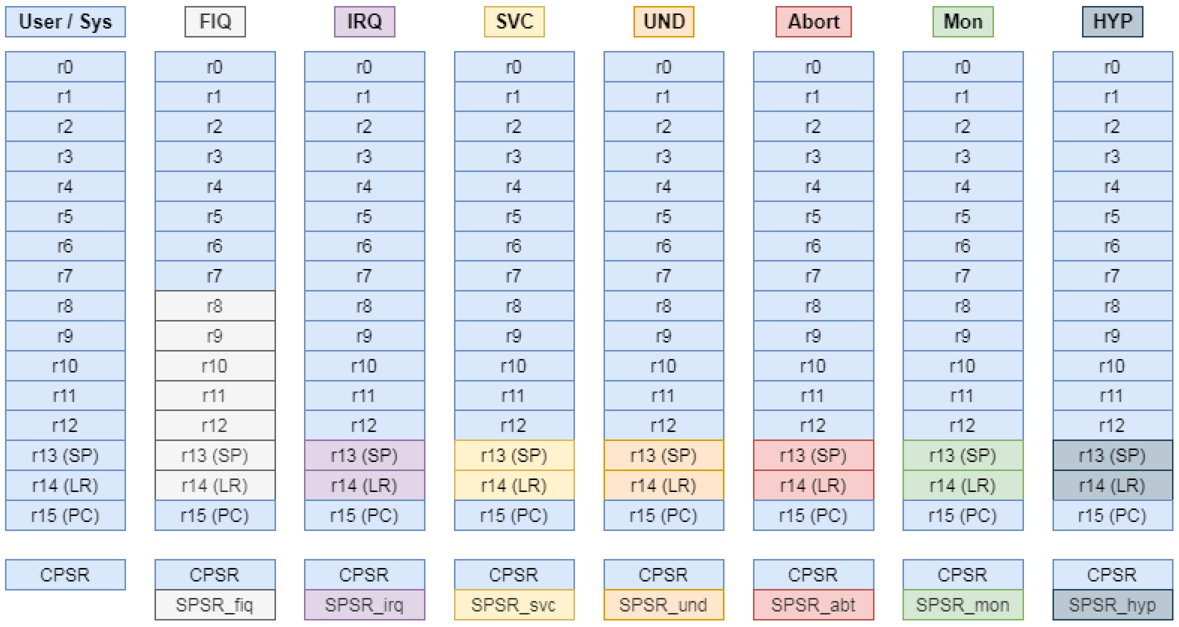


Figure 5-9 ARM List of registers per CPU mode

ARM processors, depending on the version, have several general-purpose registers, a program counter (PC), an application program status register (APSR), a current program status register (CPSR) and saved program status registers (SPRSs). Some of those registers are hidden during normal execution of the system but become available and accessible when the processor accesses a particular mode. Those registers are called banked registers. ARM processors due to the structure of that banked register, provide rapid context switching when a processor needs to handle exceptions, executing privileged instructions.

The current processor status register (CPSR) reflects the current execution state of the processor. The first five bits [4:0] define the mode where the processor is running each time. There follows the list of the CPU modes that a processor can be in as well as the combination of the bits of them in Table 1. From the list, the User mode is the only unprivileged mode while all the rest of the modes are more privileged and can be used to execute some system operations and tasks.

M [4 : 0]		Mode
Bin	Hex	
10000	10	User
10001	11	FIQ
10010	12	IRQ
10011	13	SVC

10110	16	Monitor
10111	17	Abort
11010	1A	Hyp
11011	1B	Undefined
11111	1F	System

Table 1 ARM CPU supported execution modes

The ARM architecture is in continuous development, introducing and implementing more and more new features over time and ARMv7 has some significant differences from the successor ARMv8 family of processors. Besides the supported exception levels as mentioned, another important difference is the support for 64-bit registers on ARMv8 architecture. That feature also brings change on the number of supported registers, which also affects the number and use of banked registers. ARMv7 architecture processors follow a mode-based banked register scheme where each processor mode has a dedicated set of banked registers. On ARMv8 that architecture has changed. Instead, an exception-based banked register model is followed and several "special" registers were introduced as presented in Figure 5.10.

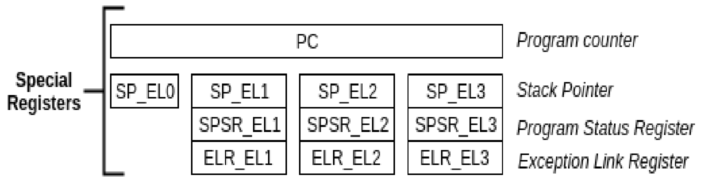


Figure 5-10 ARMv8 Special Registers

In case of an exception, the current execution state is stored in the following dedicated registers based on the exception level:

- Exception Link Register (ELR)

  and
- Saved Processor State Register (SPSR)

Where the ELR register is to hold the return address after the completion of the exception and the SPSR register holds the value of the PSTATE fields. As we mentioned earlier, on ARMv7 architectures, each processor mode had its own dedicated SPSR register to maintain and store the value of the CPSR. However, on ARMv8 each exception level has a dedicated SPSR register for that task.

Under the ARMv8 instruction set architectures, there is no equivalent of the CPSR register like on the ARMv7 instruction set architecture. Instead, it has a collection of fields as Processor state (PSTATE). Processor state fields are accessible through the use of specialised MRS and MSR instructions respectively.

5.4.4 Memory management

One of the key operations and functionalities handled by an operating system is the memory management system, mapping and translating virtual addresses, also known as linear addresses in the computing architecture, of user applications to the actual physical address on a system.

A memory management system performs a dynamic allocation of regions of memory to operating systems and applications. Normally, an operating system or a hypervisor if we are referred to a virtualized system, keeps track of a virtual to physical translation address mapping into a translation table also called a pages table. Those translation tables are stored in memory and controlled by an OS or a hypervisor therefore are not a static content but continuously updated based on the running task or application.

Normally, an operating system owns all the physical memory on a system and is responsible for the management of this as well as the allocation of dedicated memory spaces to applications on demand. In non-virtualized systems, the translation of virtual memory address to physical memory occurs and is managed by the OS, however, ARM virtualization extensions enable a 2-stage memory translation process similar to the shadow pages functionality on x86 architectures. This 2-stage memory translation is a hardware-embedded feature enabled where the first translation phase is handled by the Guest OS by mapping Guest virtual memory translated to Guest physical memory by the Guest OS while the second translation phase is handled by the hypervisor where the intermediate physical address (IPA) is translated to actual physical memory as shown in the following Figures 5.11 and 5.12.

If the Stage-2 translation feature is enabled, then the following procedure takes place as illustrated in the Figure 5.11:

- Stage-1: At first the virtual address (VA) is translated into an intermediate physical address (IPA)
- Stage-2: The results of intermediate physical address (IPA) is mapped to a physical address (PA)

The first stage of translation is handled by the Guest OS while the stage two translation is controlled by the hypervisor. A Stage-2 translation provides a secure way of accessing physical memory only from HYP mode, completely transparent to Guest OS and user.

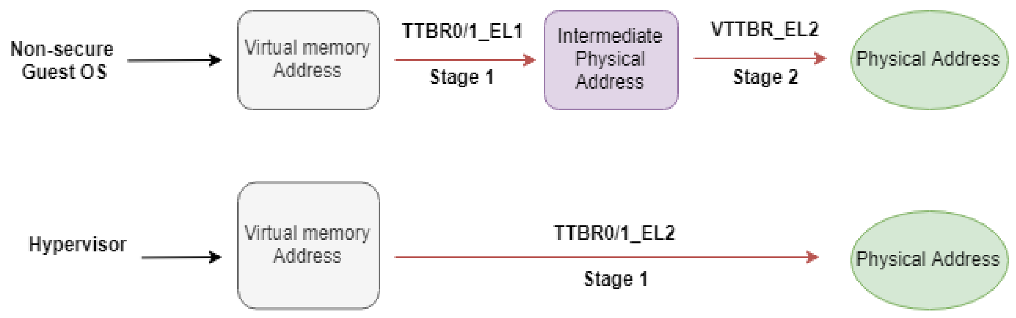


Figure 5-11 ARMv8 2-stage table translation management

In User mode, there can be many translation tables while on the hypervisor level (HYP mode) and on secure monitor level, only a single translation table exists.

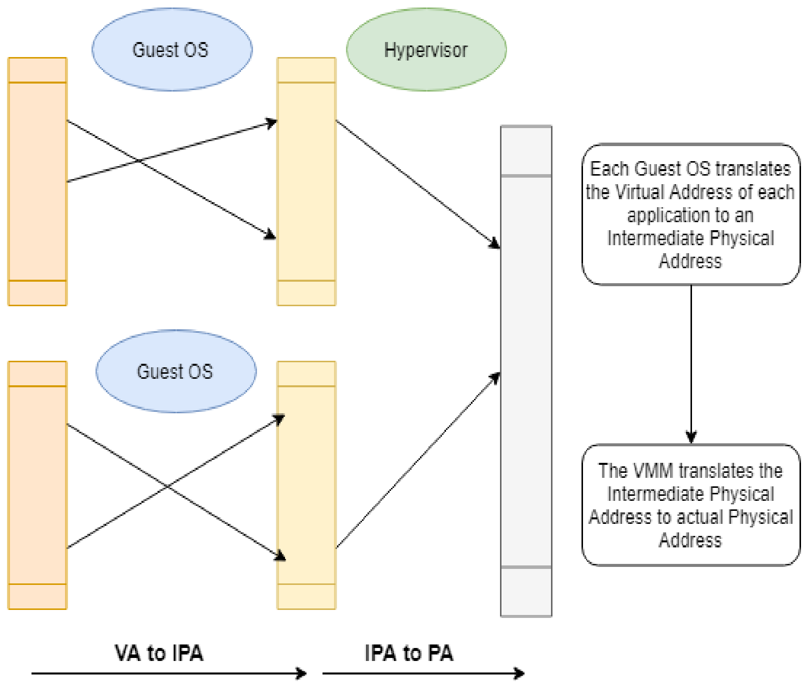

Guest OS
Hypervisor
Guest OS
Each Guest OS translates the Virtual Address of each application to an Intermediate Physical Address
The VMM translates the Intermediate Physical Address to actual Physical Address
VA to IPA
IPA to PA

Figure 5-12 ARMv8 2-stage memory translation system

5.4.5 Comparison between ARM and x86 architectures

At a higher level, both architectures support virtualization features that have a similar concept of functionality, but the implementation is done through a different structure. Both on ARM and x86 architectures, virtualization extensions are embedded to ISAs with the goals to implement and satisfy Popek and Goldberg [31] principles in order to provide an efficient and effective virtualized system.

Seen from a higher perspective, both architectures support a two-stage memory translation mechanism as explained above but Intel refers to it as Extended Page Tables (EPT). Both architectures also support interrupts and exceptions handlers, supported by the hypervisor and returns to a Guest with a virtualized value. Finally, both maintain and store Guest and host state separately.

Major differences between Intel's VT-x virtualization suite and ARM are that the latter gives the flexibility and privilege to a software programmer to decide which and what Guest state the hypervisor should save rather than it being done automatically by the hardware. The x86 architecture follows an orthogonal privilege scheme where the hypervisor runs in the privilege ring like the kernel does but unlike on ARM where hypervisor runs on a new mode (HYP), that is executed on a distinct CPU mode, more privileged than kernel mode..

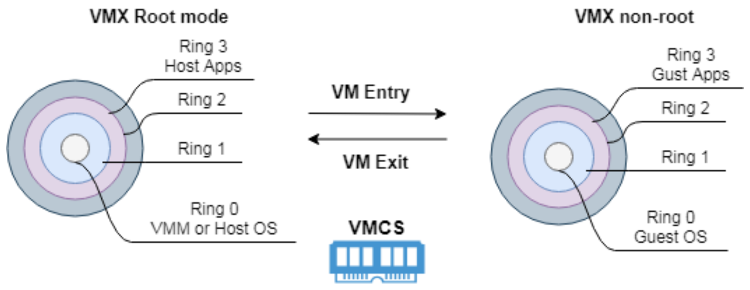


Figure 5-13 Intel's x86 privileges ring scheme

During a Bare metal VM live migration the CPU state of a Guest can be easily identified. Intel's hardware-assisted virtualization is enforced with a specialised data structure called VMCS as demonstrated in Figure 5.14 that keeps and maintains the state of a VM as well as the state of the host machine before transit to a non-root mode. In contrast, on ARM architectures there is no similar data structure. ARM gives freedom to developers to configure and save the type and number of registers that they want to maintain.

A VMCS is created for each of the virtual CPUs on a virtualized system that contains all the sensitive information of the current state of a vCPU. A Guest state is loaded in a CPU when a VM Entry instruction is executed while the host state is loaded when a VM Exit instruction is executed. By implementing these VT-x extensions, no OS modification or binary translation is needed in order to support multiple VMs. However, frequent execution of VM Exit instructions slows down system performance and increases the latency. VT-x introduced two CPU operations that allow a CPU to run and be either in root mode or in non-root mode.

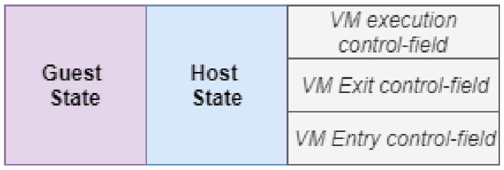


Figure 5-14 Intel's x86 VMCS packet structure

In contrast, on x86 architecture there is no such a separate extra CPU mode and privileged layer. Intel introduces an orthogonal virtualization architecture to separate user mode from kernel and virtualization by dividing to non and root mode. Figure 5.8 illustrates the VMX root and non-root modes in correlation to the x86 ring architecture which traditional x86 processors use.

VMX root mode is used by a VMM while in VMX non-root operation a Guest OS runs. Both states support all four privilege levels, so a Guest OS can run at its intended privilege level. Under VMX non-root mode lots of instructions can cause a VM Exit instruction increasing the latency and degradation of performance. For example, in case of a move operation that take place between a register and a control register like "MOV from/to CR3" (control register) or in cases of an external interrupt and during a page fault, a number of several VM Exit and VM Entry operations are triggered in order to serve those requests resulting on degradation of system's performance.

Transition from root to non-root operation is handled by two VMX transitions. VM Entries is called during the transition from a root privileged environment to a non-root, while during the execution of a VM Exit instruction the operation and control returns to root (hypervisor).

A CPU running on non-root mode is equivalent to a non-virtualizable environment when sensitive and privilege instructions from VMs need to trap to hypervisor, CPU swaps from non-root to root mode. In addition, ARM traps to HYP mode.

5.4.6 ARMv7 vs ARMv8

In the last few years, the ARM architecture has continued to make progress by introducing new versions, even if this pace of advancement creates confusion and frustration in the software development community. Mainly, most of the changes and differences among ARM versions are focussed on the Instruction Set Architecture (ISA), the operational and system modes and list of available registers.

In version 8 of the architecture (ARMv8), ARM introduces support for the 64-bit instruction set and offers the capability for two execution states, AArch64 with 64-bit registers and AArch32 with 32-bit registers for backward compatibility with the preceding ARMv7 architecture. In comparison, ARMv7 uses the A32 and T16 instruction sets (32-bit and 16-bit, respectively).

Therefore, the live migration scheme will differ from one ARM version architecture to another, and adjustments need to be made in order to provide compatibility with both AArch32 and AArch64 execution states. This section will provide a brief overview of the core differences among ARMv7 and ARMv8 architectures based on the adjustments that need to be made so our live migration scheme will work on both ARMv7 and ARMv8 versions.

The four main differences that need to be considered between the ARMv7 and ARMv8 are:

- The availability of two execution states
- The number of the available registers
- The introduction of the suffix *_ELn* based on the exception layer
- The access to the System registers

The AArch32 execution state is an evolution of ARMv7 enhanced with some new instructions and is mainly used for compatibility reasons. When the processor enters the AArch64 execution state, 64-bit wide registers are available as well as a new exception model architecture. It is therefore important for our live migration scheme to know the execution state in order make adjustments related to the available registers.

Additionally, the ARMv7 architecture provides 16 general purpose 32-bit registers, ranged from R0 to R15. On the other hand, ARMv8 provides 31 64-bit wide registers. However, each of these registers can be used either as a 64-bit (X0-X30) or as a 32-bit wide register (W0-W30). As such, the same register has two ways of representation based on the execution state as Figure 5.15 illustrates.

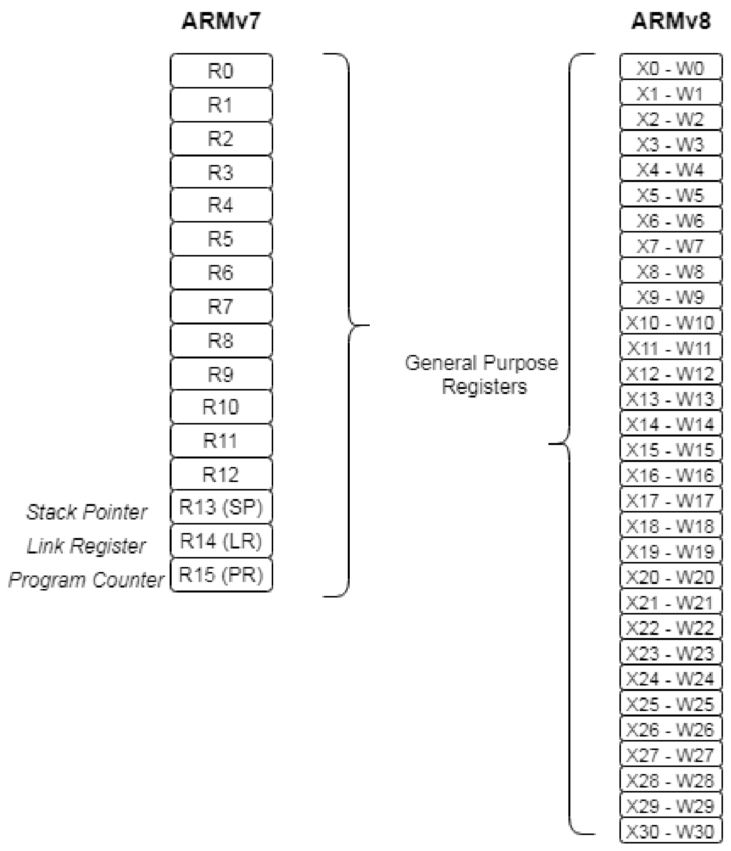


Figure 5-15 *List of general-purpose registers on ARMv7 and ARMv8 architecture*

As we mentioned, ARMv8 introduces a new exception model architecture using the prefix *ELn* as Figure 5.16 shows where *n* can take values from 0 to 3 with 0 being the least privileged, normally this is the user mode and 3 to be the most privileged. That also brings changes in the use of the bank registers compared to the ARMv7 architecture.

Furthermore, there is a huge architectural design difference in the way that the ARMv7 architecture treats the use of existing banked registers while on ARMv8 has embedded those are called special registers. On ARMv7 when the CPU changes between different modes, there are some banked registers dedicated to that specific mode making the content switching process much faster. However, that concept changed significantly on ARMv8. Instead of having banked registers based on the CPU mode, ARM introduce banked registers based on the level of the exception that is received.

Moreover, on ARMv7 the architecture contains two special Program Status Registers (PSRs), the Current Program Status Register (CPSR) and the Saved Program Status Register (SPSR) which stores the state of the processor at a given time and the value of the CPSR when an exception or interrupt signal is received respectively. In ARMv8 the current program status register is not available when the processor is running in AArch64 execution state. The functionality of the CPSR is replaced with a Processor state (PSTATE) register which contains all the information about the current state of the processor, including the current mode.

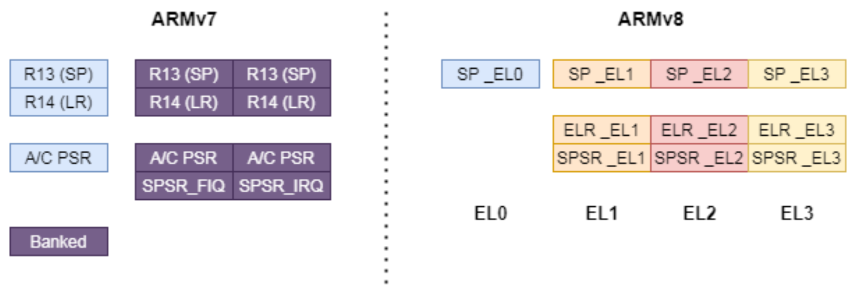


Figure 5-16 Banked registers and special registers

On ARMv7 the System Control Register known as SCTLR is accessed using the Coprocessor 15 (CP15) which offers access to many features. In contrast, CP15 is not needed on ARMv8 and the software can access system registers using the MSR and MRS instructions. In Figure 5.17 we follow an example of the use and access of the system control registers through the P15 and SCTLR registers respectively.

ARMv7	
MRC	p15, <R>
MCR	p15, <R>
ARMv8	
MRS	<R>, SCTLR_ELn
MSR	SCTLR_ELn, <R>

Figure 5-17 Accessing system registers on ARMv7 and ARMv8

Another significant change is the naming conversions for calling the registers. On ARMv8 most system registers now end using the format of the Exception level that are called or used.

So, based on the differences as explained above, creating a universal live migration scheme on ARM systems is extremely challenging. Our design and implementation is based on ARMv8 since it was the latest version and is embedded in the processors that are intended to be suitable for edge and datacentre infrastructures.

5.5 ARM Hypervisors

ARM architecture processors cover a wide range of hardware platforms, and are widely found in applications on embedded devices like mobile phones, PDAs, tablets and laptops etc. Hypervisors on ARM based systems is therefore an attractive area of research to the academic community. Hypervisors not only work as a management tool to the system and underlying hardware infrastructure, offering an enhanced layer of security by preventing unprivileged instructions or malicious access to kernel's operation but also enabling the ability to execute multiple OSs on a single platform. Two types of hypervisors can be found in applications on ARM based platforms, real-time hypervisors, which cover most embedded devices and system hypervisors finding implementation mostly on desktop/server systems. Although the ARM architecture has existed in the IT industry for many years now, only a small number of hypervisors exist in comparison to x86 processor architectures. This is mainly due to continuous development of the ARM architecture which leads to incompatibility from one version to another. For example, the number of registers, new CPU modes, and register format differences that can be found from ARMv6 to ARMv7 and ARMv8 models.

Hypervisors are not only a management tool for servers, they are also used by embedded devices and the automotive industry. Our interest and application focuses on Type 1 (Bare metal) hypervisors for data centre infrastructures which have direct access to the hardware, offering native performance and better management controlling the access to the underlying hardware infrastructure. A list of some existing ARM hypervisors in the research community as well in industry are listed in Table 2.

Real-time Hypervisors	**Desktop/Servers Hypervisors**
Xen	Xen
ViMo	ViMo-S
OKL4	Xvisor
RTZVisor	
Minos	
ARMVisor	

SierraVisor	
Xvisor	

Table 2 ARM-based Type 1 and 2 Hypervisors

With the introduction of powerful ARM processors into data centre infrastructure, the development of ARM hypervisors becomes an active field for the research community. Only a few hypervisors are currently available to server infrastructures.

Xen: Is one of the most famous Type 1 hypervisors finding application on both ARM and x86 architecture environments. Is a member of a (PV) Paravirtualization family which means that the Guest OS needs some kernel modifications in order to run in a virtual machine without the need of emulation. Xen paravirtualization support on ARM system is achieved through hypercalls. Hypercalls work in a similar way like systemcall works for the operating system when it needs to execute some privileged instruction on hardware. Hypercalls are an advanced processor instruction, enabled when an HVC processor instruction is called, where a Guest OS informs the hypervisor to handle a privilege instruction. Mostly it is used when the hardware does not offer virtualization extensions, like Cortex – A5, A8, and A9 processors models. Xen is one of the most famous hypervisors in the industry. When it is installed by default it creates a virtual machine called Dom0 (Domain 0) where a set of device drivers reside called PV Backends (Paravirtualised). Dom0 provides access to the unprivileged Guest machines. Each Guest virtual machine created is called DomU (Domain Unprivileged) which is installed with a set of Para virtualized frontend drivers. On Xen each PV Backend is shared with multiple PV frontends where in practice every time a Guest DomU makes an I/O request that is performed and delivered by Dom0. Although, Xen supports live migration functionality, performance is poor due to the Domain structure model that follows and the need for continuous use of hypercalls.

ViMo-S: ViMo-S is another Type 1 hypervisor on ARM systems. However, ViMo-S is under development and works only for experimental purposes. ViMo-S is the successor of the ViMo project, a hypervisor designed to find implementation on ARM based mobile platforms. However, ViMo was not designed based on ARM VEs. Therefore, its successor, ViMo-S was developed based on ARM VEs taking advantage of more privileged execution layer, EL2, offering the support of console, network and disk virtualization devices. Like Xen, it makes use of a domain structure creating by default the Dom0 virtual machine. ViMo-S offers the support of unmodified Guest OS unlike Xen by using full virtualization for CPU and Memory taking advantage of the ARM VEs while using Paravirtualization for I/O requests and interrupts. However, ViMo-s remains a research proposal with no migration features or support for full hardware virtualisation.

Xvisor: Xvisor is a Type 1 (aka Bare metal) monolithic hypervisor, meaning that it runs directly on the hardware while it contains all necessary hardware drivers in order to provide access to the underlying hardware resources, including storage, network and input/output peripheral devices. Consequently, it can find application and implementation on limited hardware support. Nevertheless, it has attracted a lot of interest in the research and academic community. It is available for a wide range of both embedded systems and server platforms. Xvisor finds implementation on a wide range of hardware platforms supporting a huge range of ARM processor models like ARMv6, ARMv7 and ARMv8. Xvisor offers both the support for full virtualization to unmodified Guest OSs and paravirtualization solutions using hypercalls, similar to Xen hypervisor, for compatibility with some older processor architectures.

The development community of Xvisor is one of the few that is still in continuous development, offering upgrades and improvements adding new functionalities. Xvisor offers detailed documentation of how to get installed on a wide range of ARM based platforms covering a huge group of embedded platforms as well computing and server systems like Raspberry hardware models and AppliedMicro X-Gene 1 server platforms. Although Xvisor does not offer any kind of migration functionality, however, the lightweight code footprint, well designed structure that takes advantage of ARM VEs, has led is leading to becoming a promising hypervisor of choice for many future research projects. to it becoming our choice to extend and adapt our code in order to implement the migration functionality on that.

Since host device drivers are part of Xvisor's source code, there is no need for additional context switch and scheduling mechanisms when a host interrupt is rising, eliminating performance overhead. As the following Figure 5.18 illustrates, during the normal execution of the Guest OS when an interrupt is raised, Xvisor running on HYP mode can handle it. Furthermore, processor scheduling takes place per CPU while load balancing for multiprocessors is handled as a separate entity.

An extensive analysis and comparison of Xvisor to Xen and KVM hypervisors was executed and presented by Anut Patel et al [249, 269] where it was shown that Xvisor, due to monolithic design, performs much better than Xen. All key operations of Xvisor run on a single software layer with the highest privileges, including virtualisation features.

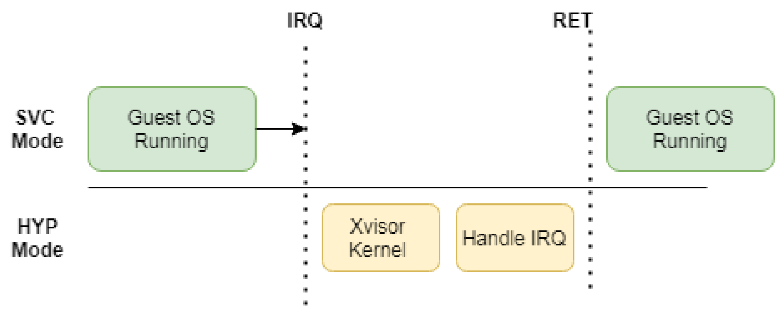


Figure 5-18 Interrupt handling process

5.6 ARM Live Migration

Existing ARM hypervisors lack an efficient, operational live migration scheme. The increased demand on ARM based systems, their adoption on data centres and targeting of Edge Computing solutions with the announcement of ARMv9 to be available in the market, make live migration an important and necessary tool in this space.

The introduction of hardware-assisted virtualisation extension features on ARM processors increased interest in discovering its capabilities by both the academic and research community as well as the industry. Two of the most widely known competitors and widely used open-source hypervisors, Xen and KVM, leveraging ARM virtualization extensions introduced support for unmodified Guest operating systems on ARM systems. According to the performance evaluation of Christoffer Dall et al [118], operations like interrupt handling and content switching are completed faster on a Type 1 hypervisor, are compared to a Type 2 hypervisor on ARM, while Xen Type 1 hypervisor on ARM achieves a higher performance compared to x86. In contrast, a Type 2 hypervisor like KVM on ARM experiences a higher performance overhead compared to the x86 architecture. Therefore, our interest is more focused on a Type 1 hypervisor on ARM based systems to support Bare metal instances.

A hypervisor normally performs and manages several tasks and has several responsibilities that cost on a system's performance, such as memory management and translation, device emulation, exception and interrupt handling, instruction trapping and context switching. However, when it comes to live migration of ARM based Bare metal instances, the list of responsibilities can be remarkably reduced, something that could minimize if not eliminate the performance overhead.

To the best of our knowledge at the time of writing this thesis, no existing, operational solution exists on ARM-based hypervisors to perform a live migration among ARM systems. Therefore, we proposed a live migration implementation based on the lightweight Xvisor hypervisor architecture.

5.7 Summary

For decades ARM chips have dominated embedded devices, and extensive application has been found on a variety of devices like wearables, mobile and modern smart technologies. With the rapid growth of IoT devices and the increased demand for Edge Computing architecture design implementations, ARM in an effort to remain competitive announced the introduction of Cloud-to-Edge specific cores available for servers and data centre infrastructures. Modern AI and machine learning cores were not the first approach of ARM in the data centre world. The widely used Cortex X processor family covers a wide range of systems and needs.

Virtualization is an integral part of today's systems, being found widely on Cloud datacenters and many other areas. Therefore, ARM could not be out of that. Most recent versions of Cortex X processors support hardware-assisted virtualization. Therefore, ARM like any other vendor who wants to stay in market introduced the support of hardware assisted virtualization features in most recent versions of Cortex X family processors. Although hardware assisted virtualization features are available on ARM processors, a limited number of hypervisors are available in the market that are appropriate to support server and data center infrastructure.

6 High level design overview for a CPU live migration scheme

6.1 Introduction

In Chapter 3 we introduced our novel idea of utilizing Bare metal instances on Micro Data Centres as edge nodes to support the Edge Computing paradigm. The lack of availability of a live migration scheme on Bare metal instances attracted our interest in developing such a scheme targeting ARM based systems. This chapter introduces a case study where our novelty of executing a CPU state live migration process could find practical implementation. The following sections provide a high-level design overview and the requirements that need to meet to implement and support our scheme., analysing the core components that take part during the migration process as well as the phases of implementation. We provide a detailed description of the infrastructure dependencies and requirements where our novel scheme finds application.

6.2 Case Study

The rapid growth of the IoT as explained in Chapter 2 requires drastic solutions and workload redistribution from the present "all-to-Cloud" architectural model where it mainly dominates, to a more robust, distributed architecture capable of providing low-latency, high efficiency, and flexible performance to end users. The emergence of Edge Computing solutions as mentioned in Section 2.3 to offload the traffic generated from a variety of IoT devices to the cloud, brings data processing and computation closer to the edge of a network rather than shipping data produced by IoT devices directly to the cloud. Modern edge cloud solutions find practical implementation through the utilization of Micro Data Centres installed in distributed locations as discussed in Section 2.4

More and more use cases in the research and academic community show interest in exploring the performance of Micro Data Centres at the edge. For example, an attractive subject that has emerged is the utilization of ARM based Raspberry PIs (RPIs) as Edge Gateways on edge cloud architectures instead of using high-performance computer servers [247]. The driving force that gave rise to explore that field was when the authors in [232, 233, 243] managed to compose a cluster of RPIs consisting of 300 or more single nodes offering great opportunities due to their high energy efficiency, cheap implementation, and easy adaptability due to the small footprint of design, making them highly suitable for Micro Data Centre infrastructures where the attributes of portability, flexibility and efficiency are highly in demand. The combination of a large number of RPIs in a cluster helps to overcome and offset the drawback of the performance limitations that an RPI has due to structure and hardware architecture limitations, achieving a resource aggregation.

Figure 6.1 illustrates a typical edge architecture where an RPI cluster might find implementation, playing the role of an edge gateway as proposed in most recent use cases in academia [234, 235]. In this scenario, edge gateways are deployed in various geographical locations where IoT data is gathered for additional processing and computation before being backhauled to the cloud. On a cluster of RPIs, compute performance is gathered and shared among the same tasks, achieving higher compute, scalable performance feasible to support and deliver high levels of demand.

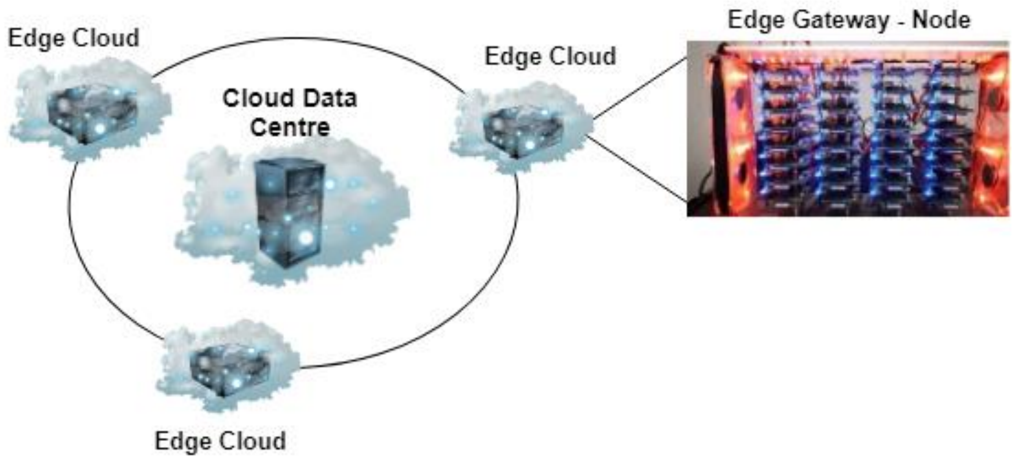


Figure 6-1 Adoption of a cluster of RPIs architecture at the edge

Recent studies have tried to investigate and evaluate the performance of a cluster of RPIs at the edge by testing a variety of workloads. Authors through several experiments have demonstrated the efficient and effective utilization of ARM based RPIs at the edge, reporting results of testing a variety of workloads and data. The demand for Edge Computing solutions inspired Ana Juan et al. [234] to introduce an ad-hoc edge cloud computing model which could easily be adopted and deployed on demand. The proposed design solution is based on the deployment of a containerised architecture which was composed of RPIs endpoints. Microservices and containerisation have become a popular architecture for containerised applications, known as microservices, enabling a continuous integration and delivery methodology. A containerised orchestration system consists of a centralised master controller and several worker nodes as Figure 6.2 illustrates. The master node controls and manages the resources and handles the distribution of manageability of workload among the workers, while the actual applications are hosted in worker nodes. As Ana Juan suggested [234], masters and workers could easily be replaced by a single board of

RPIs or even a cluster of them, creating a Kubernetes cluster capable of hosting and serving a highly efficient workload.

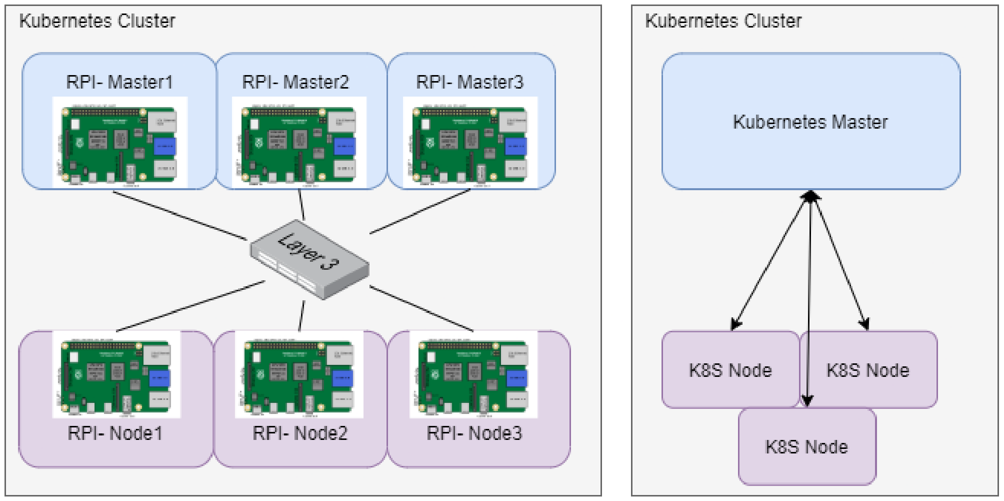


Figure 6-2 Microservice orchestration cluster architecture composed of RPIs

Working in the same field of interest, Claus Pahl et. al. [235] proposes a container-based edge cloud platform as a service architecture hosted on clusters of RPIs. Both approaches adopt the same architecture model as illustrated in Figure 6.2. Similarly, Remo Scolati et. al. [237] proposed a container-based lightweight Big Data streaming solution for edge Cloud infrastructures, based on clusters of RPIs. Authors in recent research experiments [244, 246] introduce and evaluate the utilization of clusters of stackable single board RPIs as edge nodes for data processing and analysis for the performance of machine learning applications and tasks. The evaluation results of the upper use cases show that modern versions of RPIs, working in a bundle architecture, can handle and perform advanced high performance, real-time operations while maintaining some Bare metal characteristics. Adopting a Bare metal architecture offers higher data privacy and users' isolation while system resources are assigned to dedicated tasks. In a similar way, edge cloud RPIs are members of an isolated network where single instances or bundles of them work on dedicated tasks. Even on containerised infrastructures the impact on system performance is minimal since we talk about an OS-level virtualization form and not a hardware-based virtualization. However, all these studies and use cases rely on the existence of a virtualization form such as containerisation in order to support the concepts of scalability, flexibility and resource manageability.

A single board, free of the virtualization layer acts as a single unit working on dedicated tasks, delivering higher computing performance as a result. However, fault tolerance, workload load balancing, quality of service and reliability are crucial attributes in order to guarantee a seamless integration and processing architecture so maintaining RPI boards as backup systems or having replicas of an existing topology in a cluster of RPIs in case of unexpected hardware failures, is highly desirable and affordable from both a cost and space point of view. However, the lack of support for performing live migrations among those platforms due to virtualization limitations makes this a challenging task. Therefore, our proposed solution and implementation can find extensive application in such environments. Migrating the system's state from a single RPI to another during intensive tasks like real-time data processing and data analytics could be valuable. Even more so in the case of failure, when memory and CPU state migration could maintain the continuity of the service since ARM based systems adopt and follow a memory-mapped architecture. Let us consider a use case scenario where we have configured the RPIs to process and execute a series of batch processes such as performing a Big Data analytic function on real time data flowing through the network. The RPIs are all connected together, through a LAN network where each of the RPIs is performing individually. However, all RPIs are part of the same project, each of them works on a dedicated task. A hardware failure or crash of the system requires the replacement of the single board of the failed RPI with a new one, as well as to start the processing from the start. In order to avoid this corruption and disruption of the process, we could perform a state migration of the CPU and memory components since storage is shared among the bundle of RPIs and no peripheral devices are needed. Our proposed solution could work and find implementation in that case transferring the CPU state through the network using a UDP socket for higher efficiency.

6.3 Design Decisions

When it comes on the design, development, and implementation of a live migration scheme especially on Bare metal instances, several requirements need to be met and decisions need to be taken. Live migration is a complicated process with many requirements as we have shown in Section 4.2. The state running on environment A with a specific infrastructure needs to reach location B with similar if not identical infrastructure maintaining the same environmental variables where identical infrastructure means that the key components such as CPU architecture, peripheral devices and memory capacity should match between the environments. Following the same principles, designing such an implementation on Bare metal instances is a challenging and very demanding process. Although Bare metal live migration scheme adopts the same basic principles as common Type 2 hypervisors support on virtualized systems there are some core differences. The main difference is that instead of taking place on top of a virtual machine manager (VMM), migration is performed among physical instances, migrating hardware state from a source to a destination system over the network. On top of this, there are some general requirements or goals that must be fulfilled in order to a migration scheme to be considered as efficient and practical. In the following section we will see those requirements as well as design decisions made based on higher constraints.

6.3.1 Base requirements for the development of a Bare metal live migration scheme

First, and most important, a Bare metal live migration model needs to be OS independent and free of user-interventions. A live migration should not be tied to a specific operating system while users should not be aware or interact with such a process in order to avoid any modifications that could interfere with the normal operation of the live migration process. Users need to be free to select an operating system of their choice and not limited on choices. Furthermore, in order to fulfil the fault tolerance attribute and acting proactively, a live migration process must be independent and user-free able to be triggered and executed at any time.

Moreover, an important aspect of the live migration process is to maintain system performance on the same levels as much as possible before, during and after execution of such process. The concept of performance, in this case, has many aspects that need to be taken into consideration such as the performance degradation and overhead that a functionality like that can cause, affecting the host system performance and the total amount of time that a live migration takes until completion. A software needs to run most of the time idle in the background minimizing the effect of system performance while inspects the incoming and outcoming calls on the system resources. During the execution of the live migration process, that software needs to be capable enough to access, gather as well as transfer the state among systems.

Several approaches have been proposed through time related to a live migration scheme on Bare metal instances with some of them finding practical implementation while others are just merely conceptual suggestions. Previous studies [250] suggested the introduction of a new module as part of the operating system kernel, however that solution requires kernel modifications, making it inefficient and impractical. Additionally, another proposed solution was using the operating system hibernation functionality, where the state is saved in memory or stored on disk. In both cases, this solution was inefficient since a huge amount of memory is demanded or a significant degradation performance was experienced during the migration of the state from the disk.

The first Bare metal live migration scheme that successfully finds practical implementation was introduced by Fukai et al. [37] called BitVisor. BitVisor is a thin, lightweight hypervisor layer, which accesses and exposes hardware resources to the Guest machine without virtualization but takes advantage of virtualization technologies like the VMCS structure in order to perform and deliver parts during a live migration execution. Following that and based on BitVisor's architecture, Im Jaeseong et al [220], proposed an on-demand virtualization layer making some modifications of the source code. The virtualization could be enabled and disabled during normal operation of the Guest and taking action only during the migration process reducing the performance costs. However, all those cases find practical implementation only on x86 architecture.

On ARM architectures, at the time this thesis was written and to the best of our knowledge, there is no similar functionality or any practical solution The emerging of ARM processors targeting on the edge, and the need of live migration on Bare metal instances being part of a Micro DC infrastructure, led us on develop a novel live migration scheme on ARM based on the following design decisions

6.3.2 Design choices of our novel scheme

Having in mind the base requirements as explained in Section 6.3.1 and inspired by Fukai's [37] implementation scheme on x86 architecture, we take the following design decisions for our novel live migration scheme on ARM Bare metal instances. A thin, lightweight hypervisor installed on top of the hardware where will act as a mediator that reads and passes values from the hardware to the software and needs to have the right permissions and privileges in order to gain access to the hardware resources.

The structure and nature of the hypervisor we believe that will be more efficient and effective If be monolithic, enclosing all the required drivers of the system. In general, there are two Type 1 hypervisor classifications, a monolithic and a microkernel type. The core difference between those two categories is the way that a hypervisor manages and deals with access to the hardware resources and drivers. A monolithic hypervisor contains all the host drivers of all the hardware resources that it needs to access, meaning that the drivers are part of the hypervisor architecture and code including networking and I/O peripheral devices. Additionally, it maintains all the hardware drivers and services like CPU scheduling, file management, memory management operations on the same partition. In that way, all the operations and access to the hardware resources reside and are executed within the most privileged level making the code more trustworthy. The advantages of using a monolithic hypervisor compared to a microkernel one are the efficiency and performance improvements while the correct system drivers are always available so it is very rare to face issues with unsupported or incorrect drivers. However, a monolithic hypervisor is larger in size than a microkernel since it needs to include all the necessary support drivers and, in case of a service crash, the entire functionality of the hypervisor crashes too. That architecture makes a monolithic hypervisor more dependent on specific hardware, limiting support. We believe that a monolithic Type 1 hypervisor fits better to a Bare metal live migration scheme since it offers a higher performance which is one of the main requirements. Furthermore, it provides higher security knowing that the implemented code is tested, and the installed drivers cover all the services and devices. In the case that a user manually wants to interfere with the underlying infrastructure or install customized drivers, this will not affect the migration phase which makes the scheme more reliable and secure.

Although there is a definition conflict on combining Bare metal instances with a hypervisor running on top of them, in our case, the hypervisor intervenes in the normal operation of the system only when a live migration process takes place minimizing the performance overhead. With respect of the definition of Bare metal instances only a single tenant, also known as the Guest machine, is running as a host on each system. That makes hypervisor's operation much simpler as eliminating the need for content switching or CPU scheduling mechanisms to take action and the need for virtualization of the peripheral devices by using pass-through mechanism which also reduce the impact of the hypervisor on the system.

The ARM architecture provides a clear structure on the level of privileges in association with the Exception levels as introduced in Chapter 5. That structure also makes the design process much easier since a dedicated privilege and exception level is used by the hypervisor layer while a Guest OS can maintain direct access to the hardware. Through that way, a type 1 hypervisor is executed on a higher, more privileged level than Guest OS without affecting or slowing down Guest operation.

Of course, creating a hypervisor from scratch is an extremely demanding and time-consuming project. Therefore, we decided to develop our live migration scheme based on one of the existing ARM based hypervisors. Although ARM processors are becoming more and more in demand there are still few Bare metal hypervisors available to the industry. Therefore, we must choose from a limited number of available Type 1 monolithic hypervisors with capabilities to support data centre and Edge Computing environments. Xvisor hypervisor looks to be the best candidate for our implementation.

The choice of Xvisor leading on a more restrictive list of development boards where we can test our scheme. The high popularity of RPIs in the industry led us to choose this as the development board. The decision to work with the latest version of ARM processors automatically allows us to work with two of the most recent RPI versions, the RPI3 and RPI4 that both make use of the ARMv8 architecture processor. We will learn and discover some of the features of RPI platforms in the Section 6.3.3.

One of the peculiarities that exist when designing a Bare metal live migration scheme is that it is tied with the underlying system infrastructure. Therefore, it is very challenging to develop a universal model that fits and works upon any hardware specifications. This is especially so when it comes to the ARM architecture, where the instruction set architecture and register specifications vary from one version to another. So, based on the differences as explained in Section 5.4.6, creating a universal live migration scheme on ARM systems is extremely challenging. Our design and implementation are based on ARMv8 since it was the latest version and is embedded in the modern processors that are intended to be suitable for edge and datacentre infrastructures. Therefore, the live migration scheme will differ from one ARM version architecture to another, and adjustments need to be made in order to provide compatibility with both AArch32 and AArch64 execution states.

Following, we will analyse and describe our choices of the Xvisor as the hypervisor and the RPI as the development board.

6.3.3 Xvisor Architecture Design

From the presented list of ARM based hypervisors as reviewed in Section 5.5, we chose to develop and apply our system on Xvisor. The main reasons why we chose this hypervisor over the others are listed below:

- Xvisor is a member of the monolithic type of hypervisors covering a wide range of peripheral and hardware drivers.
- It provides a wide range of functionalities as well as the ability to enable or disable them. Through that way, we can adjust and customize nya part of it in order to increase system performance.
- Xvisor is under continuous integration and development and an active support community is available to answer any questions or concerns.
- The Xvisor hypervisor supports a wide range of ARM-based systems and architectures from embedded devices to servers for data centre infrastructures.

A detailed analysis of the design architecture of Xvisor will help us to understand further about the installation and configuration process as well as how to integrate our novel approach and code contribution. As discussed above, Xvisor is a Type-1 monolithic hypervisor, also known as bare-metal, which offers a lightweight, portable, flexible virtualization layer implementation. Figure 6.3 shows a high-level architecture of the Xvisor infrastructure. It supports both ARM and x86 CPU architectures, covering a huge list of board models, so for the purpose of our experiment we can build, configure and install Xvisor on an RPI 3 board. Xvisor follows the same general guidelines and principles that most hypervisors follow and use, by referring to virtual machines as a Guest instance or Guest environment. On Xvisor any process running on the background is represented by a virtual CPU instance. In order to separate vCPUs assigned to a Guest instance from those that are used by the system, Xvisor categorises virtual CPUs of the system as Normal and Orphan vCPUs respectively. Normal vCPUs are referred to the number of vCPUs that are used by a Guest instance, compared to Orphan vCPUs that are used by internal and background processes necessary for hypervisor functionality. Like on any system, a CPU is running on two modes, the privileged or unprivileged mode. Normal vCPUs run in User (unprivileged) mode since they are used by Guest machines while Orphan vCPUs which run in Supervisor (privileged) mode and are controlled by the hypervisor itself.

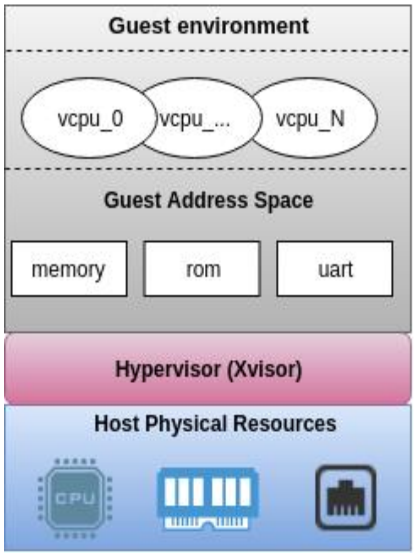


Figure 6-3 Xvisor high level system architecture

The following Figure 6.4 gives a high-level overview of the Guest architecture design as it resides in an Xvisor infrastructure. All the components dedicated to a Guest machine are memory mapped into smaller memory portions, called regions and each region is a sequence of memory addresses reserved in physical memory. That gives great flexibility in controlling, accessing, and manipulating memory and peripherals via memory mapped functions.

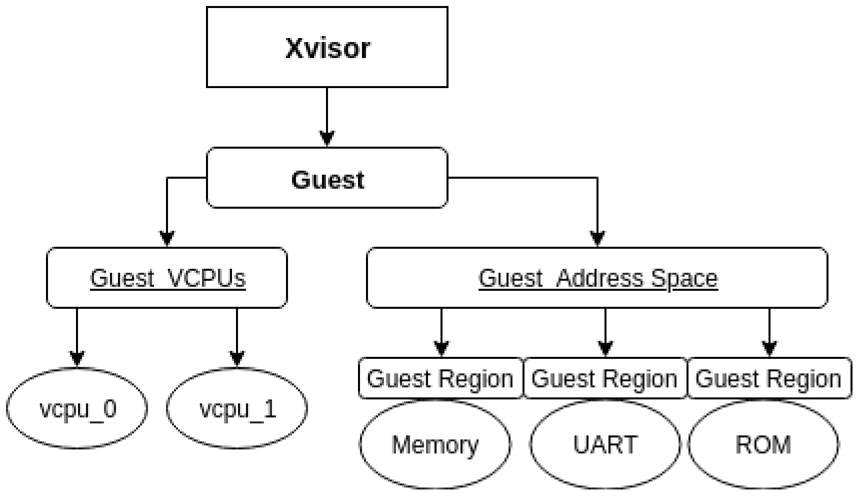


Figure 6-4 Xvisor Guest Logical Architecture

The content of a vCPU architecture consists of two groups of registers, the Arch_registers and the Arch_private as shown in Figure 6.5 The Arch_registers contain the registers that are updated by the processor itself, running in User mode, such as general-purpose registers and status flags. Both Normal and Orphan vCPUs have their own copy of Arch_registers. On the other hand, Arch_private registers are the group that a vCPU accesses when running in supervisor mode such as when a normal vCPU tries to read/write to those, and an exception is raised. Orphan vCPUs do not require an Arch_private group of registers since it always runs in Supervisor mode.

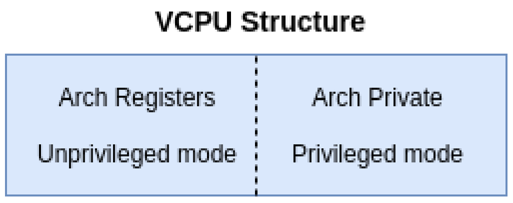


Figure 6-5 Xvisor vCPU structure

Xvisor also provides an extensive list of debugging commands which allow us to confirm the normal operation of the system and to determine and extract information from the hardware components. Later we will explain some of those commands as part of our implementation to initiate a CPU live migration process.

Xvisor is under continuous development, where more and more features are enabled over time. The version of Xvisor that was available at the time of this thesis (v.0.2.11), does not for instance fully support a network stack which makes migration of resources via an Ethernet network a more challenging task.

Although, Xvisor makes use of the open source LwIP TCP/IP stack, not all the API calls are ported and configured to support all kinds of platforms and network drivers. Taking into consideration those limitations, our implementation makes use of a UDP communication tunnel between the two end nodes.

6.3.4 Introduction of Raspberry Pi

RPIs have become well known in both the business industry and research community as an increasing number of use cases try to discover the capabilities of RPIs as both IoT devices and edge systems. Therefore, for our implementation, we chose to use an RPI platform due to low cost, energy efficiency and because is also supported by the Xvisor hypervisor.

A Raspberry Pi (RPi) is a small footprint desktop computer in a pocket-size. It is therefore a cheap solution that is finding application in many projects in both academia and industry, from sensors up to Edge gateways for processing aggregate data collected by a variety of IoT devices. Due to its small size it is a highly portable device that can be installed almost anywhere.

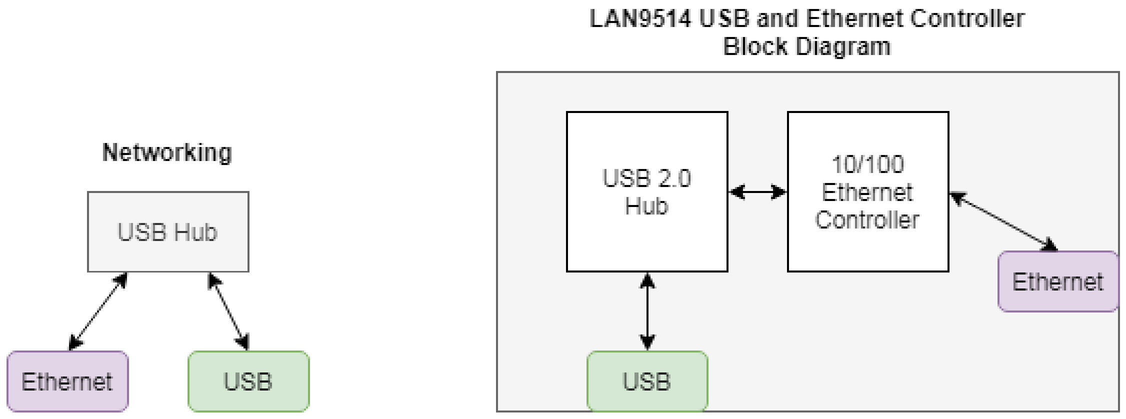
All generations of RPIs are developed with ARM architecture processors. The first-generation RPI (Pi 1) used ARMv6 while RPI 2 started using the ARMv7 cortex family processor. Both the latest RPI 3 and 4 generations come with ARMv8 Cortex family processors, powerful enough to fit on server infrastructures and support high-performance workload. The following table 3 gives a high-level overview of the different generations of RPIs including the supported Broadcom SoC and ARM processor version.

RPI generation	**Broadcom SoC**	**ARM processor**
Raspberry Pi Zero	BCM2835	ARMv6
Raspberry Pi 1	BCM2835	ARMv6
Raspberry Pi 2	BCM2836	ARMv7
Raspberry Pi 3	BCM2837	ARMv8
Raspberry Pi 4	BCM2711	ARMv8

Table 3 Overview of the available generations of RPIs

An important detail to mention here is that USB and Ethernet ports on an RPI board make use of the same bus system internally. As the following block diagram shows in Figure 6.6, the data exchanged through those ports pass through the USB hub. So, there is a generic controller of a LANxxxx type serving requests and further configuration for both USB and Ethernet ports. For example, on RPI 3 where our implementation is based, a LAN9514 controller is used. As the following block diagram shows, the

Ethernet PHY port is communicating with the Ethernet controller where it communicates with the USB Hub interface. Therefore, developing or porting an existing network stack on RPI is more challenging since specific configuration needs to be done in order to make use of the USB bus. Furthermore, a slight bottleneck may occur since data travels through the same bus system in order to serve both networking and USB requests. We will therefore need to consider this design information during the hypervisor configuration in order to enable networking on an RPI.



Networking
USB Hub
Ethernet
USB
LAN9514 USB and Ethernet Controller Block Diagram
USB 2.0 Hub
10/100 Ethernet Controller
Ethernet
USB

Figure 6-6 Block diagram of RPI networking infrastructure

6.4 Components of our Live migration Scheme

As we mentioned, currently there is no other similar work proposing a live migration scheme on ARM Bare metal instances. Creating a universal live migration scheme on ARM is very challenging and demanding. Therefore, for our proof of concept we decided to design, configure, and apply our live migration scheme based on the ARMv8 architecture focusing on CPU state which is the core point of difference between ARM and x86 architectures since the remaining parts that we will see and explain in this section at some points are covered well from existing works in the research community.

This section describes the main elements which are essential to the reconstruction of the state of a system during and after the execution of a Bare metal live migration as well as the techniques and methodologies which we need to adopt in order to achieve that outcome.

As we have described in Section 4.1.1, the same core elements that take place during a VM live migration process, take place in a Bare metal live migration too. The key difference is the lack of virtualization, meaning that instead of taking the virtual version of the system, we migrate the actual physical state. However, based on the design decisions described in the previous section, the actions of preservation and migration of the state in our case will be handled by a thin hypervisor layer based on Xvisor. Through that layer we can gain access to the memory, CPU and peripheral state.

On virtualized environments during a live migration the main components that take part during the process are transferred in the following order, memory, cpu and last but not least the peripheral connections. However, a Bare metal live migration scheme should adopts a slightly different approach by transferring first the memory state in an iterative manner including peripheral connections and then the CPU state. That's because on ARM architectures, peripherals are accessible through a memory mapped I/O methodology. So, by transferring the memory state we perform part of the peripheral devices state at the same time. Moreover, during the migration of the CPU state we make use of some portion of memory in order to store the values which we later use to fill in the data part of the network packet. In that way we keep intact the memory state.

As we outlined above and explain further in Section 6.4.1, Xvisor performs a mapping of the Guest environment to the physical memory space, dividing it into several regions where each region corresponds to a resource making the manageability and configuration easier. Additionally, Xvisor creates virtual CPU cores which are allocated to the Guest user and the various services running on the system. So, by capturing the state of the virtual core, our system can effectively represent the state of the Guest system at each specific time as we explain further in Section 6.4.3. Finally, I/O connections with peripheral devices and resources are performed through a memory mapped method. Devices and resources are mapped to a pre-defined range of physical memory and Xvisor is able to access those addresses using simple read/ write commands like treats memory requests as we show in Section 6.4.2.

The thesis does not focus and bother with the migration of the storage elements. This is not a concern since modern implementations allow a shared storage infrastructure accessible through the network such as SAN and NAS which both the source and destination systems can access at any time.

6.4.1 Capturing memory

Memory migration is a well-studied subject in the research and academic community [5, 114, 268] finding implementation through the pre-copy methodology, as the most efficient choice as explained in section 4.1.1. During the implementation of a pre-copy, memory pages are transferred from source to destination in an iterative format. Applying a memory migration from system to system is highly dependent on three factors, the total amount of supported physical memory, the physical addressing scheme of the platform, and the corresponding memory management system of the hypervisor layer. In this section we will outline the memory migration process for our Bare metal system using an XVisor hypervisor running on RPI 3 boards.

Each platform hardware specification defines the address memory space as well as the subsections that are divided and assigned to various internal resources. The RPI hardware specification defines the address memory space where physical memory and I/O memory mapped peripheral devices are accessible. Based on those specifications, the Xvisor hypervisor is able to read/ write to memory using the pre-defined memory addresses and communicate with the I/O peripheral devices.

In general, a hypervisor manages the allocation of resources in a Guest machine. In the same way, Xvisor allocates and maps a "Guest address space" for each Guest machine. A Guest address space is divided into several regions where each region belongs to memory and the various I/O peripherals such as the Programmable Interrupt Controller (PIC), Universal Asynchronous Receiver/Transmitter (UART) or a liquid-crystal display (LCD)). Additionally, each Guest region has a unique physical address and size that is mapped direct to the physical memory of the host. That mapping relationship is illustrated in Figure 6.7. The memory capacity, as well as the addressing scheme on a Raspberry Pi platform, differs from one version to another. On a RPI 3 model the memory capacity is 1 GB and the physical addressing range goes from 0x000000 to 0x3FFFFFFFF, while the address range from 0x3F0000000 to 0x3FFFFFFF is allocated and mapped to I/O peripherals.

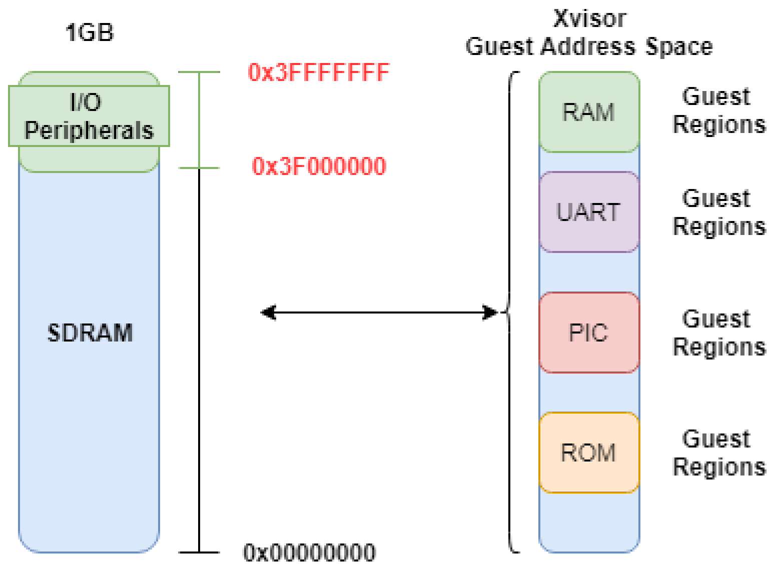


Figure 6-7 Xvisor memory address scheme mapping on RPI physical addresses

One of the advantages of the ARM architecture is that all peripheral bus addresses are mapped to physical memory. So, accessing memory-mapped addresses in physical memory tends to access the bus addresses and through this relationship we are able to access the peripherals. Therefore, through a memory migration process on ARM systems, we can facilitate the migration of two components in one, both the system memory and peripheral connections.

6.4.2 Capturing peripherals

Generally, there are two methods of communication between hardware peripheral devices and the CPU, using either a memory-mapped I/O (MMIO) or port-mapped I/O (PMIO) method. In contrast to the x86 systems architecture, most ARM systems support a memory mapped I/O methodology. Peripheral I/O device registers are mapped or linked to a predefined memory range of addresses, so, an I/O device can write to or read from memory to set or get information and data. In that way, I/O devices are accessible just like any other memory address. Also, when an interrupt comes from a peripheral, the CPU accesses that device for reading /writing purposes through a memory location at a particular address.

Memory-Mapped Input Output (MMIO) is the process that peripheral hardware devices use to interact by reading from and writing to dedicated, predefined memory addresses. ARM based systems make use of memory-mapped I/O (MMIO) process in order to interact with hardware peripheral devices by reading from and writing to predefined memory addresses. Each peripheral device is described by a range or group of memory registers which requires special access through a memory map. Access to a peripheral device occurs through an offset from the peripheral base address which is defined by the SoC of a system. Through that memory-based mapping architecture all peripheral hardware devices can be described by an offset from the Peripheral Base Address.

So, for example on RPIs boards there are three types of addressing scheme, the bus addresses, the physical addresses, and the virtual addresses as illustrated in Figure 6.8. All three addressing schemes provide a way of communication between software and the peripherals. A software package when it tries to access peripherals can do that by using either the virtual addresses or through the use of the Direct Memory Address (DMA) controller where it must use the bus addresses, while in some cases where the software has the efficient permissions and privileges to access the RAM directly, then it must use the physical addresses.

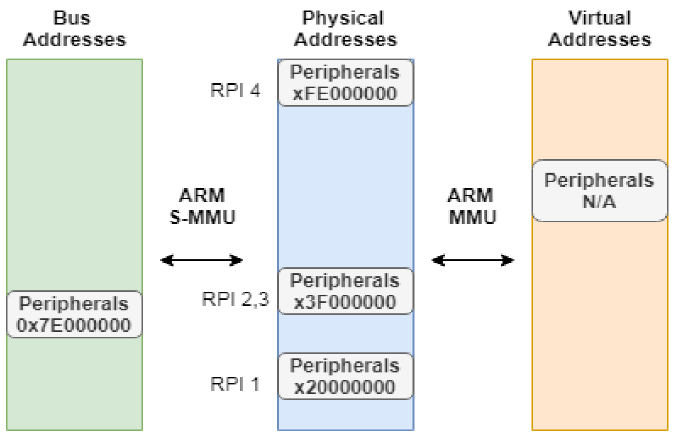


Figure 6-8 Example of RPIs addressing map diagram

Performing a live migration of peripheral device state in this case can be easily performed by continuously monitoring access to the range of memory addresses where I/O peripheral registers are mapped. On ARM-based systems, the memory-mapped I/O methodology is accomplished using the I/O memory management unit (IOMMU) and the Generic Interrupt Controller (GIC). As Figure 6.9 illustrates, ARM systems are enhanced with two separate memory management units, an I/O and the traditional MMU. The difference between those two systems is that the first one controls the translation, access, and communication of the incoming requests from a DMA controller to the physical memory. IOMMU is responsible for performing the translation and protection from unauthorised access to a specific memory page or memory region that is assigned and dedicated to a peripheral device, adding an extra layer of security while the traditional MMU translates the virtual address of a CPU to physical memory.

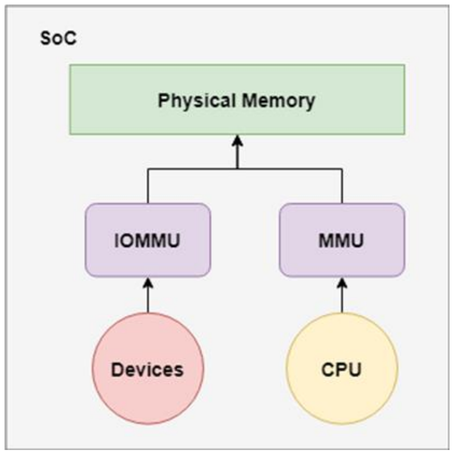


Figure 6-9 ARM memory management unit architecture for devices and CPU

The ARM System Memory Management Unit architecture (SMMU) defines the IOMMU architecture that is implemented on ARM processors to control the access and configuration of the hardware devices. Figure 6.10 below shows the logical parts of the SMMU architecture. The flow of communications among the client devices and the SMMU is that a memory access is requested by a client device which is raised to the SMMU. The latter is responsible for performing an address translation for that request. It then accesses the main memory on behalf of the client device.

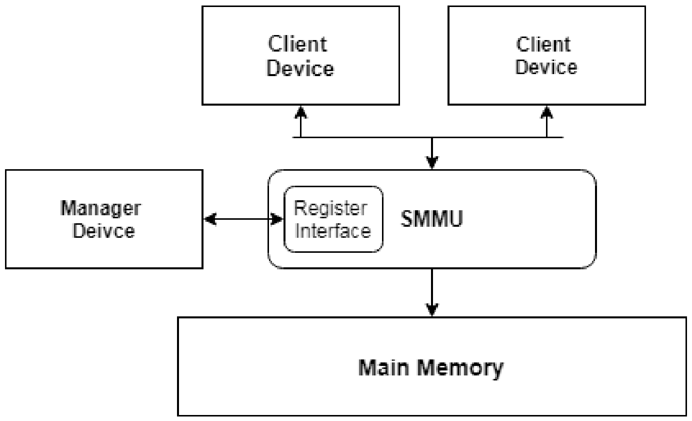


Figure 6-10 ARM SMMU system architecture

The ARM Generic Interrupt Controller (GIC) is part of the processor architecture. There are several versions of GIC, and the most recent processors are enhanced with the latest version of GICv3 and GICv4. The GIC is a controller that handles all the IRQ interrupts that are coming from the peripheral devices which can raise such an interrupt. A peripheral device sends an IRQ interrupt to the GIC. Then the GIC is able to serve and forward that interrupt to the connected cores. An I/O peripheral device creates IRQ interrupts which flow through a Distributor to a CPU Interface that is responsible for handling that type of interrupt. Then the CPU Interface forwards the received IRQ interrupt to CPU cores. In the GIC architecture, Interrupts can be identified by using ID number knowns as INTIDs.

The GIC architecture consists of the following set of logical components as shown in the diagram in Figure 6.11:

- Distributor:
  The Distributor is the core part of the GIC that handles all the sources of interrupts and keeps the routing configurations. There are several categories of sources like:
  - LPI Locality-specific Peripheral Interrupt (LPI)

    - PPI Private Peripheral Interrupt (PPI)
    - SPI Shared Peripheral Interrupt (SPI)
    - Software Generated Interrupt (SGI)
- Redistributor:

  Provides the configuration settings for interrupt sources like PPIs and SGIs.
- CPU Interface:

  The CPU Interface is responsible for forwarding and delivering the IRQ interrupt to the available core.
- Interrupt ID:

  When an I/O peripheral device sends an interrupt signal to the GIC, the GIC identifies and knows the source of the interrupt based on a unique identification number known as Interrupt ID.

Knowing the architecture and internal components of the GIC controller is valuable in the case of a peripheral I/O migration state. An interrupt is raised each time when the software needs to communicate and access a peripheral device. Different kinds and types of interrupts are handled and served by separate CPU interfaces. Therefore, when a change of device is needed to be done to a predetermined status, that change can be triggered through the use of the proper GIC register linked to that type of interrupt.

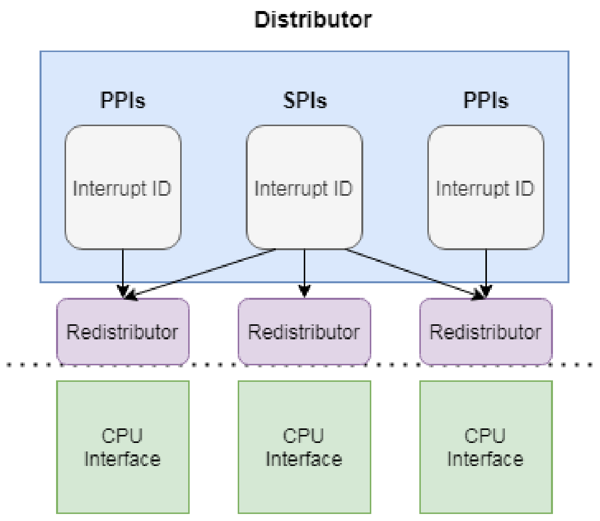


Figure 6-11 ARM GIC architecture diagram

6.4.3 Capturing Essential CPU state

A processor consists of several components including the registers, which are high-speed temporary locations that store processor data and instructions. Different types, kinds, and architectures of processors contain different numbers and configurations of registers which are used for different purposes, so there is no simple approach to capturing this information. Capturing a CPU state is a very challenging task especially on ARM based Bare metal instances, where no implementation like VMCS is available to encapsulate Guest CPU state into a data structure and easily migrate and adjust to another system as covered in Chapter 3. Furthermore, to the best of our knowledge, there is not an official definition that defines an “essential CPU state” that it can be used to achieve a reconstruction of the initial Guest state through a migration process from system A to system B.

Capturing CPU state is highly dependent on the processor's architecture, meaning that the specific processor's architecture defines which registers need to be preserved. Based on that, a hypervisor needs to understand that list in order to be able to read, capture and preserve them during the migration process. So, in our case we need to preserve the right group of registers based on the ARMv8 architecture as supported by the RPI 3 model.

Firtst, we need to define and explain from our point of view what we is considered as the essential CPU state. In our scheme as essential CPU state is considered to be the group of the general purpose registers that are used and take part during execution of an application as well as the group of registers that describe the processor’s configuration status at that specific time. Although ARM documentation suggests that the following list of elements needs to be saved and restored during a content switching process, this has not been proven by the research community that could also be sufficient to perform a CPU state migration.

- General purpose registers X0-X30
- Advanced Floating registers V0-V31
- Status registers
- TTBR0_EL1 and TTBR0
- Thread Process ID Registers
- and Address Space ID (ASID)

Since it is now common for a processor to have multiple cores, Xvisor assigns one or more physical cores to a Guest machine that are defined as virtual cores from Xvisor's architecture perspective. Each core's execution state is mainly described by three subgroups of registers which are a group of general-purpose registers, a group of special purpose registers, and processor’s state (PSTATE). On the ARMv8 architecture in order to accomplish a CPU state migration the following group of registers need to be preserved and migrated, 31 general purpose registers, and PSTATE, as listed in Figure 6.12. In detail, this includes 31 general purpose registers (X0-X30), 4 Stack point registers (SP_ELn), 3 exception link registers (ELR_ELn), and 3 saved processor state registers (SPSR_ELn). The functionality and purpose of the registers is described in Sections 5.4.2 and 5.4.3. Furthermore, while the PSTATE can be considered as a register, it actually represents a group of registers and flags that preserve the current process state.

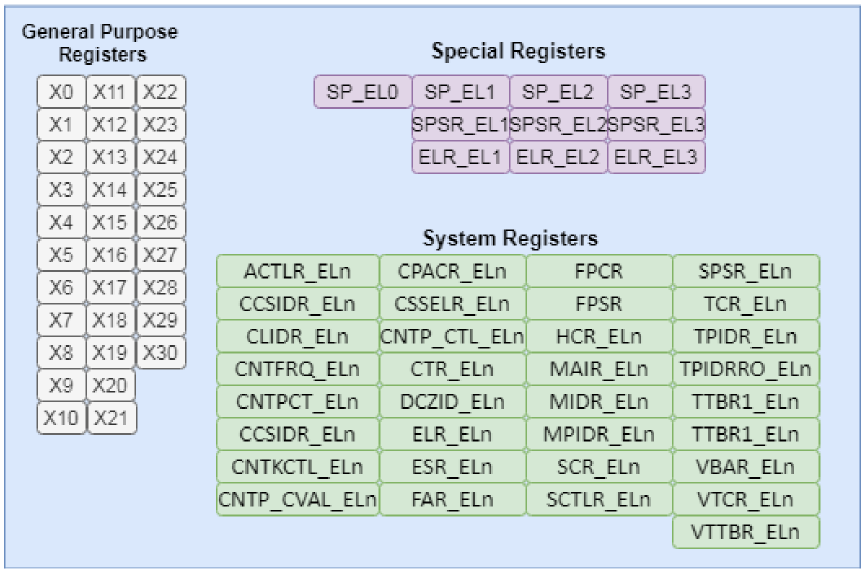


Figure 6-12 List of registers compose a CPU state

It is very important from a performance perspective that the CPU state migration be limited to the minimal necessary number of registers. It is normal that the higher the number of the migrated registers, the longer the CPU migration time which in turn affects and increases the total migration time. ARM processor has over 100 registers most of them are system registers giving a global picture of the state of the processor. Processors typically perform several content switching processes between tasks and processes running on a system. Exactly what registers need to be saved and restored varies based on the operating system and the architecture. Some of the registers that take part in a content switch process are the general-purpose registers, some status registers and some translation table registers. Having that in mind, our belief is that working with a single Guest environment does not necessitate the perseveration of all registers but only those which describes the content state of the Guest user space, minimizing hypervisor's overhead.

6.5 Design Implementation Phases

In this section, we introduce a design that covers the execution of a CPU state migration based on the ARM architecture, the design works on both ARMv7 and ARMv8 architectures however, our testing is based on ARMv8. Our proposed model is divided into four phases, the configuration phase, the reconnaissance phase, the migration phase and the confirmation phase. Starting with the configuration phase, this includes any prerequisite configuration in order to prepare the underlying infrastructure and resources where the migration process will be implemented, such as hypervisor's layer settings and networking infrastructure configuration details. This is followed by the reconnaissance phase, which describes the information gathering process related to crucial details and state that is needed in order to execute and trigger the migration process, such as the numbers and IDs of the vCPUs. Then the migration phase begins, which consists of retrieving, storing, and transferring vCPU registers at the source, and receiving and adopting the data at the destination. The final phase of the design model is the confirmation phase, where verification messages give feedback and confirm the successful completion of the migration process.

This model can find application on any ARM based platform that meets the following conditions:

- A hypervisor infrastructure that provides a defined structure of a vCPU.
- A hypervisor infrastructure that includes hard-coding the list of architecture-based registers.

6.5.1 Configuration Phase

We now explain how to adopt these phases on an infrastructure where Xvisor is being used. Starting with the configuration phase, the Xvisor hypervisor first needs to be loaded on both the source and destination systems before we deploy anything else since the creation of the Guest OS and the migration process depends on the existence of it. The installation process of Xvisor is based on the pre-existence of the U-boot bootloader which will load the kernel and bring up the OS. In our case, U-boot helps us to load the Xvisor kernel with the hardware description of the underlying board as well as the Linux kernel of the Guest instance. During the boot process of Xvisor, it creates the description plan of the Guest machine by default, allocating the specified number of vCPUs and memory regions. However, the Guest machine does not get created automatically so we need to trigger and execute the Guest machine first in order to start it running.

Before any state can be transferred from source to destination, a communication channel also needs to be set up between the two nodes. So, the networking settings must be applied in a manner where both servers belong on the same network and are able to communicate with each other. Furthermore, Xvisor gives us the capability and flexibility to define the number of vCPUs that we want to assign on a Guest. By default, two vCPUs are assigned to a Guest instance. We can modify the number of the vCPUs either during the build process of the source code of Xvisor or before we start the execution of the Guest instance. Once the number of the vCPUs is defined, the Guest machine starts on the source system. Part of the configuration process is to configure and modify the live migration functions, by adjusting the IP addresses and port numbers of the source and destination systems that we configure for the live migration process to use.

6.5.2 Reconnaissance Phase

Once the configuration phase is complete, the reconnaissance phase can begin. The execution of the live migration process requires some information to be given by the user. This includes the id number of the vCPUs and the range of the Guest address space that contains all the memory blocks of each Guest region. This information is given as input arguments to each function (memory migration, CPU migration). We can easily gather that information by calling some of the debugging commands that the architecture of Xvisor offers to us. Potentially, going forward, that process could be automated by integrating the information gathering process as part of the source code. Xvisor then divides the vCPUs into two categories, normal and orphans, where normal are referred to vCPUs attached and used by the Guest machine. Therefore, finding the ID numbers of the Guest vCPUs can be done by listing the normal vCPUs. Taking and passing that information to those functions can be resolved internally through the source code.

6.5.3 Migration Phase

In general, during the performance of a live migration process, the three major components that are transferred from source to the destination server are the CPU state, the memory and the peripheral connections. The architectural design of Xvisor allows us to migrate the state of the peripheral devices at the same time through a memory migration, since a memory mapping scheme is used in ARM systems. Based on that, the migration phase can be completed by calling two functions. A function that takes as arguments the id numbers of the vCPUs and performs a CPU state migration, and a function that takes the memory address range of the Guest regions in order to perform a memory migration. Each of those functions first creates a TCP or UDP session based on customer preferences, then captures and transmits that state to the destination server. Transmission of the state can occur through either a UDP or TCP session. UDP is preferred in the case that we desire higher performance, while TCP offers trustworthy migration. In each case, the appropriate APIs are called in order to set up the communication channel and then send and receive the data between the source and destination servers.

The destination server starts by waiting in the listening mode by calling a “receive state” function where it is waiting for incoming connections and ready to receive the data (state) from the source. At first, the memory state is received and adapted to the destination Guest address space, followed by the CPU state. Following this, once the destination server has received that state, it adapts the values and creates the Guest machine. After the completion of those stages and once we have confirmed that the Guest machine at the destination server is booted properly, the Guest machine at the source machine is ready to be deleted.

The live migration scheme outlined here works with the latest ARM versions such as ARMv7 and ARMv8 which both support CPU virtualization extensions. Although on ARMv7 version virtualization features come as part of a distinct CPU mode, the HYP mode, on ARMv8 it is enhanced as part of the ISA having a dedicated execution layer and registers.

6.5.4 Confirmation Phase

Once the migration phase is completed, the system needs to confirm and verify the successful receipt and integrity of the shipped data from source to destination. In a real use case scenario, the migration phase either will succeed and the state will resume on the destination, or it will fail, returning an execution error message, so the confirmation phase is eliminated. In order to increase the integrity and reliability of the migration phase a UDP protocol is used while the CPU state is migrated through a dedicated Ethernet connection between the source and destination. If the state has been altered during the migration phase the migration will fail protecting from a corrupted state to resume on the destination. In the case where the migration state fails the state continues running on source with no issue.

Currently, our experimental scheme makes use of the UDP communication and transportation protocol that lacks reliability. Therefore, for proof of concept of our design and in order to verify the successful operation and integrity of the migrated state, during the experimental process we make use of the network traffic analyser tool that captures the data as it reaches the destination. During the migration, the source keeps a copy of the migrated data while the destination prints the received data to the standard output. In that way, through a cross-check we can confirm that the right fields and values of data reach the destination system. Furthermore, through the network analyser tool (Wireshark) we check that there is no modification to the data after the reception by the destination system or at the hypervisor layer.

6.6 Summary

In this chapter we observed the tremendous interesting in the adoption of RPIs at the edge of a network. Inspired by this trend we explored the development and support of live migration process on ARM based systems. We discussed the core differences between the ARMv7 and ARMv8 processor versions as well as the reasons why we chose to work our live migration scheme on the latest ARMv8 version. Furthermore, we discussed the design decisions and requirements for the implementation of a live migration scheme on ARM based systems. The need of a monolithic, lightweight hypervisor led us to choose Xvisor as our underlying layer and develop our implementation having as primary targets RPIs boards. Lastly, we analysed the parts and phases of our live migration scheme as well as the components that take part in the migration process.

In the following Chapter 7, we work on the implementation of our live migration scheme on Xvisor explaining all the enhancements and modifications we need to apply in order to support live migration between two systems.

7 Implementation

7.1 Introduction

This chapter gives a detailed analysis and explanation of our source code implementation for the support of a CPU state live migration as this finds application on ARM based instances through the Xvisor hypervisor. As we introduced in Chapters 5 and 6, ARM systems such as Raspberry PIs attract huge interest from the research and academic community on exploring them as Edge Computing deployment solutions. However, there is a lack of availability for an ARM based live migration scheme targeting Bare metal instances such as single RPI boards. Therefore, our novel implementation assists in the performance of a CPU live migration between RPI boards that could be installed and serve as edge nodes offering high service availability and fault tolerance. This chapter covers the core functionalities of the establishment and execution of a CPU live migration process among two ARM based systems as illustrated in Figure 7.1. We do not cover a full Bare metal state migration including memory, storage and peripherals but we focus on the CPU state migration part. This is because the development of a CPU live migration framework is the most challenging part of all the required components that take part in a Bare metal live migration process because of the dependencies and peculiarities of the processor's architecture. The remaining components such as memory, storage and peripherals can largely be based on existing work. For example, a storage migration process is of limited value since network-based shared topologies such as SAN and NAS are increasingly being adopted, which both the source and destination systems can access at any time. Additionally, memory state migration processes have been studied extensively in the literature, where proposed solutions and implementations find application on both x86 and ARM architectures. As we explained in the previous chapter, the performance of a memory migration on ARM based systems, also completes a peripheral migration state since a memory mapped methodology is adopted. Therefore, the only challenging part with no current practical implementation on ARM systems is the application of a CPU state migration where our novelty is focussed.

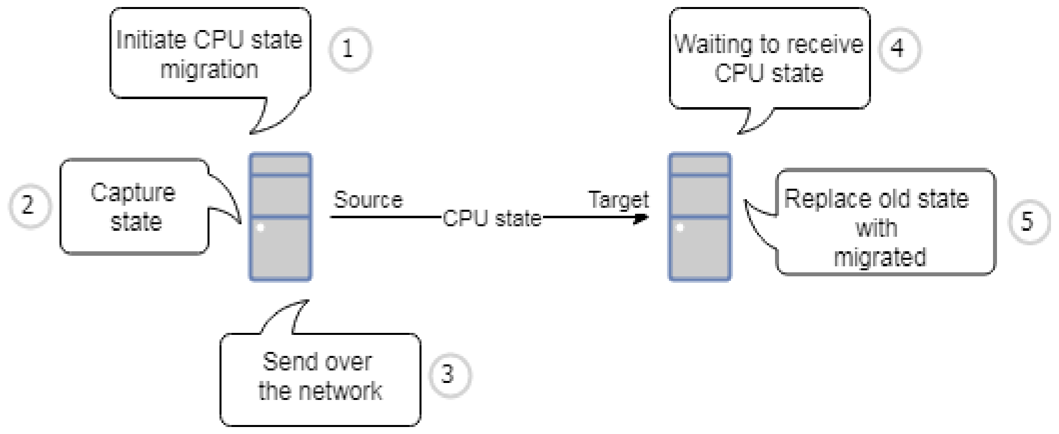

Initiate CPU state migration
1
Waiting to receive CPU state
4
2
Capture state
Source
CPU state
Target
Replace old state with migrated
5
Send over the network
3

Figure 7-1 CPU live migration implementation steps

Our implementation covers four main phases where each of those phases consists of a few additional steps. Those phases are:

- Phase 1: Networking configuration
- Phase 2: Retrieve CPU state (source)
- Phase 3: Receive CPU state (destination)
- Phase 4: Adoption of CPU state (destination)

All these phases are part of the design approach as introduced and explained earlier in Section 6.5. More specifically, phase 1 is part of the configuration phase, phase 2 is part of the reconnaissance phase, phase 3 is part of the migration phase while phase 4 is a mixture of both migration and confirmation phases. Phase 1 covers the configuration related to the underlying networking infrastructure and the support of a ported TPC/IP protocol. Xvisor gives the ability and support for a TCP/IP network stack, however this feature is not enabled by default. Therefore, at first, we need to make the appropriate modifications in order to support a network connection between the source and destination and make the necessary API calls for the construction of a UDP packet.

Phase 2 explains the source code for retrieving the CPU registers values from the source and all the dependencies related to this task such as the way that Xvisor communicates and interacts with the hardware. Phase 3 analyses the source code at the destination host that receives the migrated list of registers. In this phase we present a sample of the receive function that could work on Xvisor in a future release, as well as the sample of code that we used in our proof-of-concept experiment lab.

Finally, phase 4 gives a sample of the code which performs adoption of the received registers by the destination host.

In order to implement a CPU live migration functionality as part of the Xvisor, first we need to develop and integrate the appropriate menu including the options for the migration as part of the existing Xvisor's menu. For convenience, we have integrated the list of the options as part of the existing "ping" menu as provided by Xvisor. However, we could easily create a separate menu, giving the name of our choice that performs the same functionality. This could be a task for future work.

Algorithm 1 describes the logic of our implementation. In line 7 we integrate the migration of the CPU registers from source to destination. Line 9 performs the adoption of CPU registers at the destination host while in line 11 the reception functionality starts waiting for the incoming state.

ALGORITHM 1: XVISOR PING MENU

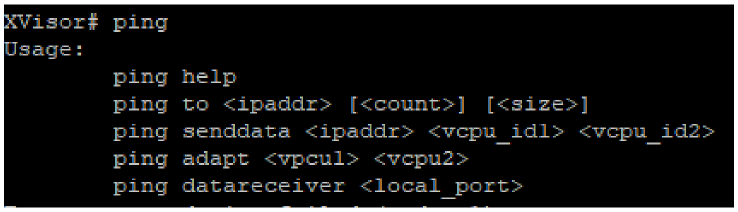
```
1:  Require: Installation of Xvisor hypervisor
2:  Function cmd_ping_usage(input parameters)
3:  If pass `help` then
4:      Show ping menu
5:  Else if pass `ping to` then
6:       Ping the address;
7:  Else if pass `ping senddata` then
8:       Execute cpu migration;
9:  Else if pass `ping adapt` then
10:      Replace cpu values;
11: Else if pass `ping datareceiver` then
12:      Receive cpu values;
13: End function
```

The code implementation of the menu with the options is presented in Figure 7.2 while Figure 7.3 shows the output of the menu as it appears to the user.

```
static void cmd_ping_usage(struct vmm_chardev *cdev)
{
	vmm_cprintf(cdev, "Usage: \n");
	vmm_cprintf(cdev, " ping help\n");
	vmm_cprintf(cdev, " ping to <ipaddr> [<count>] [<size>]\n");
```

```
        vmm_cprintf(cdev, " ping senddata <ipaddr> <vcpu_id1> <vcpu_id2>\n");
        vmm_cprintf(cdev, " ping adapt <vpcu1> <vcpu2>\n");
        vmm_cprintf(cdev, " ping datareceiver <local_port>\n");
}
```

Figure 7-2 Code integration for CPU migration into Ping menu



```
XVisor# ping
Usage:
        ping help
        ping to <ipaddr> [<count>] [<size>]
        ping senddata <ipaddr> <vcpu_id1> <vcpu_id2>
        ping adapt <vpcu1> <vcpu2>
        ping datareceiver <local_port>
```

Figure 7-3 List of the available sub-commands of the ping command

7.2 Networking configuration

Besides the menu integration, some modifications to the existing source code of Xvisor are required to support our implementation. Those prerequisites are mostly related to enabling some of the networking functionalities like the desired networking stack.

As we mentioned in Chapter 6, by default the Xvisor architecture supports two types of networking stacks, an internal network packet switching framework stack and a network stack for socket programming. The network stack for socket programming is optional and therefore by default is disabled. Xvisor is integrated with the open-source LwIP TCP/IP stack. So, in order to achieve, create and open a TCP/IP session between two systems running Xvisor, we need first to enable support for the available TCP/IP network stack as that is provided through LwIP.

Moreover, the supported drivers related to the Ethernet and the USB need to be ported and enabled since a connection via an Ethernet cable is supported by porting the relative drivers of the USB ports. This is mostly a requirement of the RPI's specifications and boards from other vendors may only need the use of the Ethernet drivers.

This is followed by the changes that need to be configured as part of the compilation and build phase of the Xvisor in order to include and make available the additional functionalities which are necessary for the support of our live migration scheme through the network. Someone who wants to use Xvisor on an RPI board and wants to perform a live migration by adopting our live migration scheme, needs to follow and make those changes in order to enable the necessary functionalities and options. Figure 7.4 shows the modifications that we need to perform in order to enable the support for the networking TCP/IP stack which by default as we mentioned before is disabled. We just replace the default option from “n” (no) to “y” (yes).

```
menu "Network Stack Options"
    depends on CONFIG_NET

config CONFIG_NET_STACK
    tristate "Network stack
support"
    depends on CONFIG_NET
    default y
```

Figure 7-4 Enabling Network Stack Options

Figures 7.5 and 7.6 show the source code modifications that need to be done in order to enable the supported drivers for the Ethernet end USB interfaces respectively as well as the appropriate controller. The name of the supported Ethernet and USB controller is based on the hardware. Based on the official documentation of RPI 3 the supported drivers of the controller are part of the SMSC95xx family.

```
menuconfig CONFIG_ETHERNET_DRIVERS
        tristate "Ethernet driver support"
        default y
        depends on CONFIG_NET_DEVICES
        help
                Select ethernet drivers
```

Figure 7-5 Enabling the support of Ethernet drivers

```
menuconfig CONFIG_USB_NET_DRIVERS
        tristate "USB network driver support"
        default y
        depends on CONFIG_USB && CONFIG_NET_DEVICES
        help
                Select USB network drivers.
```

```
if CONFIG_USB_NET_DRIVERS

config CONFIG_USB_SMSC95xx
        tristate "SMSC95xx driver"
        default y
        help
                USB SMSC95xx network device driver
Endif
```

Figure 7-6 Enabling the support of USB drivers

So, we have now enabled support for LwIP and properly configured the networking Ethernet interface. Furthermore, as we will see in the code that follows, we chose to make use of UDP protocol rather than using a connection-oriented session between the source and destination. One of the reasons is due to the version of Xvisor (v0.2.11) that was available during the time that this thesis took place, where support for TCP API calls is not fully implemented by the LwIP network stack. Furthermore, our network infrastructure is envisioned to be a private, isolated network consisting of only two users at this stage. Therefore, it can be considered reliable and secure enough that no other recipients exist to intercept or interfere with the migration process. By using UDP protocol we also do not incur additional latency or transmission delays since a smaller header is used in comparison to TCP and less configuration is needed on our side to create a UDP socket.

Let us now consider the specific client/server scenario, where the source host plays the role of the client while the destination host plays the role of the server by utilizing a Socket API. A socket API is a collection of API calls that enable the communication between applications programs. There are three types of sockets, a stream socket, a datagram socket, and a raw socket which is the one that we decided to make use of in our implementation. With a raw UDP socket, unlike TCP, the client does not need to form a connection with the server while the server just waits for datagrams to arrive. The following diagram in Figure 9.7 illustrates the socket API calls or methods in order to initiate and transfer data between source and destination. Primary functionalities of a socket API are the establishment of a connection, send and/or receive data and close a connection which are performed with the call of the methods as illustrated in Figure 7.7. At the source we call the list of functions as provided by the library of the LwIP stack such as lwip_socket(), lwip_bind(), lwip_sendto() and lwip_close() [238]. The lwip_socket() is the first function that we need to call, specifying the type of communication as well as the desired protocol family following by the lwip_bind() which binds the created socket to an interface and assigns an IP address while the lwip_sendto() function is used to transmit data through UDP while the lwip_close() terminates the socket. Similarly, on the destination host we make use of the API calls that a Unix socket provides to a Linux system [239]. The only difference is that on the destination host we use the recv() method which is used to receive data over a UDP.

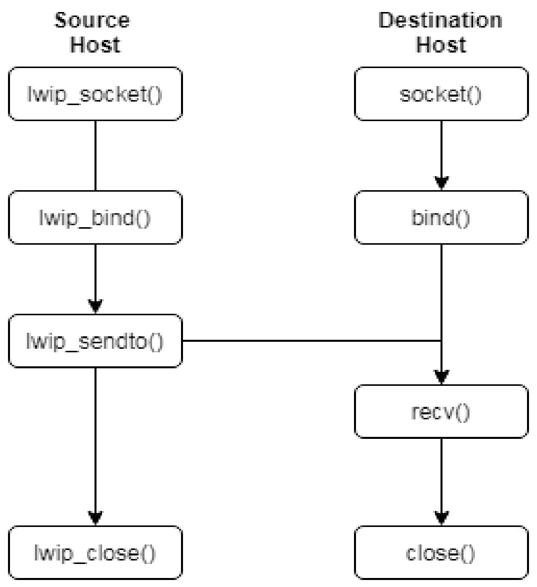


Figure 7-7 LwIP API calls setting up a UDP server – client channel

7.3 Retrieve CPU at source

To retrieve the CPU state of a Guest instance several sub-tasks need to be done which involves a deep understanding of the hypervisor layer implementation. Algorithm 2 provides the pseudo code of our approach.

ALGORITHM 2: TRANSMIT CPU REGISTERS WORKFLOW

```
1:    Require: Existence of Guest vCPUs
2:    Input: IDs of the vCPUs
3:    Function senddata(vCPU1, vCPU2)
4:    Set timer;
5:    If IDs of vCPUs are valid then
6:        For each vCPU do
```

```
7:            Read registers;
8:            Store registers in a data array;
9:            Create a UDP socket;
10:           Prepare packet;
11:           Load data from array into packet;
12:           Send packet to destination;
13:           Stop timer;
14:       end
15:   end
16:   End function
```

Through the execution of `senddata` function as defined in algorithm 2 the following actions are implemented:

- Identification of the mapped vCPUs
- Read and store the registers of each vCPU
- Creation of a UDP packet shipping the captured registers as data
- Send the formatted UDP packet

Retrieving the CPU state means capturing the values of the CPU registers in real time. On Xvisor, a vCPU is a virtual representation of a CPU so, in order to perform a state migration, we need to gain access, read and then store the values of the registers that describe a vCPU. However, we don't want to migrate the values of any vCPU, but only those that are assigned and used by the created Guest machine. Therefore, we need to understand how a Guest machine and vCPU structure are handled and represented by the Xvisor infrastructure code.

On Xvisor a Guest instance is represented by the following components as defined in the sample of code in Figure 7.8:

1. A global unique identification number (ID) in the case that more than one Guest instance exists.
2. A Guest Device Tree node configuration which is a data structure that maintains the configuration of that specific Guest instance.

3. A vCPU Count which is the number of the total vCPU instances which are attached to this Guest. By default, Xvisor assigns two vCPUs for each Guest, however, depending on the hardware specifications of the host system more than two vCPUs can be attached.
4. A vCPU List of the vCPU instances which are attached to this Guest
5. A Guest Address Space
6. An architecture dependent part of the vCPU which contains the state of the Guest machine. The architecture dependent part consists of the Arch Registers structure and the Arch Private as described earlier in Section 6.3.2.

```
struct vmm_Guest {

      /* General information */
      u32 id;
      struct vmm_devtree_node *node;
      /* VCPU instances belonging to this Guest */
      vmm_rwlock_t vcpu_lock;
      u32 vcpu_count;
      struct dlist vcpu_list;
      /* Guest address space */
      struct vmm_Guest_aspace aspace;
      /* Architecture specific context */
      void *arch_priv;
};
```

Figure 7-8 Code structure

The core part of a Guest machine and the key to our novel contribution is the structure of a vCPU. On Xvisor as Figure 7.9 shows, a vCPU structure consists of:

1. A global unique identification number (ID) in case more than a single vCPU is attached on a Guest instance.
2. A Device tree node data structure
3. A flag which indicates if that vCPU is Normal or Orphan. We pointed out the distinction between normal and orphan as defined by Xvisor architecture in Section 6.3.2.

4. Starting value of the Program Counter.
5. The size of the vCPU stack
6. The Host CPU which that vCPU is running.

```
struct vmm_vcpu {
      /* General information */
      u32 id;
      u32 subid;
      char name[VMM_FIELD_NAME_SIZE];
      struct vmm_devtree_node *node;
      bool is_normal;
      bool is_poweroff;
      struct vmm_Guest *Guest;
      /* Start PC and stack */
      virtual_addr_t start_pc;
      virtual_addr_t stack_va;
      virtual_size_t stack_sz;
      /* Architecture specific context */
      arch_regs_t regs;
      void *arch_priv;
      /* Virtual IRQ context */
      struct vmm_vcpu_irqs irqs;
};
```

Figure 7-9 Code structure defining a vCPU on Xvisor

The default configuration of the Xvisor assigns two vCPUs per Guest instance. If we want to increase the compute power and performance of the Guest instance, we can modify and change the default number of assigned vCPUs during the source code compilation of the Xvisor or through the console. If you want to add or delete a VCPU at boot-time then we need to edit the /boot.xscript file and increase or decrease vcpu_count value passed to "vfs guest_fdt_load" command. Else If we decide to add vCPU for existing Guest instance we need to execute the commands in Figure:

Delete vCPU # guest destroy guest0 # devtree node del /guests/guest0/vcpus/vcpu1 # guest create guest0 # vfs guest_load_list guest0 /images/arm64/virt-v8/nor_flash.list
Add vCPU # guest destroy guest0 # devtree node copy /guests/guest0/vcpus vcpu2 /guests/guest0/vcpus/vcpu1 # guest create guest0 # vfs guest_load_list guest0 /images/arm64/virt-v8/nor_flash.list

Figure 7-10 Increase or Decrease the number of the assigned vCPUs per Guest Instance

In the case that we increase or decrease the default number of the vCPUs additionally we need to adjust our configuration of the number of migrated vCPUs. For example, with three vCPUs, the function "senddata" as explained in Algorithm 2 needs to be done as follow: senddata(vCPU1, vCPU2, vCPU3)

To read and store the values of the registers, we first need to know the exact list of registers. Based on the processor architecture, Xvisor maintains hardcoded, architecture specific, lists of registers. The following Figures 7.10 and 7.11 show an example of the hardcoded definition of the core and general-purpose registers as defined on Xvisor for an ARMv8 and ARMv7 architecture respectively.

```
struct arch_regs {
      /* X0 - X29 */
      u64 gpr[CPU_GPR_COUNT];
      /* Link Register (or X30) */
      u64 lr;
      /* Stack Pointer */
      u64 sp;
      /* Program Counter */
      u64 pc;
      /* PState/CPSR */
      u64 pstate;
} __packed;
```

```
struct arch_regs {
      u32 sp_excp; /* Stack Pointer for
Exceptions */
      u32 cpsr; /* CPSR */
      u32 gpr[CPU_GPR_COUNT]; /* R0 - R12
*/
      u32 sp;     /* Stack Pointer */
      u32 lr;     /* Link Register */
      u32 pc;     /* Program Counter */
} __packed;
```

Figure 7-11 Structure of calling ARMv8 registers *Figure 7-12 Structure of calling ARMv7 registers*

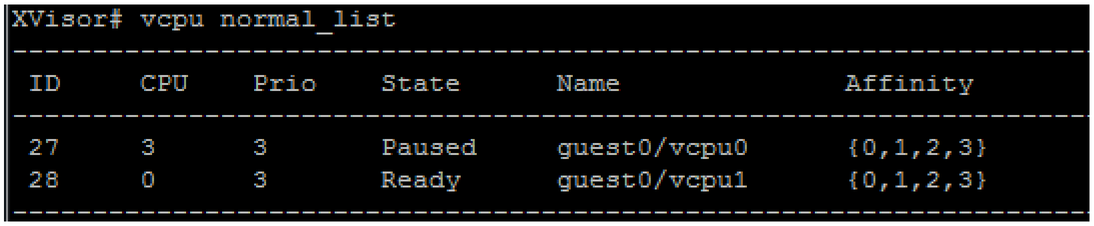
Access to both the architecture specific dependent and independent registers happens through the use of pointers as shown in the following Figure 7.12. Using the "regs" pointer we can access and read the values of the general-purpose registers while with the pointer named "archi_priv" we read the values of the registers as updated by more privileged exception levels.

```
#define arm_regs(vcpu)          (&((vcpu)->regs))
#define arm_priv(vcpu)          ((struct arm_priv *)((vcpu)->arch_priv))
#define arm_Guest_priv(Guest) ((struct arm_Guest_priv *)((Guest)->arch_priv))
```

Figure 7-13 Defining the pointers of the architecture registers

In our implementation we chose to make use of the ARMv8 architecture. So, in order to communicate with the CPU and read the values of the registers we need to use the naming conversion as defined in the arch_regs block as explained above in Figure 7.10. Similarly, in case we need to adjust the process to an ARMv7 architecture we just replace the name of the registers. For example, in ARMv7 the general-purpose registers are only thirteen while in order to access the current state we make use of the CPSR register instead of the PSTATE like we do on ARMv8.

Coming back to the "ping senddata" command, as we introduced in the menu panel, this takes three parameters, the destination host IP address and the IDs of the migrated vCPUs. By default, when a Guest instance is created, Xvisor assigns two vCPUs. We can identify the ID numbers that we need to pass in our command using the supported "vcpu normal_list" subcommand as shown in Figure 7.13. This command lists the currently available vCPUs that are assigned to a Guest instance.



```
XVisor# vcpu normal_list
--------------------------------------------------------------------------------
 ID      CPU     Prio     State        Name                 Affinity
--------------------------------------------------------------------------------
 27      3       3        Paused       guest0/vcpu0         {0,1,2,3}
 28      0       3        Ready        guest0/vcpu1         {0,1,2,3}
--------------------------------------------------------------------------------
```

Figure 7-14 Output of the vCPU normal list execution command

In the first part of the source code, we validate that those IDs of the given vCPUs exist. Afterwards, we pass those vCPUs to another function which can access, read and store the values of the registers into a data array.

The following section of code in Figure 7.14 performs those actions. With the help of the two pointers regs1 and regs2 we are accessing the named registers of the vCPU1 and vCPU2 respectively.

```
/*Put the values of register inside the array*/
   data_buffer[0] = regs1->sp;
   data_buffer[1] = regs1->lr;
   data_buffer[2] = regs1->pc;
   data_buffer[3] = (regs1->pstate & 0xffffffff);
   data_buffer[4] = regs2->sp;
   data_buffer[5] = regs2->lr;
   data_buffer[6] = regs2->pc;
   data_buffer[7] = (regs2->pstate & 0xffffffff);
```

Figure 7-15 Storing registers' values into a data structure

Additionally, we can expand the list of the registers we want to capture using the appropriate pointer and the right name of the register as that defined in the embedded hardcoded list.

Once we have stored the values of the registers in the data array we need to create and set a UDP socket giving the destination details. We have hardcoded the network configuration details for both the source and destination hosts at this stage, including the IP address and UDP ports as shown in Figure 7.15. Both source and destination hosts initiate a UDP socket, listening on port 6002 while we have declared the source IP address to be 192.168.0.10 and for the destination host to be 192.168.0.11.

```
#define SENDER_PORT            6002
#define RECEIVER_PORT          6002
#define SENDER_IP              "192.168.0.10"
#define RECEIVER_IP            "192.168.0.11"
```

Figure 7-16 Define globally Networking configuration details

Lastly, for performance evaluation, we make use of a timer that counts the overall time it takes to complete a CPU live migration process. The timer starts counting once the "ping_senddata" function is triggered until the data migration process is completed. The sample of code in Figure 7.16 describes the integrated part of the timer.

```
//Initial timer
u64 timer_stamp = 0;
u64 mult, start_tstamp, end_tstamp;
// Start timer
start_tstamp = vmm_timer_timestamp();
// Stop timer
end_tstamp = vmm_timer_timestamp();
// Print the total migration time
timer_stamp = (end_tstamp - start_tstamp);
vmm_cprintf(cdev, "timer_stamp = %"RPId64" nanoseconds\n", timer_stamp);
```

Figure 7-17 Setting up timer for performance evaluation

Appendix A-1 and A-2 present the integration of our source code for the performance of a CPU migration implementation. Once we have validated that the number of vCPUs exist and are valid, we pass the information to the “netstack send udpdata” nested function which is responsible for handling the data captured from the registers and transmit them through the network.

Let us next describe some of the core parts of that implementation in order to understand how it works and what it does. Once the “ping senddata” command is executed the “cmd_ping_senddata” function is triggered that first checks that the right number of parameters passed which is performed by the following sample of code in Figure 7.17.

```
if((argc < 1) || (argc > 3)) {
    cmd_ping_usage(cdev);
    return VMM_EFAIL;
 }
```

Figure 7-18 Passing arguments validation function

Then, the part of code in Figure 7.18 transforms the IP address of the destination host to the right format and converts the ID numbers of the vCPUs from a string to an integer.

```
str2ipaddr(ipaddr, argv[0]);
 id1 = atoi(argv[1]);
 id2 = atoi(argv[2]);
```

Figure 7-19 Conversion of IP address from string to integer format

Following this, we need to check that the passed numbers of vCPUs exist and have been created by Xvisor, a task which accomplished by the code described in Figure 7.19.

```
vcpu1 = vmm_manager_vcpu(id1);
        if (!vcpu1) {
                vmm_cprintf(cdev, "Failed to find vcpu\n");
                return VMM_EFAIL;
        }
vcpu2 = vmm_manager_vcpu(id2);
        if (!vcpu2) {
                vmm_cprintf(cdev, "Failed to find vcpu\n");
                return VMM_EFAIL;
        }
```

Figure 7-20 Validation process of the given vCPUs

Afterwards, we access and read the registers of each of the vCPUs using pointers as explained in the previous section while at the same time we store them into a data structure array as shown in Figure 7.20.

```
data_buffer[0] = regs1->sp;
data_buffer[1] = regs1->lr;
data_buffer[2] = regs1->pc;
data_buffer[3] = (regs1->pstate & 0xffffffff);
data_buffer[4] = regs2->sp;
data_buffer[5] = regs2->lr;
data_buffer[6] = regs2->pc;
data_buffer[7] = (regs2->pstate & 0xffffffff);
```

Figure 7-21 Store registers into a data array

Finally, once we have stored the values of the registers into the data array, appendix A-3 shows the code implementation that creates a UDP socket filling the data portion of the packet with the data from the array.

7.4 Source code Implementation – Receive CPU state at destination host

At the time when the implementation development and the practical experiments took place, Xvisor (v2.11), did not support Live migration or any kind of interaction between two distinct Xvisor systems. Therefore, for proof of concept of our implementation as a destination host, we made use of a Linux distro, called Raspbian in the first instance but subsequently implemented the functionality in Xvisor also.

Algorithm 3 presents the logic behind the receive functionality that operates on the destination host. Line 3 shows that the receive function takes as input parameters the UDP port as well as the number of migrated vCPUs. It creates and sets up a socket as shown in line 4 and 5 while once the data has reached the destination, we store them to a temporary array (line 8). The last step is to replace those registers with the native one of the destinations.

ALGORITHM 3: RECEIVE CPU REGISTERS WORKFLOW

```
1:   Require: Existence of Guest vCPUs
2:   Input: Data over network, IDs of the vCPUs
3:   Function receivedata(UDP_port,vCPU1,
     vCPU2)
4:   Create a UDP socket;
5:   Set socket in listening mode;
6:   If packet received then
7:       For each vCPU do
8:           Load registers to a data array;
9:           Replace/Adopt registers from data
             array to CPU registers
10:      end
11:  end
12:  End function
```

Appendix A-4 includes the block of code for the receiver side implementation based on the Raspbian operating system. We have developed a script in C that performs the following tasks: creates a UDP socket that listens for incoming data on the predefined port and once the data are received print them to the screen. In this way, we confirm that the migrated values of the registers from the source side match the received values at the destination.

Next, we briefly analyse and explain some of the functionalities of the above functionality.

- By calling the socket() function we create a Unix socket interface defining the protocol that it will support. In our case this will be a UDP socket as shown in Figure 7.21.

```
socket(AF_INET, SOCK_DGRAM, 0)
```

Figure 7-22 Creation of a network socket

- Figure 7.22 shows the configuration of the Ethernet interfaces with the IP address and port number of source and destination interfaces where data will be sent and received respectively.

```
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons(PORTSERVER);

clientaddr.sin_family = AF_INET;
clientaddr.sin_addr.s_addr = inet_addr("192.168.0.10");
clientaddr.sin_port = htons(PORTCLIENT);
```

Figure 7-23 Socket interface configuration

- The bind() function binds that socket to our local interface while the recvform() function is waiting for incoming data that it will store into a temporary array called "buf" as explained in the following Figure 7.23. Finally, we print all the elements of that array on the screen.

```
bind(s, (const struct sockaddr *)&serveraddr, sizeof(serveraddr)
recv_len = recvfrom(s, buf, BUFLEN, 0, (struct sockaddr *)&clientaddr, &slen)
```

Figure 7-24 Socket data reception function

This is a common script that could run on any Linux distribution system making use of Unix socket API calls. Through that script, we observe the ability of Xvisor to communicate with other systems and the ability to transmit vital, sensitive information about the hardware. Potentially, that could be useful in the case that we would like to keep a system state backup or system logs about the state of the hardware on a centralised server for future data analytics tasks.

With this work done however, we next need to develop and configure the receiver implementation on Xvisor, which poses a more challenging task because of the limitations on the supported TCP/IP network stack. As we mentioned in the previous section, from the list of the menu, the "ping datareceiver" command takes as arguments the UDP port where the data should be sent by the source. Afterwards, it creates a UDP socket as before using the API functions of the ported LwIP network stack as illustrated in Figure 7.24. In this implementation, both the source and destination hosts use the LwIP methods in comparison to the one that we have shown above where the destination host uses only the Unix API calls.

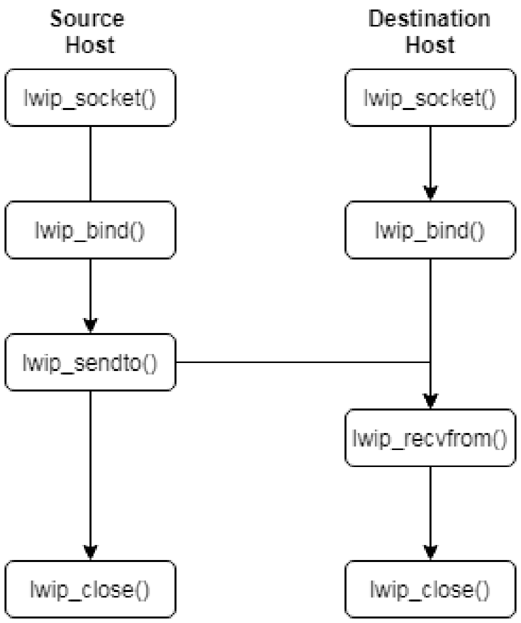


Figure 7-25 LwIP API calls setting a UDP client/server socket

In Figure 7.25, the block of code that performs the receiver operation as described above. We first perform input data validation and then we apply a type conversion that is necessary in order to process the information in the right format.

```
static int cmd_ping_datareceiver(struct vmm_chardev *cdev,
                     int argc, char **argv)
{
      void *arg;
      u8 port;
      vmm_printf("***---%s---***\n", __func__);
```

```
        vmm_printf("***---%s---***\n", argv[0]);
        if((argc < 1) || (argc > 2)) {
                cmd_ping_usage(cdev);
                return VMM_EFAIL;
        }
        port = atoi(argv[0]);
        netstack_receive_udpdata(port);
        return VMM_OK;
}
```

Figure 7-26 Main code of the data receiver function

Appendix A-5 describes the nested functionality which is part of the main "cmd ping datareceiver" function above. The "netstack receive udpdata()" function handles the creation and reception of incoming data at the preconfigured port.

The code of the receiver side on Xvisor has many common parts with the one of the sender as we explained in Figure 7.17. Similarly on the receiver side, we create a raw type socket and we assign an IP address to it as well as a port while we pre-define the IP address of the sender making available that socket to that specific sender. Once the socket is ready it listens for incoming data. Since the data is transferred using UDP encapsulation, the sender and receiver sides do not need to establish a session prior to data transmission. The main difference is that instead of calling the sendto() API call, we call the recvfrom() API call.

7.5 Source code Implementation – Adoption of CPU state at destination host

Once the data has been successfully received at the destination host, the "ping adapt" command is executed to replace the current CPU register values with the new one of the source. Two main actions need to be performed in order to adopt and replace the existing values of the registers. First, we need to check and confirm that the required level of resources are available. The same number of the migrated vCPUs from the source host need to be available at the destination host. However, we don't need to create a Guest instance on the destination in order to have the appropriate number of resources available prior to CPU migration. During the build process of Xvisor we pre-define the number of the available vCPUs that we want to assign on a Guest instance. The sample of code in Figure 7.26 does that by taking as input the ID numbers of the migrated vCPUs.

```
vcpu1 = vmm_manager_vcpu(id1);
     if (!vcpu1) {
      vmm_cprintf(cdev, "Failed to find vcpu\n");
      return VMM_EFAIL;
     }
vcpu2 = vmm_manager_vcpu(id2);
     if (!vcpu2) {
      vmm_cprintf(cdev, "Failed to find vcpu\n");
      return VMM_EFAIL;
     }
```

Figure 7-27 Validation of the passing vCPUs IDs

Once we confirm they are available and we have reserved the right level of resources, we can initiate the adoption process whereby the received data is loaded into the appropriate registers at the destination. The core functionality on the adoption process is based on the use of an array that works as a temporary storage for the values of the incoming registers, this is done in order to avoid any loss of data. Then we load the values from the array to the CPU registers at the destination, this is essentially a common load and store operation as supported by the ARM instruction set architecture. The following sample of code in Figure 7.27 performs this task.

```
/*Store the values of register inside the array*/
 data_buffer[0] = regs1->sp;
 data_buffer[1] = regs1->lr;
 data_buffer[2] = regs1->pc;
 data_buffer[3] = (regs1->pstate & 0xffffffff);

 regs2->sp = data_buffer[0];
 regs2->lr = data_buffer[1];
 regs2->pc = data_buffer[2];
 regs2->pstate = data_buffer[3];
```

Figure 7-28 Temporary storage for the received registers

So, as we explained in the previous section, in order to retrieve and store the value of a register into a temporary data structure like an array we make use of the following general type of code:

X [ ] = pointer -> register;

where X is the name of a data structure, pointer is a defined kind of pointer that points to an allocated memory address and register is the name of the register that we want to capture. In this case we the reverse process in order to load or restore a value to a register, again using a temporary data structure we perform the following type of code:

Pointer -> register = X [ ];

Figure 7.28 below shows the source code implementation for the adoption for the migrated CPU state. The “cmd ping adapt” function performs the required validation checks while appendix A-7 describes the nested function "netstack adapt registers" which executes the actual replacement of the received registers with those at the destination.

```
static int cmd_ping_adapt(struct vmm_chardev *cdev,
                   int argc, char **argv)
{
      int id1, id2;
      struct vmm_vcpu *vcpu1;
      struct vmm_vcpu *vcpu2;
      if((argc < 1) || (argc > 3)) {
            cmd_ping_usage(cdev);
            return VMM_EFAIL;
      }
      id1 = atoi(argv[0]);
      id2 = atoi(argv[1]);

      vcpu1 = vmm_manager_vcpu(id1);
      if (!vcpu1) {
            vmm_cprintf(cdev, "Failed to find vcpu\n");
            return VMM_EFAIL;
      }
```

```
        vcpu2 = vmm_manager_vcpu(id2);
        if (!vcpu2) {
                vmm_cprintf(cdev, "Failed to find vcpu\n");
                return VMM_EFAIL;
        }

        netstack_adapt_registers(arm_regs(vcpu1), arm_regs(vcpu2));
        return VMM_OK;
}
```

Figure 7-29 Source code for the execution of the adaption process

The sample of the code above was tested at the source system using some dummy data passing on from one vCPU to another. As we mentioned in Chapter 6, by default two vCPUs are assigned to a Guest machine. Using as source values some dummy data of the first vCPU, we stores these values into a temporary array. Then, we loaded those values from the array into the second available vCPU. Through that process we confirm and test that the source values of the vCPU1 replaced the initial values of the vCPU2 on the same host machine.

However, there are concerns about the success of the process on separate hosts. Ideally, the reception and adoption actions of the registers on the destination host should be merged into a single action in order to maintain system consistency and efficiency of the process. However, due to limitations that we mentioned in Chapter 6 as well as earlier in this chapter, we could not perform and test that. A suggested configuration step that is noted as future work.

7.6 Summary

In this chapter we introduced our novelty and enhanced functionality for the performance of a CPU live migration process available to ARM based Bare metal instances finding application as part of the Xvisor hypervisor architecture. We presented the configuration and code changes that need to be done prior to compilation of Xvisor in order to include our novel implementation for the performance of CPU state migration and then we explained our integrated menu that lists the commands for the transmission, reception, and adoption of the migrated registers. Following on in Chapter 8, we perform a demonstration of the CPU live migration as this takes place between two ARM based RPI hosts while we evaluate the time it takes to complete a CPU live migration in relation to the number of migrated registers.

8 Practical Experiment & Evaluation

This chapter demonstrates the execution of our CPU live migration process as explained and implemented in the previous chapter. This practical experiment covers the installation and configuration of the Xvisor hypervisor followed by the execution of our novel contribution of a CPU live migration process. Furthermore, we evaluate the impact that the number of migrated register groups has on the overall migration performance and completion time of the process.

8.1 Setting up the Lab Environment

Working on an isolated, internal network with no external connections or communications, we are going to perform a CPU live migration that emulates an operational scenario. The lab environment consists of two Raspberry PI (RPIs) generation three model B boards, as illustrated in the following Figure 8.1, therefore, our lab environment is considered secure enough from network vulnerabilities or threats that can interfere, putting in danger the integrity of the migrated data.

Source
Destination

Figure 8-1 Lab topology

A high-level overview of the characteristics and system configuration of each RPI board is given by Table 4, while a brief description of the steps and milestones that are covered in this experiment is given by the diagram in Figure 8.2.

Board	Raspberry Pi 3 – Model B (source)	Raspberry Pi 3 – Model B (destination)
ARM Processor	ARMv8	ARMv8
Kernel	Xvisor	Raspbian OS
TCP/IP Protocol	UDP	UDP
IP Address	192.168.0.10	192.168.0.11
Port	6002	6002

Table 4 Lab configuration overview

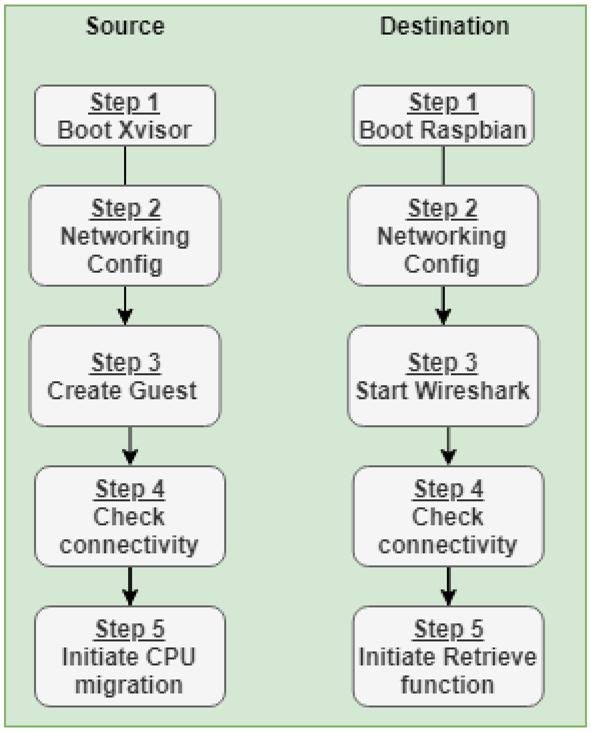


Figure 8-2 Experiment Milestones

As Figure 8.2 shows, starting at the source host, we install and boot the Xvisor hypervisor. Once it is booted, we make the necessary networking configuration changes and adjustments, so it can be part of the same local network as the destination host. Our lab environment is a private LAN topology, consisting of just the hosts. Therefore, it is considered secure enough with dedicated bandwidth to make use of UDP at the transport layer for the state migration process. Although TCP is more reliable through the error recovery process that it provides, by using UDP we achieve higher performance and less network latency since fewer bytes of overhead flow through the network. UDP also consumes less bandwidth and fewer processing cycles.

After that we create the Guest environment on the source. At the same time, at the destination host which is a common Raspbian OS, we make the same networking configuration and we start the local network packet monitoring tool called Wireshark on the Ethernet channel. We validate connectivity between the source and destination using the ping command. Once that is successful, we start the retrieve function on the destination host, setting it to listening mode for incoming packets while we initiate the CPU live migration process at the source. The initiation process as well as all the internal sub-processes was fully explained by our implementation in Chapter 7.

8.2 Configuration of the Source System

8.2.1 Step 1: Boot Xvisor on Source

Following the official documentation provided by the Xvisor community, in order to install and boot Xvisor the assistance of the U-boot bootloader is needed. Both the U-boot and Xvisor images are pre-installed in a formatted SD card, which is installed in the source RPI. All the necessary files are U-boot compatible. Through a serial connection to the source RPI using a software like Putty [115] and once the RPI is powered on, we need to interrupt the normal execution of the U-boot bootloader and provide the boot image of Xvisor. Once the U-boot> prompt comes up the boot files of Xvisor stored in SD card need to be loaded into the physical memory of the RPI board.

Those files are:

- the binaries of Xvisor compatible to U-boot (uvmm.bin),
- the device tree binaries that describe the hardware components of the RPI3 hardware (bcm.dtb) and
- the Guest Linux kernel that the Guest machine uses (udisk.img).

The following list of series of commands describes the loading phase of the required files as explained above. Additionally, this step can be automated using an "autoboot" configuration file that contains the following list and order of commands as part of the U-boot initial configuration.

1. We change partition in order to read from the SD card using the command: `mmc dev 0:0`

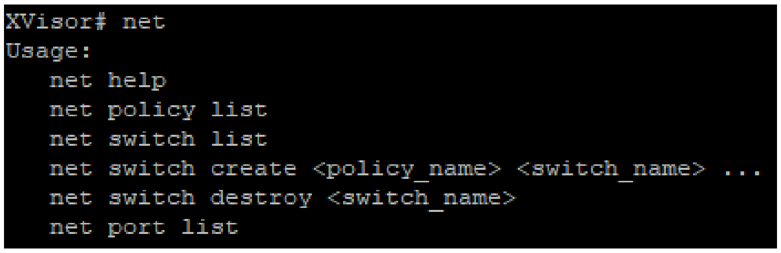
2. We extract and load the Xvisor binary to the physical memory using the command: `ext4load mmc 0:2 0x200000 uvmm.bin`
3. We extract and load the device tree binaries about the hardware SoC to the physical memory by executing the command: `ext4load mmc 0:2 0x800000 bcm2837-rpi-3-b.dtb`
4. We extract and load the Guest binaries to the physical memory using the command: `ext4load mmc 0:2 0x2000000 udisk.img`
5. Last, we boot Xvisor by executing: `bootm 0x200000 0x2000000 0x800000` which combines all the loaded files from the memory.

The “***bootm***” command instantly triggers and boots the Xvisor kernel. Once the Xvisor is loaded, we are prompted with the ***Xvisor#*** menu. We can confirm that it is fully functional by executing the “help” or “version” commands.

8.2.2 Step 2: Networking Configuration of Source system

Xvisor provides great flexibility by giving us the opportunity to adjust the configuration based on the needs and demands of the user. All the supported functionalities are controlled by a "configuration" file that gives us the ability to choose if we want to enable or disable each one of them, such as the network stack, which is an optional feature and disabled by default. During the migration process, we make use of the ported API functions that the LwIP suite offers. As we explained in the implementation chapter, through the source code we enable support for the Network stack as well as support for the USB and Ethernet controller.

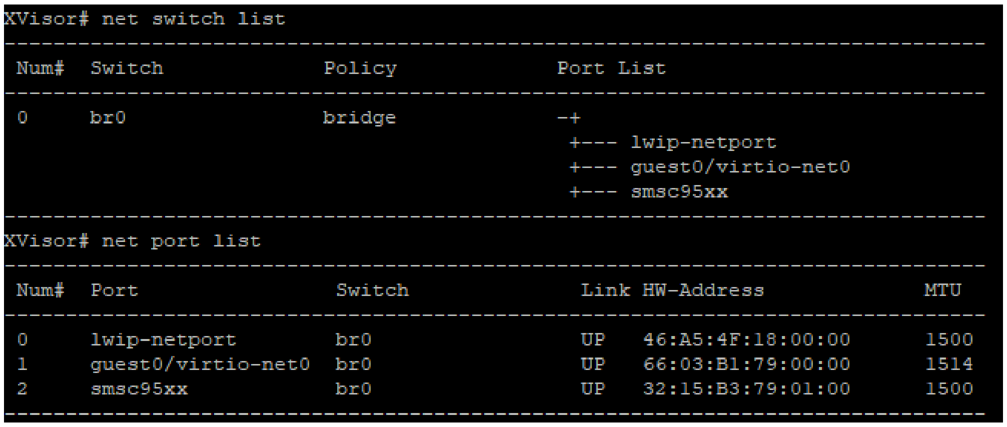
First, we confirm that the network TCP/IP stack is enabled and ready for use. Xvisor is equipped with a massive list of debugging and verification commands that allow us to identify and troubleshoot the currently available configuration and operation as well as monitor some system statistics and analytics data. As we noted earlier, by typing “help” it shows us the entire list of the available commands supported by Xvisor as is currently built. Figure 8.3 shows a list of the available subcommands as with the required parameters that a user needs to pass related to the networking configuration.



```
XVisor# net
Usage:
   net help
   net policy list
   net switch list
   net switch create <policy_name> <switch_name> ...
   net switch destroy <switch_name>
   net port list
```

Figure 8-3 Xvisor net debugging menu

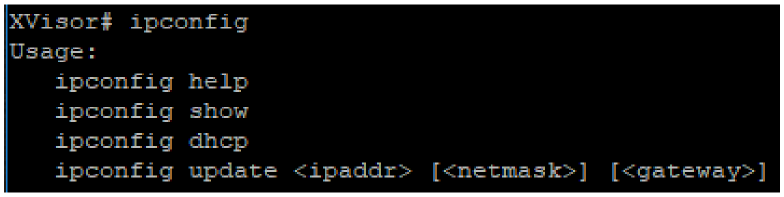
The use of the “net switch list” and “net port list” commands as illustrated in Figure 8.4, give us all the necessary information that we need to know about the configuration of the network virtualization. As we can observe, the system makes use of a bridge policy where the USB and Ethernet drivers of both host and Guest machines are connected and ported together. Another very useful feature is the “net port list” command where the MAC addresses of all the enabled network interfaces are listed. We keep a note of the MAC address of the “lwip-netport” where the network traffic generated by the Xvisor interfaces passes through as that information is necessary for the reconnaissance phase.



```
XVisor# net switch list
--------------------------------------------------------------------------------
 Num#  Switch             Policy             Port List
--------------------------------------------------------------------------------
 0     br0                bridge             -+
                                              +--- lwip-netport
                                              +--- guest0/virtio-net0
                                              +--- smsc95xx
--------------------------------------------------------------------------------
XVisor# net port list
--------------------------------------------------------------------------------
 Num#  Port                Switch              Link HW-Address           MTU
--------------------------------------------------------------------------------
 0     lwip-netport        br0                 UP   46:A5:4F:18:00:00    1500
 1     guest0/virtio-net0  br0                 UP   66:03:B1:79:00:00    1514
 2     smsc95xx            br0                 UP   32:15:B3:79:01:00    1500
--------------------------------------------------------------------------------
```

Figure 8-4 Xvisor networking information

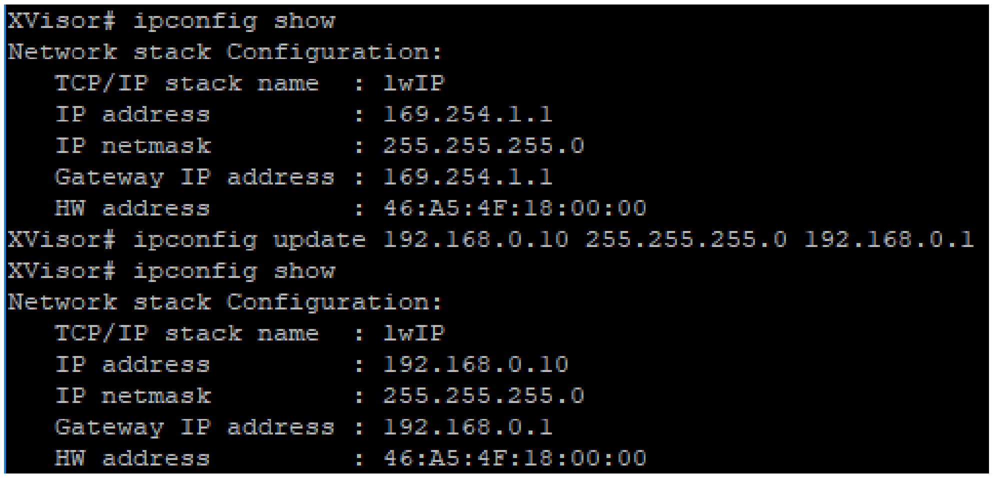
Starting with the configuration of the networking settings, both the source and destination systems are members of the same private network and need to communicate by making use of internal IP addresses. Since no DHCP server exists inside our private network, Xvisor automatically assigns a link local IPv4 address from the 169.254.0.0/24 range also known as Automatic Private Internet Protocol Addressing (APIPA). For our lab environment we chose to make use of the 192.168.0.0/24 IP addresses block. Otherwise, any private address reserved by the Internet Assigned Numbers Authority (IANA) is acceptable and valid.



```
XVisor# ipconfig
Usage:
   ipconfig help
   ipconfig show
   ipconfig dhcp
   ipconfig update <ipaddr> [<netmask>] [<gateway>]
```

Figure 8-5 Xvisor ipconfig debugging menu

Figure 8.5 above, lists all the available subcommands that the parent “ipconfig” menu provides related to the network configuration settings. At first, we execute the “ipconfig show” command in order to identify and later confirm the current IP address of the system. In order to configure the IP address of the Xvisor the execution of the “ipconfig update <ip_address> <netmask> <default gateway>” sub-command is applied with the desired values. Figure 8.6 shows the execution of those commands as well as the IP address before and after execution of the update command.



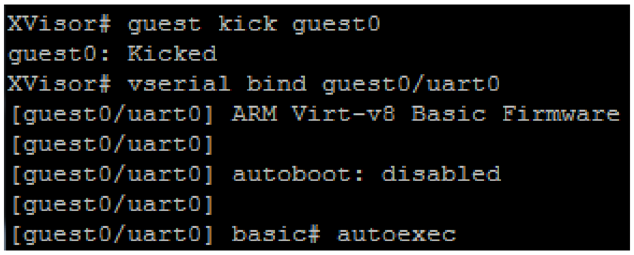
```
XVisor# ipconfig show
Network stack Configuration:
   TCP/IP stack name  : lwIP
   IP address         : 169.254.1.1
   IP netmask         : 255.255.255.0
   Gateway IP address : 169.254.1.1
   HW address         : 46:A5:4F:18:00:00
XVisor# ipconfig update 192.168.0.10 255.255.255.0 192.168.0.1
XVisor# ipconfig show
Network stack Configuration:
   TCP/IP stack name  : lwIP
   IP address         : 192.168.0.10
   IP netmask         : 255.255.255.0
   Gateway IP address : 192.168.0.1
   HW address         : 46:A5:4F:18:00:00
```

Figure 8-6 Xvisor IP address configuration

The final part of the configuration phase is to define the number of vCPUs that we want to allocate to the Guest machine. By default, at the boot process Xvisor assigns two vCPUs per Guest instance. If it is necessary or desired, we can add or remove an allocated vCPU to a Guest machine either during the build phase of the Xvisor source code, or during the normal operation. Furthermore, at this stage we must modify the source code of the migration process in order to adjust to the IP address of the destination system.

8.2.3 Step 3: Create and start the Guest environment

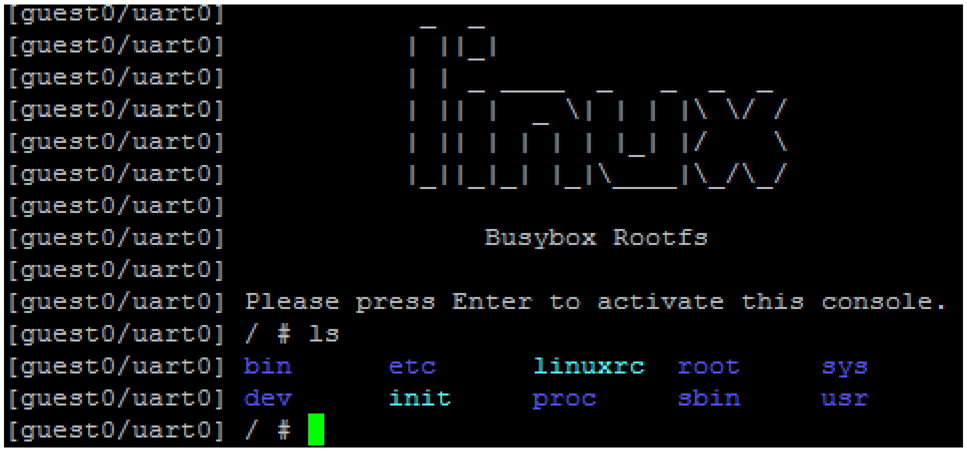
Once we have finished with the network configuration, we are ready to start the Guest machine on the source instance. Although during the boot process Xvisor creates a template of the Guest machine with the allocation of the number of vCPUs, as well as the Guest address space with all the memory regions, the Guest machine is not actually running yet.



```
XVisor# guest kick guest0
guest0: Kicked
XVisor# vserial bind guest0/uart0
[guest0/uart0] ARM Virt-v8 Basic Firmware
[guest0/uart0]
[guest0/uart0] autoboot: disabled
[guest0/uart0]
[guest0/uart0] basic# autoexec
```

Figure 8-7 Xvisor load Guest environment

We need to trigger (or kick as it is known in the Xvisor language) a Guest machine and then load the Linux kernel image from memory in order to start it running. As illustrated in Figure 8.7 above, the “Guest kick Guest0” command starts execution of the Guest environment. Interaction with the Guest environment can then happen by binding the serial interface to the Guest machine. The command “vserial bind Guest0/uart0” does that for us as is presented in Figure 8.8. This allows us to communicate and send characters to the Guest machine via a virtual serial interface which Xvisor internally creates for us. The “autoexec” command starts loading the Guest Linux kernel image into the Guest0 machine. As we can see in Figure 8.8, once the loading process of the Linux kernel has completed, the Linux logo appears in line with the traditional bash environment.

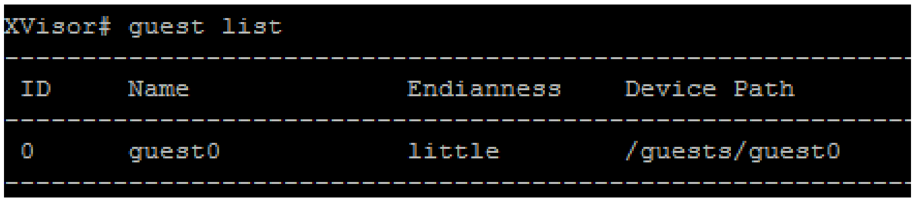


```
[guest0/uart0]          _  _
[guest0/uart0]         | ||_|
[guest0/uart0]         | |_ _ ____  _   _  _  _
[guest0/uart0]         | || | |  _ \| | | |\ \/ /
[guest0/uart0]         | || | | | | | |_| |/    \
[guest0/uart0]         |_||_|_| |_|\_____|\_/\_/
[guest0/uart0]
[guest0/uart0]               Busybox Rootfs
[guest0/uart0]
[guest0/uart0] Please press Enter to activate this console.
[guest0/uart0] / # ls
[guest0/uart0] bin       etc       linuxrc   root      sys
[guest0/uart0] dev       init      proc      sbin      usr
[guest0/uart0] / # 
```

Figure 8-8 Xvisor Guest Linux environment

Xvisor offers a “magic” combination of “***Esc + x + q***” characters that interrupts the serial connection from the Guest machine and returns it back to the Xvisor prompt. During that action, operation of the Guest machine does not stop, neither do the actions to save /restore Guest VM state as happens on other hypervisor environments or on x86 architecture systems. The only thing that the magic key combination does is to unbind the serial interface back to Xvisor. The Guest machine is still running as a process in the background.

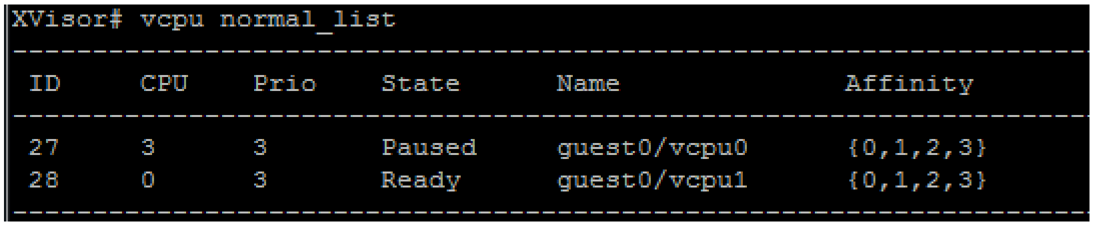
We can confirm the creation of the Guest machine by calling the “Guest list” command as shown in figure 8.9.



```
XVisor# guest list
--------------------------------------------------------------
 ID     Name                 Endianness    Device Path
--------------------------------------------------------------
 0      guest0               little        /guests/guest0
--------------------------------------------------------------
```

Figure 8-9 Xvisor list of the available Guest machines

As discussed in previous chapters, part of the reconnaissance phase is to identify the ID numbers of the allocated vCPUs in the Guest machine. During the execution of the CPU live migration process we pass the ID identification numbers of the assigned vCPUs in the Guest machine as arguments in order to identify which group of registers of which CPUs need to migrate to the destination. We can easily find the ID numbers of the allocated vCPUs in a Guest machine by calling the “vcpu normal list”.



```
XVisor# vcpu normal_list
------------------------------------------------------------------------
 ID     CPU    Prio    State      Name                 Affinity
------------------------------------------------------------------------
 27     3      3       Paused     guest0/vcpu0         {0,1,2,3}
 28     0      3       Ready      guest0/vcpu1         {0,1,2,3}
------------------------------------------------------------------------
```

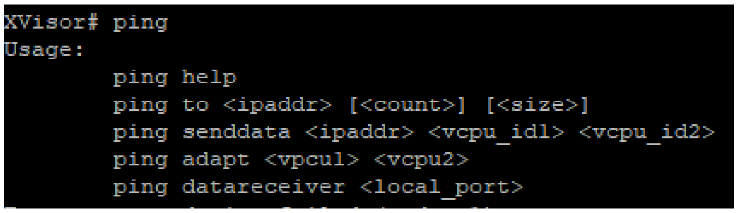
Figure 8-10 Xvisor list of normal vCPUs

As we can see from the output of the command above in Figure 8.10, the vCPUs with ID numbers 27 and 28 are both allocated to machine Guest0. Keeping note of those, we have collected all the necessary information that we need in order to execute and start the migration phase.

8.2.4 Step 4: Check connectivity with Destination system

Before we start a CPU live migration process, we first need to confirm and verify connectivity among the source and destination systems by using the PING command, giving the IP address of the destination host. Figure 8.11 shows, the list of the available subcommands related with ping. The “ping to <ip_address>

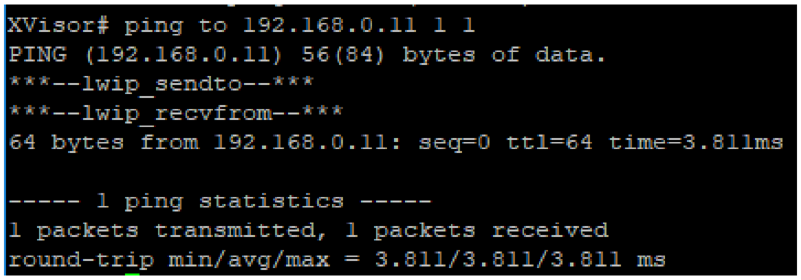
<count> <size>" command works like the traditional PING command, taking as arguments the IP address of the destination and the values of the size and count of packets.



```
XVisor# ping
Usage:
        ping help
        ping to <ipaddr> [<count>] [<size>]
        ping senddata <ipaddr> <vcpu_id1> <vcpu_id2>
        ping adapt <vpcu1> <vcpu2>
        ping datareceiver <local_port>
```

Figure 8-11 Xvisor ping debugging menu

Figure 8.12 proves that the ping command successfully reaches the destination machine from the Xvisor prompt. Furthermore, for troubleshooting purposes, we have enabled some debugging outputs that confirm the call and execution of the relative LwIP API functions to send and receive the related ICMP packets.



```
XVisor# ping to 192.168.0.11 1 1
PING (192.168.0.11) 56(84) bytes of data.
***--lwip_sendto--***
***--lwip_recvfrom--***
64 bytes from 192.168.0.11: seq=0 ttl=64 time=3.811ms

----- 1 ping statistics -----
1 packets transmitted, 1 packets received
round-trip min/avg/max = 3.811/3.811/3.811 ms
```

Figure 8-12 Execution of Ping command, testing connectivity

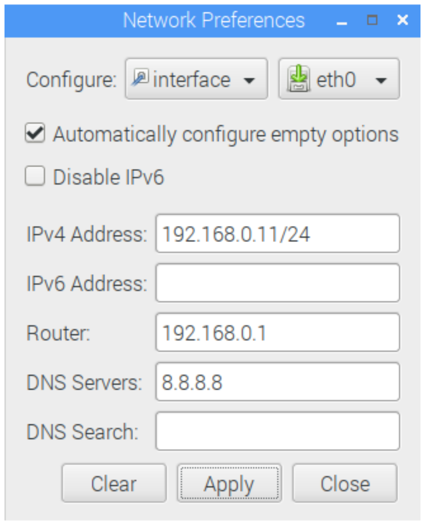
8.3 Configuration of the Destination system

8.3.1 Step 1: Boot Raspbian

The Raspbian OS is a very popular operating system based on the Debian Linux distribution specifically intended for Raspberry Pi hardware. Like many other Linux distributions, the Raspbian OS provides a user-friendly graphical interface. Using an SD card, we boot Raspbian OS on the destination RPI.

8.3.2 Step 2: Networking configuration of the destination

AS we did on the source host, we first need to configure the Ethernet network interface of the destination system to be part of the same local network, giving it an IP address from the 192.168.0.0/24 range. Figure 8.13 shows the network preferences panel of Raspbian OS where we apply the networking interface configuration settings.



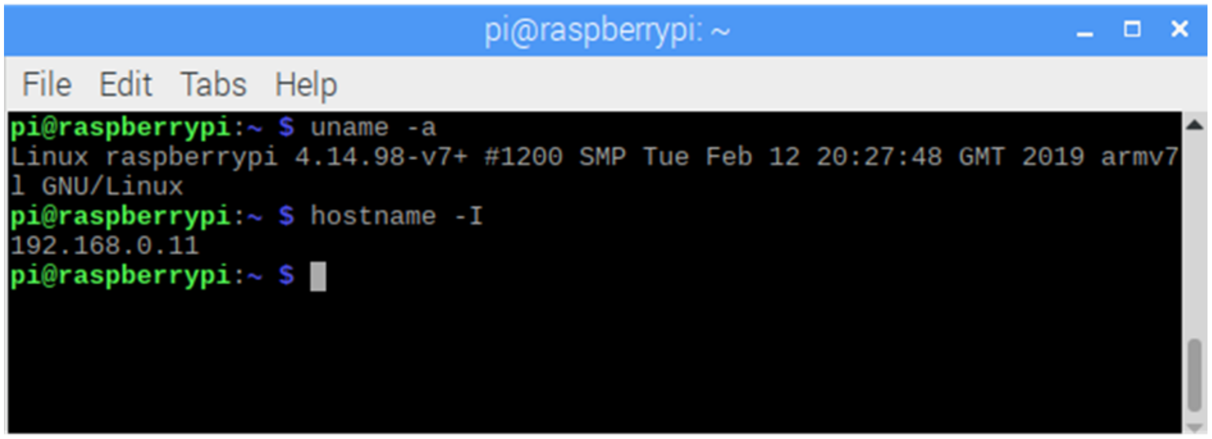
☑ Automatically configure empty options

☐ Disable IPv6

IPv4 Address:	192.168.0.11/24
IPv6 Address:	
Router:	192.168.0.1
DNS Servers:	8.8.8.8
DNS Search:	

Figure 8-13 Raspbian Network interface configuration

Figure 8.14 below, shows system details about the installed version of the Linux kernel while we can confirm the IP address that we gave the destination system.



```
pi@raspberrypi:~ $ uname -a
Linux raspberrypi 4.14.98-v7+ #1200 SMP Tue Feb 12 20:27:48 GMT 2019 armv7
l GNU/Linux
pi@raspberrypi:~ $ hostname -I
192.168.0.11
pi@raspberrypi:~ $ 
```

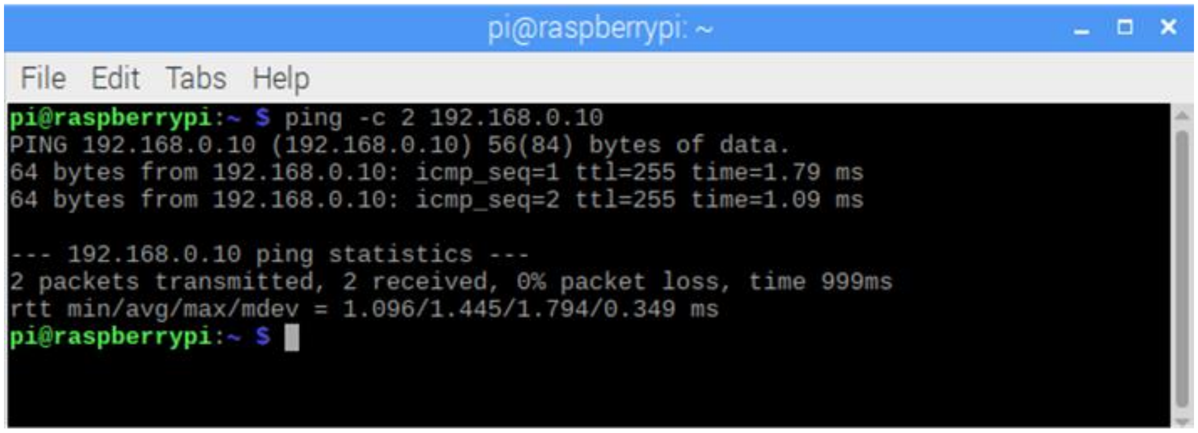
Figure 8-14 RPI system information

8.3.3 Step 3: Start Wireshark

Wireshark is an, open-source network traffic analyser and monitoring tool that is widely used in many enterprise and research projects. Wireshark allows us to capture the traffic sent between the source and destination. Through that, we can verify and inspect packet information as well as the migrated data. We start Wireshark by listening for any incoming traffic on the Ethernet interface.

8.3.4 Step 4: Verify Connectivity with Source System

Before we start the CPU live migration process, we again need to verify the connectivity between source and destination using the PING command. Even if we executed the PING command at the source getting a reply from destination it is still important to perform a two-way verification in order to update the local ARP table with the pair of MAC and IP addresses for each host. Figure 8.15 shows the successful execution of the PING command from the destination to the source, verifying the connectivity between them.

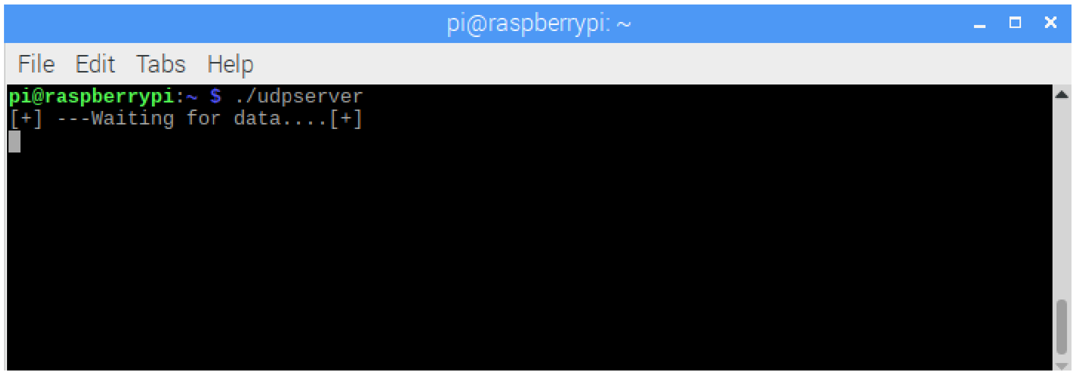


```
pi@raspberrypi:~ $ ping -c 2 192.168.0.10
PING 192.168.0.10 (192.168.0.10) 56(84) bytes of data.
64 bytes from 192.168.0.10: icmp_seq=1 ttl=255 time=1.79 ms
64 bytes from 192.168.0.10: icmp_seq=2 ttl=255 time=1.09 ms

--- 192.168.0.10 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 1.096/1.445/1.794/0.349 ms
pi@raspberrypi:~ $
```

Figure 8-15 Checking network connectivity between source and destination

8.3.5 Step 5: Initiation of the retrieve function at the destination system

At this stage, both source and destination systems have completed the initial configuration and reconnaissance phase and we are ready to move forward to our migration phase by initiating the migration process. So, we set the destination system in “listening” mode, ready to receive and accept the migrated CPU state. As we can observe in Figure 8.16, we are calling the ***udpserver*** bash script as explained in Chapter 7, which creates the UDP tunnel and puts the system in the listening mode, waiting for external connections on the pre-configured port.



```
pi@raspberrypi:~ $ ./udpserver
[+] ---Waiting for data....[+]
```

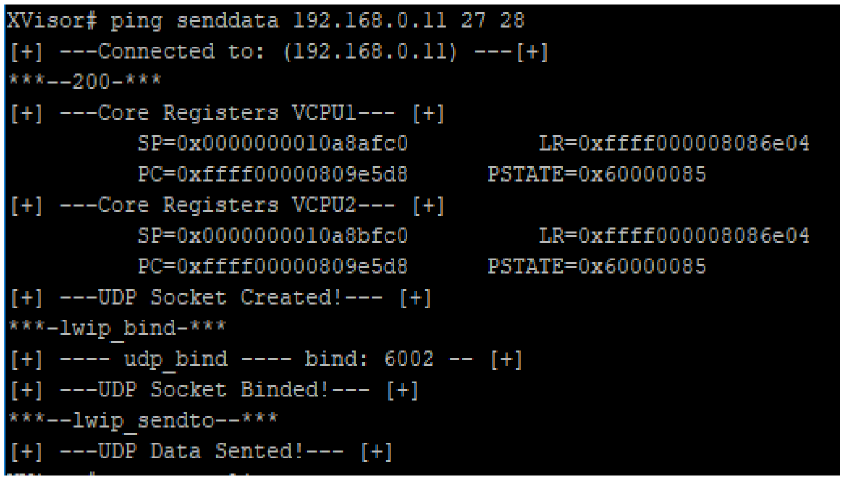
Figure 8-16 Start receiver function on the destination

8.4 Execution of CPU live migration process

In order to observe and monitor the CPU live migration process from source to the destination, we make use of Wireshark as well as the output of the debugging messages that we have code-embedded and enabled in the receiver functions. That will help us to confirm and more easily identify the migration process as triggered by the source system.

We aim to analyse the execution and the output of the migration function as is it called and executed through the Xvisor prompt interface on the source system. In Figure 8.17, the call of the "ping senddata <ip_address> <vcpu1> <vcpu2>" command takes as arguments the destination IP address and the ID numbers of the vCPUs that belong to the Guest machine. Once the command is executed the following sequence of actions occur:

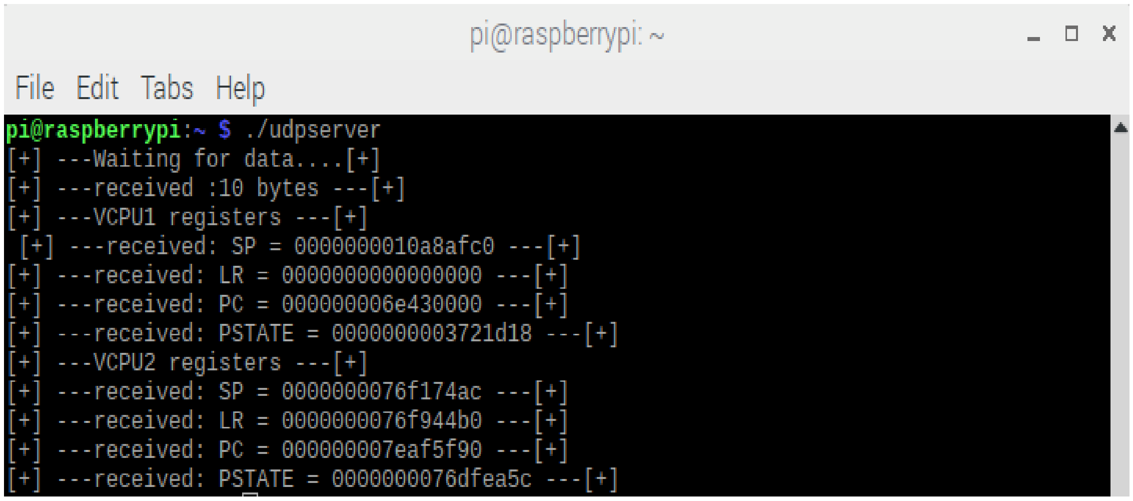
- A packet of 200 bytes is constructed, containing all the information and required headers.
- In parallel, and mostly for debugging reasons it prints the selected values of the registers from both vCPUs that will be migrated to the destination system while in the background, the migrated group of registers is temporarily stored into an array that later we pass as data in the UDP packet.
- As Figure 10.20 shows, a UDP socket is created and bound on port 6002 and the IP address that we gave earlier on the source system.
- The successful creation and bind process as further confirmation of the sent data from the source to the destination are confirmed through the debugging messages that are printed on the terminal through the migration process.



```
XVisor# ping senddata 192.168.0.11 27 28
[+] ---Connected to: (192.168.0.11) ---[+]
***--200-***
[+] ---Core Registers VCPU1--- [+]
          SP=0x0000000010a8afc0          LR=0xffff000008086e04
          PC=0xffff00000809e5d8      PSTATE=0x60000085
[+] ---Core Registers VCPU2--- [+]
          SP=0x0000000010a8bfc0          LR=0xffff000008086e04
          PC=0xffff00000809e5d8      PSTATE=0x60000085
[+] ---UDP Socket Created!--- [+]
***-lwip_bind-***
[+] ---- udp_bind ---- bind: 6002 -- [+]
[+] ---UDP Socket Binded!--- [+]
***--lwip_sendto--***
[+] ---UDP Data Sented!--- [+]
```

Figure 8-17 Call of CPU live migration function

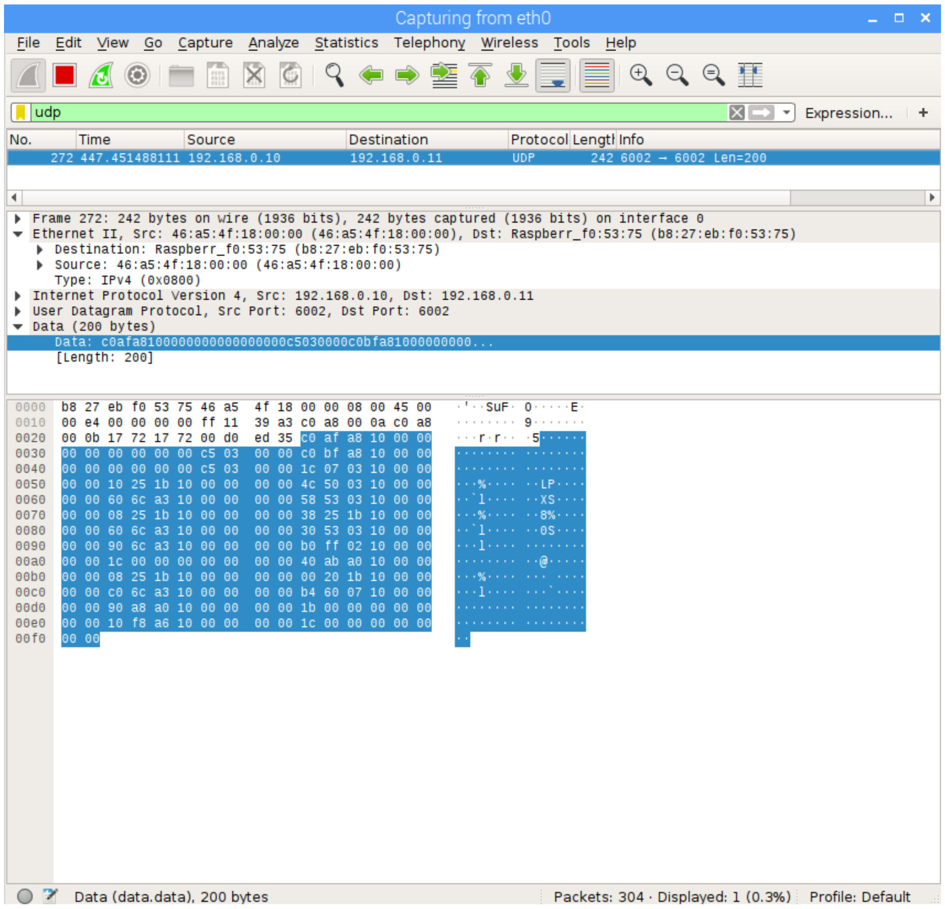
At the same time on the destination system, as Figure 8.18 indicates, the receiving script is initiated, listening, and waiting for incoming connections and data. As the data is received for verification purposes, similar to what we did on the transmission side, we print the received values of the registers. Doing a comparison of the received values with the ones to the sender, we can confirm that they match. In that way we can confirm the consistency and integrity of the migration process.



```
pi@raspberrypi:~ $ ./udpserver
[+] ---Waiting for data....[+]
[+] ---received :10 bytes ---[+]
[+] ---VCPU1 registers ---[+]
 [+] ---received: SP = 0000000010a8afc0 ---[+]
[+] ---received: LR = 0000000000000000 ---[+]
[+] ---received: PC = 000000006e430000 ---[+]
[+] ---received: PSTATE = 0000000003721d18 ---[+]
[+] ---VCPU2 registers ---[+]
[+] ---received: SP = 0000000076f174ac ---[+]
[+] ---received: LR = 0000000076f944b0 ---[+]
[+] ---received: PC = 000000007eaf5f90 ---[+]
[+] ---received: PSTATE = 0000000076dfea5c ---[+]
```

Figure 8-18 RPI - Receiving CPU register values

During the CPU live migration process, Wireshark is running in the background and captures all the networking packets. The following Figure 8.19 shows the received packets on the Ethernet interface of the destination system.

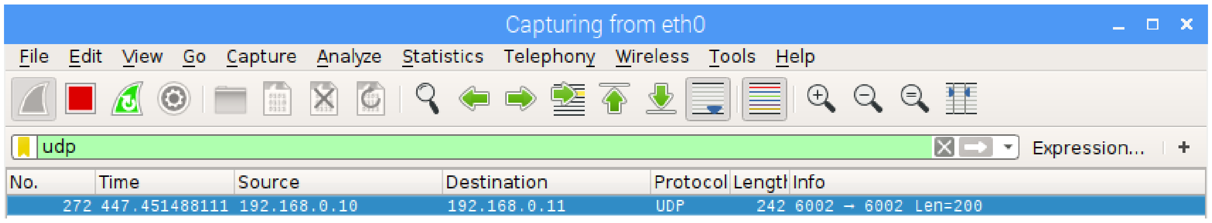

Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
udp
Expression…
+

```
No.   Time          Source        Destination   Protocol Length Info
  272 447.451488111 192.168.0.10  192.168.0.11  UDP      242 6002 → 6002 Len=200

▸ Frame 272: 242 bytes on wire (1936 bits), 242 bytes captured (1936 bits) on interface 0
▾ Ethernet II, Src: 46:a5:4f:18:00:00 (46:a5:4f:18:00:00), Dst: Raspberr_f0:53:75 (b8:27:eb:f0:53:75)
  ▸ Destination: Raspberr_f0:53:75 (b8:27:eb:f0:53:75)
  ▸ Source: 46:a5:4f:18:00:00 (46:a5:4f:18:00:00)
    Type: IPv4 (0x0800)
▸ Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.11
▸ User Datagram Protocol, Src Port: 6002, Dst Port: 6002
▾ Data (200 bytes)
    Data: c0afa8100000000000000000c5030000c0bfa81000000000...
    [Length: 200]

0000  b8 27 eb f0 53 75 46 a5  4f 18 00 00 08 00 45 00   ·'··SuF· O·····E·
0010  00 e4 00 00 00 00 ff 11  39 a3 c0 a8 00 0a c0 a8   ········ 9·······
0020  00 0b 17 72 17 72 00 d0  ed 35 c0 af a8 10 00 00   ···r·r·· ·5······
0030  00 00 00 00 00 00 c5 03  00 00 c0 bf a8 10 00 00   ········ ········
0040  00 00 00 00 00 00 c5 03  00 00 1c 07 03 10 00 00   ········ ········
0050  00 00 10 25 1b 10 00 00  00 00 4c 50 03 10 00 00   ···%···· ··LP····
0060  00 00 60 6c a3 10 00 00  00 00 58 53 03 10 00 00   ··`l···· ··XS····
0070  00 00 08 25 1b 10 00 00  00 00 38 25 1b 10 00 00   ···%···· ··8%····
0080  00 00 60 6c a3 10 00 00  00 00 30 53 03 10 00 00   ··`l···· ··0S····
0090  00 00 90 6c a3 10 00 00  00 00 b0 ff 02 10 00 00   ···l···· ········
00a0  00 00 1c 00 00 00 00 00  00 00 40 ab a0 10 00 00   ········ ··@·····
00b0  00 00 08 25 1b 10 00 00  00 00 00 20 1b 10 00 00   ···%···· ··· ····
00c0  00 00 c0 6c a3 10 00 00  00 00 b4 60 07 10 00 00   ···l···· ···`····
00d0  00 00 90 a8 a0 10 00 00  00 00 1b 00 00 00 00 00   ········ ········
00e0  00 00 10 f8 a6 10 00 00  00 00 1c 00 00 00 00 00   ········ ········
00f0  00 00                                              ··

Data (data.data), 200 bytes          Packets: 304 · Displayed: 1 (0.3%)    Profile: Default
```

Figure 8-19 Wireshark Main window - UDP packet

Taking a closer look, by analysing Wireshark's panels we can gather the following information related to the migration process of the CPU state:

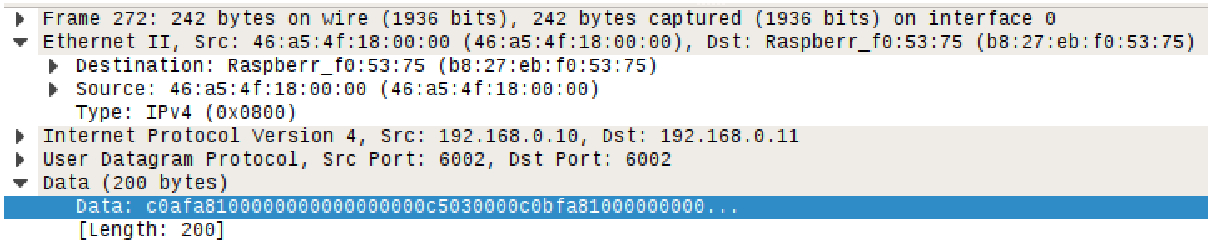
- As the top panel of Wireshark in Figure 8.20 indicates, we can clearly see a UDP packet having as a source the IP address that we have configured, 192.168.0.10 (source system) and port number 6002, received from the system with IP address 192.168.0.11 (destination system) listening on port 6002.


Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
udp
Expression…
+

```
No.   Time          Source        Destination   Protocol Length Info
  272 447.451488111 192.168.0.10  192.168.0.11  UDP      242 6002 → 6002 Len=200
```

Figure 8-20 Wireshark Packet List Pane

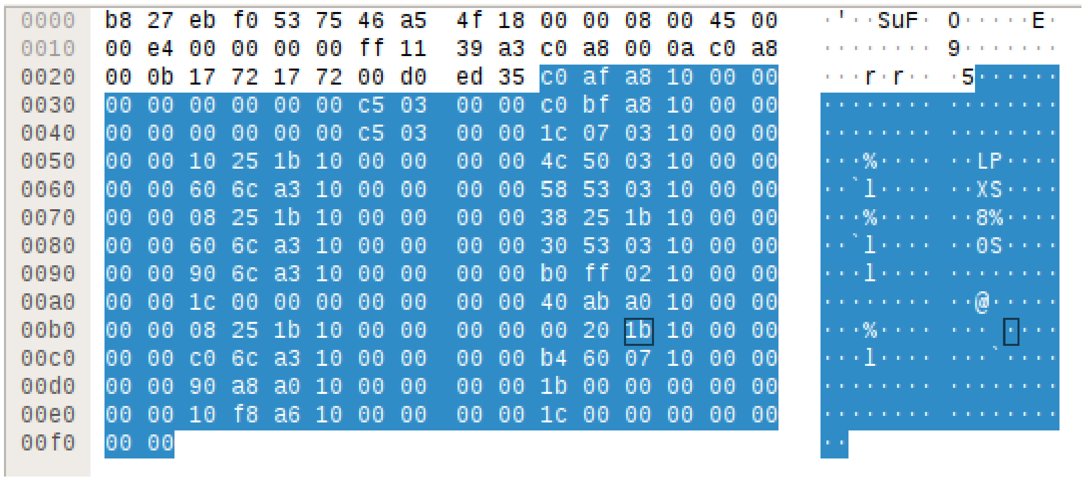
- Through a further inspection, in Figure 8.21, Wireshark gives us a full description of the content of the captured packet. We can see all the details that describe the frame section, the IP header and the UDP header while at the end resides the data portion of the packet.



```
▶ Frame 272: 242 bytes on wire (1936 bits), 242 bytes captured (1936 bits) on interface 0
▼ Ethernet II, Src: 46:a5:4f:18:00:00 (46:a5:4f:18:00:00), Dst: Raspberr_f0:53:75 (b8:27:eb:f0:53:75)
   ▶ Destination: Raspberr_f0:53:75 (b8:27:eb:f0:53:75)
   ▶ Source: 46:a5:4f:18:00:00 (46:a5:4f:18:00:00)
     Type: IPv4 (0x0800)
▶ Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.11
▶ User Datagram Protocol, Src Port: 6002, Dst Port: 6002
▼ Data (200 bytes)
     Data: c0afa8100000000000000000c5030000c0bfa81000000000...
     [Length: 200]
```

Figure 8-21 Wireshark Packet Details Pane

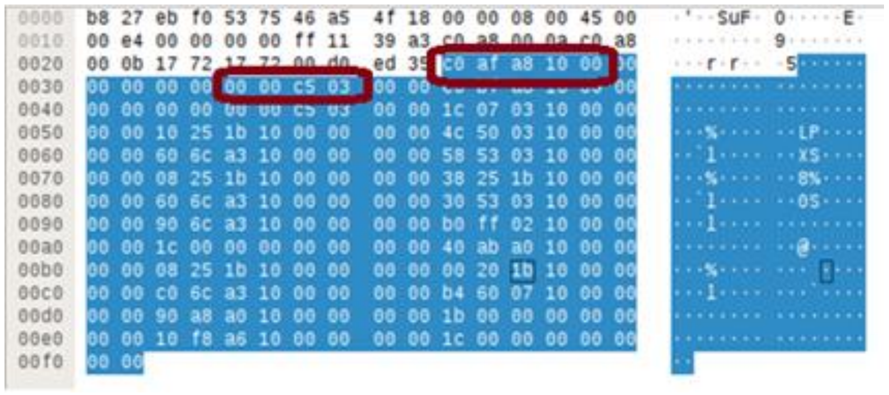
- Following, in the third panel of Wireshark's window as Figure 8.22 shows, a data portion of a packet is presented. In our case, the values of the CPU registers can be observed. The data of the presented packet are in hexadecimal representation. Taking a closer look, we can see the actual values of the registers as printed earlier.



```
0000  b8 27 eb f0 53 75 46 a5  4f 18 00 00 08 00 45 00   ·'··SuF· O·····E·
0010  00 e4 00 00 00 00 ff 11  39 a3 c0 a8 00 0a c0 a8   ········ 9·······
0020  00 0b 17 72 17 72 00 d0  ed 35 c0 af a8 10 00 00   ···r·r·· ·5······
0030  00 00 00 00 00 00 c5 03  00 00 c0 bf a8 10 00 00   ········ ········
0040  00 00 00 00 00 00 c5 03  00 00 1c 07 03 10 00 00   ········ ········
0050  00 00 10 25 1b 10 00 00  00 00 4c 50 03 10 00 00   ···%···· ··LP····
0060  00 00 60 6c a3 10 00 00  00 00 58 53 03 10 00 00   ··`l···· ··XS····
0070  00 00 08 25 1b 10 00 00  00 00 38 25 1b 10 00 00   ···%···· ··8%····
0080  00 00 60 6c a3 10 00 00  00 00 30 53 03 10 00 00   ··`l···· ··0S····
0090  00 00 90 6c a3 10 00 00  00 00 b0 ff 02 10 00 00   ···l···· ········
00a0  00 00 1c 00 00 00 00 00  00 00 40 ab a0 10 00 00   ········ ··@·····
00b0  00 00 08 25 1b 10 00 00  00 00 00 20 1b 10 00 00   ···%···· ··· ····
00c0  00 00 c0 6c a3 10 00 00  00 00 b4 60 07 10 00 00   ···l···· ···`····
00d0  00 00 90 a8 a0 10 00 00  00 00 1b 00 00 00 00 00   ········ ········
00e0  00 00 10 f8 a6 10 00 00  00 00 1c 00 00 00 00 00   ········ ········
00f0  00 00                                              ··
```

Figure 8-22 Wireshark Hexadecimal packet inspection panel

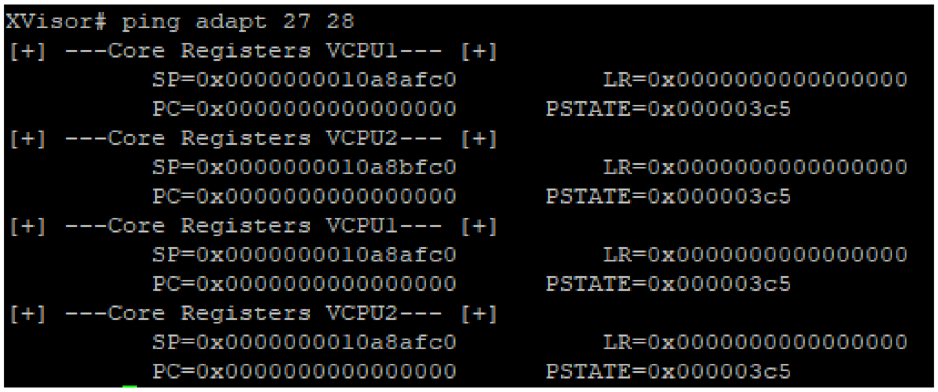
Figure 8.23 shows in red circles the actual values of the received SP register as well as the PSTATE register.



```
0000  b8 27 eb f0 53 75 46 a5  4f 18 00 00 08 00 45 00   ·'··SuF· O·····E·
0010  00 e4 00 00 00 00 ff 11  39 a3 c0 a8 00 0a c0 a8   ········ 9·······
0020  00 0b 17 72 17 72 00 d0  ed 35 c0 af a8 10 00 00   ···r·r·· ·5······
0030  00 00 00 00 00 00 c5 03  00 00 c0 af a8 10 00 00   ········ ········
0040  00 00 00 00 00 00 c5 03  00 00 1c 07 03 10 00 00   ········ ········
0050  00 00 10 25 1b 10 00 00  00 00 4c 50 03 10 00 00   ···%···· ··LP····
0060  00 00 60 6c a3 10 00 00  00 00 58 53 03 10 00 00   ··`l···· ··XS····
0070  00 00 08 25 1b 10 00 00  00 00 38 25 1b 10 00 00   ···%···· ··8%····
0080  00 00 60 6c a3 10 00 00  00 00 30 53 03 10 00 00   ··`l···· ··0S····
0090  00 00 90 6c a3 10 00 00  00 00 b0 ff 02 10 00 00   ···l···· ········
00a0  00 00 1c 00 00 00 00 00  00 00 40 ab a0 10 00 00   ········ ··@·····
00b0  00 00 08 25 1b 10 00 00  00 00 00 20 1b 10 00 00   ···%···· ········
00c0  00 00 c0 6c a3 10 00 00  00 00 b4 60 07 10 00 00   ···l···· ········
00d0  00 00 90 a8 a0 10 00 00  00 00 1b 00 00 00 00 00   ········ ········
00e0  00 00 10 f8 a6 10 00 00  00 00 1c 00 00 00 00 00   ········ ········
00f0  00 00                                              ··
```

Figure 8-23 PC and PSTATE registers values

Once we have received the CPU registers, the last step of our live migration procedure is the adoption of those values into the current values of the vCPUs at the destination. In order to achieve that, we call the "ping adapt <vcpu1> <vcpu2>" command. In Figure 8.24 we can observe the output of the adoption function as it takes place through the Xvisor command line interface. The first two outputs of the adoption functions, vCPU1 and vCPU2, describe the values of the registers at the destination they currently have before the replacement, while the two outputs at the bottom of the figure describe the values of the registers after the replacement process.



```
XVisor# ping adapt 27 28
[+] ---Core Registers VCPU1--- [+]
        SP=0x0000000010a8afc0        LR=0x0000000000000000
        PC=0x0000000000000000    PSTATE=0x000003c5
[+] ---Core Registers VCPU2--- [+]
        SP=0x0000000010a8bfc0        LR=0x0000000000000000
        PC=0x0000000000000000    PSTATE=0x000003c5
[+] ---Core Registers VCPU1--- [+]
        SP=0x0000000010a8afc0        LR=0x0000000000000000
        PC=0x0000000000000000    PSTATE=0x000003c5
[+] ---Core Registers VCPU2--- [+]
        SP=0x0000000010a8afc0        LR=0x0000000000000000
        PC=0x0000000000000000    PSTATE=0x000003c5
```

Figure 8-24 Xvisor - Call of CPU register adoption function

8.5 Evaluation of CPU migration based on the amount of migrated registers

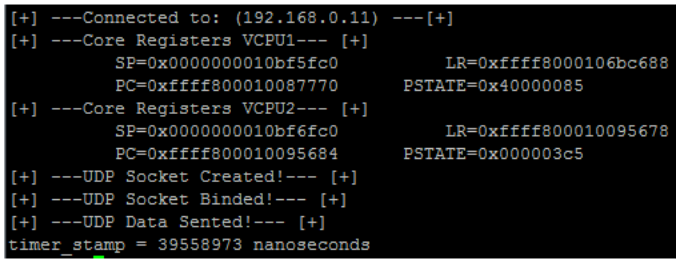
As we described in Chapter 7, most of the currently available ARM processors suitable for server and data centre infrastructures are members of the ARMv7 and ARMv8 versions. The number and size of the available registers differs from one version to another, as well as the CPU modes and exception layers as extensively analysed in Chapter 7. Based on those characteristics, an ARM state migration process differs from version to version, as the higher the number and size of the registers, the more time it takes to perform and complete a CPU state migration.

As we discussed earlier, the CPU state migration process consists of three main steps, capture, store and transfer the values of registers from source to destination host. Since on ARMv8 version there are 30 general purpose registers and 4 core registers of 64bit, while on ARMv7 version there are 12 general purpose registers and 2 core registers, less time should be required to perform such a process on ARMv7. Therefore, it’s important to identify the registers which are necessary to be captured and transfered in order to reassemble and resume a system on a migrated host.

The following experiment is a proof of the above theory. We performed a CPU state migration process several times, increasing each time the number of captured registers and measuring the time it takes to complete a live migration process to the destination host. Through this experiment we can observe how the number of CPU registers affects the overall CPU migration time. In the first instance, we performed a migration with just the 4 core registers per VCPU keeping records for the time it took to complete the task. The same process is repeated as shown in figures 8.25a to 8.25f- for the number of 8, 12, 17, 20 and 34 registers per VCPU.

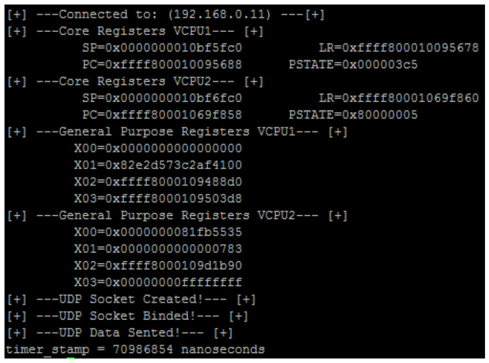
In order to evaluate the time, it takes to perform and complete a CPU live migration for a defined group of registers from source to the destination, we executed the CPU migration process several times. The following Figures from 8.25a to 8.25f shows the values of those registers as well as the time it took until we successfully migrated those registers to the destination and completed the CPU migration phase.

Starting with a group of four special registers as we can see in Figure 8.25a, increasing each time by adding some of the general-purpose registers which constitute the main state of a User mode. Figure 8.25b shows the migration performance with eight (8) registers per VCPU, following this is Figure 8.25c with twelve (12) registers, following with seventeen (17) in Figure 8.25d, while Figure 8.25e shows twenty (20) and fially Figure 8.25f shows the migration performance of all the general-purpose registers that are available to an ARMv8 processor.



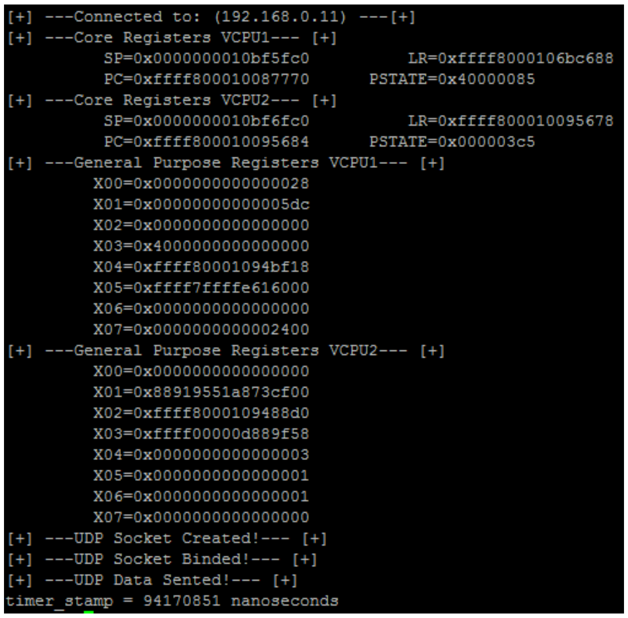
```
[+] ---Connected to: (192.168.0.11) ---[+]
[+] ---Core Registers VCPU1--- [+]
         SP=0x0000000010bf5fc0          LR=0xffff8000106bc688
         PC=0xffff800010087770      PSTATE=0x40000085
[+] ---Core Registers VCPU2--- [+]
         SP=0x0000000010bf6fc0          LR=0xffff800010095678
         PC=0xffff800010095684      PSTATE=0x000003c5
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 39558973 nanoseconds
```

a) Four registers per vCPU



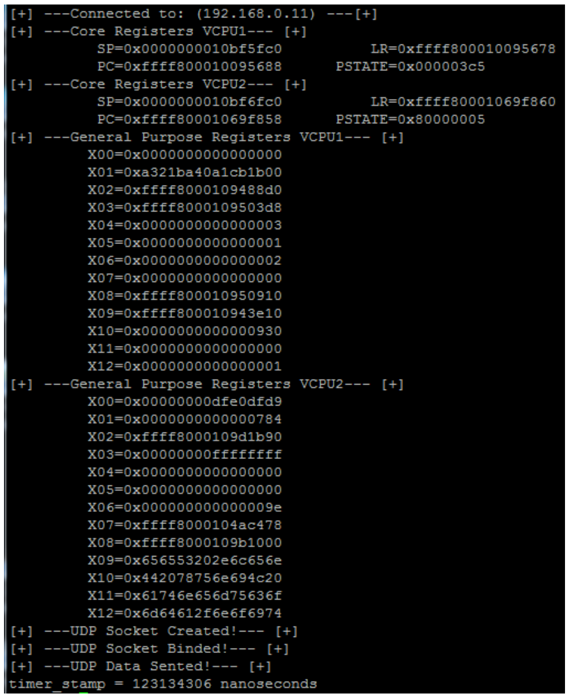
```
[+] ---Connected to: (192.168.0.11) ---[+]
[+] ---Core Registers VCPU1--- [+]
         SP=0x0000000010bf5fc0          LR=0xffff800010095678
         PC=0xffff800010095688      PSTATE=0x000003c5
[+] ---Core Registers VCPU2--- [+]
         SP=0x0000000010bf6fc0          LR=0xffff80001069f860
         PC=0xffff80001069f858      PSTATE=0x80000005
[+] ---General Purpose Registers VCPU1--- [+]
        X00=0x0000000000000000
        X01=0x82e2d573c2af4100
        X02=0xffff8000109488d0
        X03=0xffff8000109503d8
[+] ---General Purpose Registers VCPU2--- [+]
        X00=0x0000000081fb5535
        X01=0x0000000000000783
        X02=0xffff8000109d1b90
        X03=0x00000000ffffffff
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 70986854 nanoseconds
```

b) Eight registers per vCPU



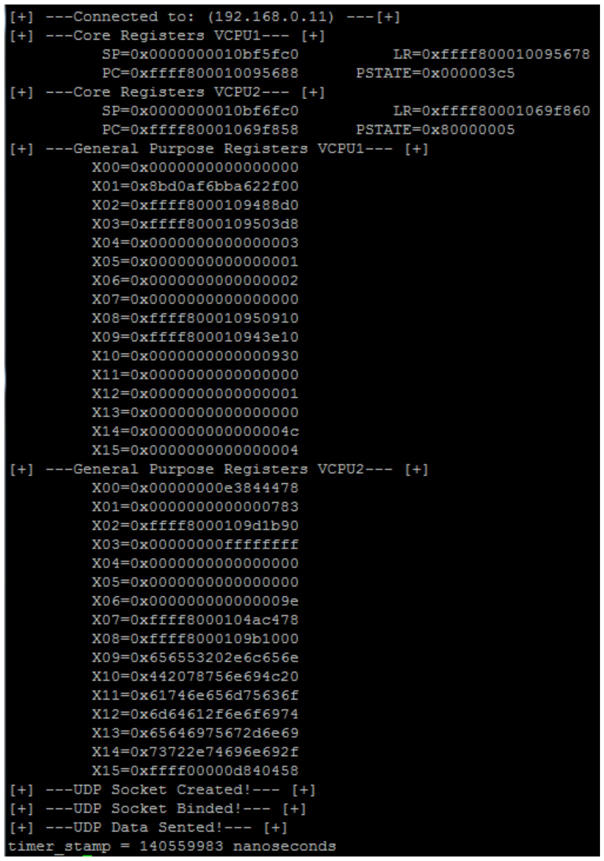
```
[+] ---Connected to: (192.168.0.11) ---[+]
[+] ---Core Registers VCPU1--- [+]
         SP=0x0000000010bf5fc0          LR=0xffff8000106bc688
         PC=0xffff800010087770      PSTATE=0x40000085
[+] ---Core Registers VCPU2--- [+]
         SP=0x0000000010bf6fc0          LR=0xffff800010095678
         PC=0xffff800010095684      PSTATE=0x000003c5
[+] ---General Purpose Registers VCPU1--- [+]
        X00=0x0000000000000028
        X01=0x00000000000005dc
        X02=0x0000000000000000
        X03=0x4000000000000000
        X04=0xffff80001094bf18
        X05=0xffff7ffffe616000
        X06=0x0000000000000000
        X07=0x0000000000002400
[+] ---General Purpose Registers VCPU2--- [+]
        X00=0x0000000000000000
        X01=0x88919551a873cf00
        X02=0xffff8000109488d0
        X03=0xffff00000d889f58
        X04=0x0000000000000003
        X05=0x0000000000000001
        X06=0x0000000000000001
        X07=0x0000000000000000
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 94170851 nanoseconds
```

c) Twelve registers per vCPU

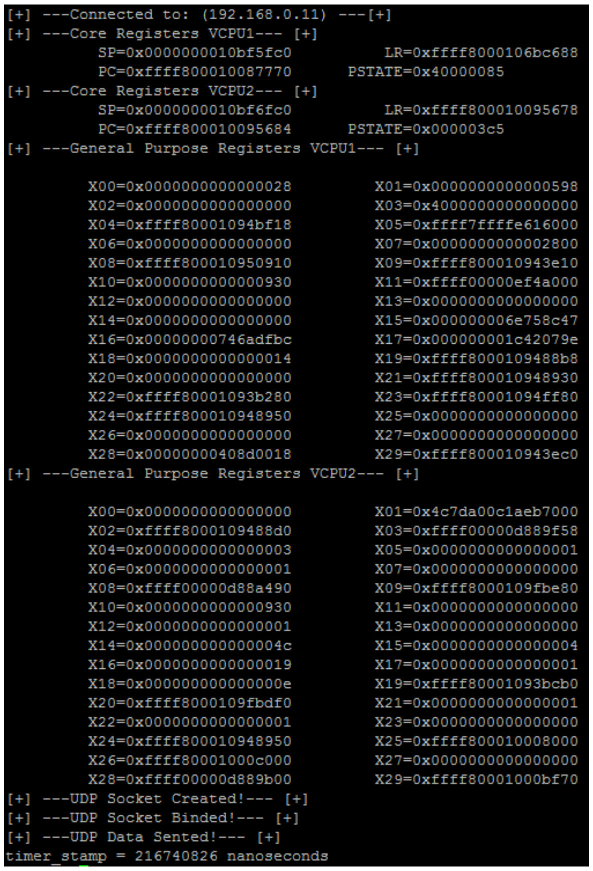


```
[+] ---Connected to: (192.168.0.11) ---[+]
[+] ---Core Registers VCPU1--- [+]
         SP=0x0000000010bf5fc0          LR=0xffff800010095678
         PC=0xffff800010095688      PSTATE=0x000003c5
[+] ---Core Registers VCPU2--- [+]
         SP=0x0000000010bf6fc0          LR=0xffff80001069f860
         PC=0xffff80001069f858      PSTATE=0x80000005
[+] ---General Purpose Registers VCPU1--- [+]
        X00=0x0000000000000000
        X01=0xa321ba40a1cb1b00
        X02=0xffff8000109488d0
        X03=0xffff8000109503d8
        X04=0x0000000000000003
        X05=0x0000000000000001
        X06=0x0000000000000002
        X07=0x0000000000000000
        X08=0xffff800010950910
        X09=0xffff800010943e10
        X10=0x0000000000000930
        X11=0x0000000000000000
        X12=0x0000000000000001
[+] ---General Purpose Registers VCPU2--- [+]
        X00=0x00000000dfe0dfd9
        X01=0x0000000000000784
        X02=0xffff8000109d1b90
        X03=0x00000000ffffffff
        X04=0x0000000000000000
        X05=0x0000000000000000
        X06=0x000000000000009e
        X07=0xffff8000104ac478
        X08=0xffff8000109b1000
        X09=0x656553202e6c656e
        X10=0x442078756e694c20
        X11=0x61746e656d75636f
        X12=0x6d64612f6e6f6974
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 123134306 nanoseconds
```

d) Seventeen registers per vCPU



```
[+] ---Connected to: (192.168.0.11) ---[+]
[+] ---Core Registers VCPU1--- [+]
        SP=0x0000000010bf5fc0          LR=0xffff800010095678
        PC=0xffff800010095688      PSTATE=0x000003c5
[+] ---Core Registers VCPU2--- [+]
        SP=0x0000000010bf6fc0          LR=0xffff80001069f860
        PC=0xffff80001069f858      PSTATE=0x80000005
[+] ---General Purpose Registers VCPU1--- [+]
       X00=0x0000000000000000
       X01=0x8bd0af6bba622f00
       X02=0xffff8000109488d0
       X03=0xffff8000109503d8
       X04=0x0000000000000003
       X05=0x0000000000000001
       X06=0x0000000000000002
       X07=0x0000000000000000
       X08=0xffff800010950910
       X09=0xffff800010943e10
       X10=0x0000000000000930
       X11=0x0000000000000000
       X12=0x0000000000000001
       X13=0x0000000000000000
       X14=0x000000000000004c
       X15=0x0000000000000004
[+] ---General Purpose Registers VCPU2--- [+]
       X00=0x00000000e3844478
       X01=0x0000000000000783
       X02=0xffff8000109d1b90
       X03=0x00000000ffffffff
       X04=0x0000000000000000
       X05=0x0000000000000000
       X06=0x000000000000009e
       X07=0xffff8000104ac478
       X08=0xffff8000109b1000
       X09=0x656553202e6c656e
       X10=0x442078756e694c20
       X11=0x61746e656d75636f
       X12=0x6d64612f6e6f6974
       X13=0x65646975672d6e69
       X14=0x73722e74696e692f
       X15=0xffff00000d840458
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 140559983 nanoseconds
```



```
[+] ---Connected to: (192.168.0.11) ---[+]
[+] ---Core Registers VCPU1--- [+]
        SP=0x0000000010bf5fc0          LR=0xffff8000106bc688
        PC=0xffff800010087770      PSTATE=0x40000085
[+] ---Core Registers VCPU2--- [+]
        SP=0x0000000010bf6fc0          LR=0xffff800010095678
        PC=0xffff800010095684      PSTATE=0x000003c5
[+] ---General Purpose Registers VCPU1--- [+]

       X00=0x0000000000000028        X01=0x0000000000000598
       X02=0x0000000000000000        X03=0x4000000000000000
       X04=0xffff80001094bf18        X05=0xffff7ffffe616000
       X06=0x0000000000000000        X07=0x0000000000002800
       X08=0xffff800010950910        X09=0xffff800010943e10
       X10=0x0000000000000930        X11=0xffff00000ef4a000
       X12=0x0000000000000000        X13=0x0000000000000000
       X14=0x0000000000000000        X15=0x000000006e758c47
       X16=0x00000000746adfbc        X17=0x000000001c42079e
       X18=0x0000000000000014        X19=0xffff8000109488b8
       X20=0x0000000000000000        X21=0xffff800010948930
       X22=0xffff80001093b280        X23=0xffff80001094ff80
       X24=0xffff800010948950        X25=0x0000000000000000
       X26=0x0000000000000000        X27=0x0000000000000000
       X28=0x00000000408d0018        X29=0xffff800010943ec0
[+] ---General Purpose Registers VCPU2--- [+]

       X00=0x0000000000000000        X01=0x4c7da00c1aeb7000
       X02=0xffff8000109488d0        X03=0xffff00000d889f58
       X04=0x0000000000000003        X05=0x0000000000000001
       X06=0x0000000000000001        X07=0x0000000000000000
       X08=0xffff00000d88a490        X09=0xffff8000109fbe80
       X10=0x0000000000000930        X11=0x0000000000000000
       X12=0x0000000000000001        X13=0x0000000000000000
       X14=0x000000000000004c        X15=0x0000000000000004
       X16=0x0000000000000019        X17=0x0000000000000001
       X18=0x000000000000000e        X19=0xffff80001093bcb0
       X20=0xffff8000109fbdf0        X21=0x0000000000000001
       X22=0x0000000000000001        X23=0x0000000000000000
       X24=0xffff800010948950        X25=0xffff800010008000
       X26=0xffff80001000c000        X27=0x0000000000000000
       X28=0xffff00000d889b00        X29=0xffff80001000bf70
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 216740826 nanoseconds
```

e) Twenty registers per vCPU

f) Thirty-four registers per vCPU

Figure 8-25 Number of migrated CPU registers a) 4, b) 8, c) 12, d) 17, e) 20, f) 34

Table 5 summarizes the results of the experiment. The first column shows the number of registers that we migrated each time while the second column shows the time in nanoseconds that it took to complete the migration. The last two columns for statistical reasons keep the subtraction of the time in nanoseconds among the migrations and the conversion of the time in seconds. Furthermore, the results and outcome of the experiment are presented in the graph below in Figure 8.26.

Registers per vCPU	**Nanoseconds (ns)**	**Subtraction**	**Seconds (s)**
4	39558973	-	0.039558973
8	70986854	-31427881	0.070986854
12	94170851	-23183997	0.094170851
17	123134306	-28963455	0.123134306

20	140559983	-17425677	0.140559983
34	216740826	-76180843	0.216740826

Table 5 Correlation of migrated registers with the time completion of CPU state migration

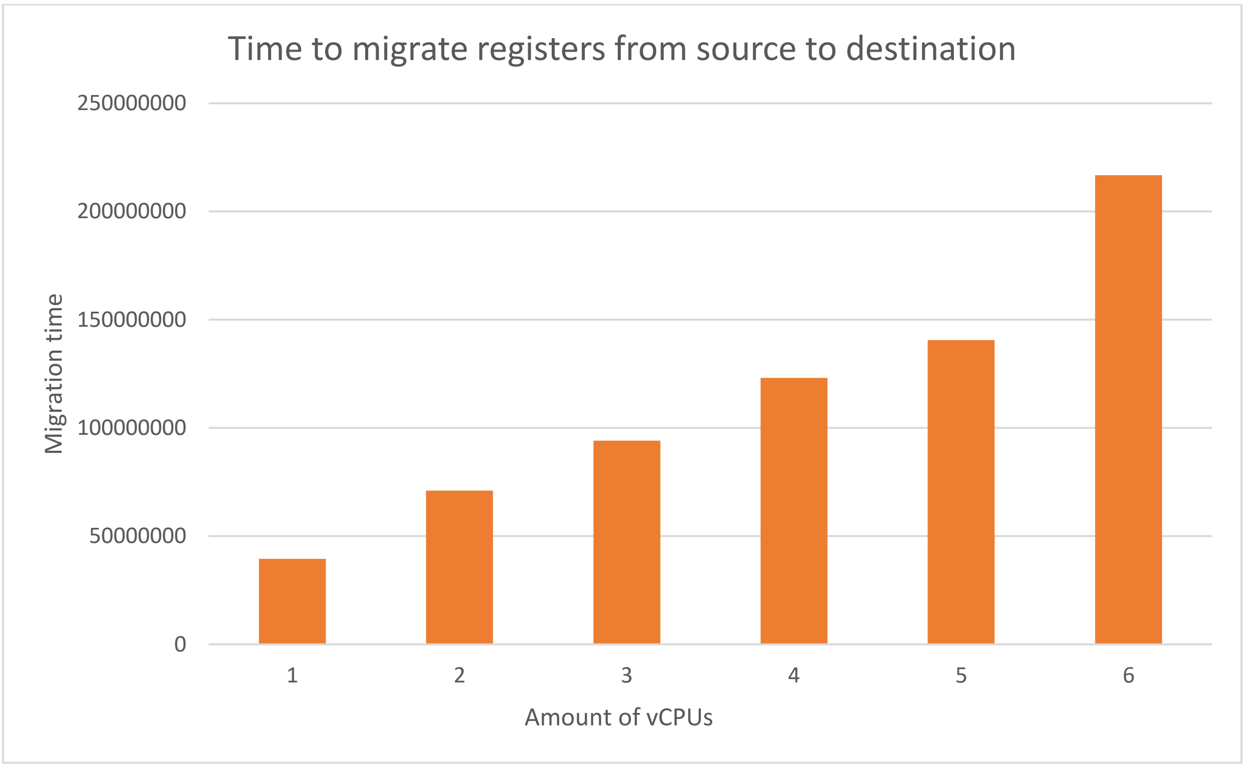


Figure 8-26 Graphical representations of CPU registers migration in relation to time

This experiment proves that by utilizing our novel migration scheme we can perform live migration of the CPU state on ARM architectures while we have the flexibility to choose which state we want to preserve and migrate from one system to another. However, during the performance of a live migration process consistency and stability of the process are two important factors. Therefore, in order to check and validate the consistency and stability of our scheme we repeated the same experiment ten consecutive times, migrating thirty-four registers in total. We selected the case of 34 registers consisting of the four main core registers and thirty general purpose registers since that group describes the state of the User CPU mode. Appendix B shows the results from the experiment that took place while Table 6 below lists the total migration time of each execution time. Additionally in Figure 8.27 we can observe that the total time of the CPU live migration is around the same levels which proves that there is consistency during the migration process and that our migration scheme is stable.

Total time of 34 CPU registers migration experiment results			
1	413941146	6	413935727
2	413941872	7	413940316
3	413936093	8	413940259
4	413937450	9	413942705
5	413941246	10	413942966

Table 6 Total time of 34 CPU registers migration experiment results

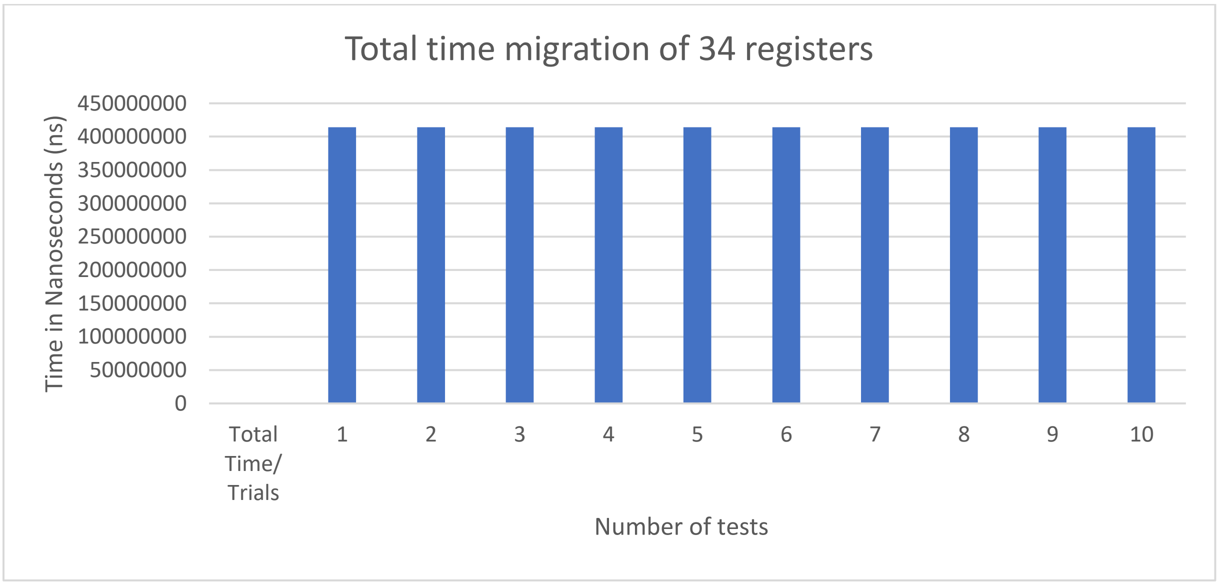


Figure 8-27 Graphical representation of the total time migration of 34 registers

As we can see, in the first round of the experiment (Table 5) the total migration time of 34 registers was 2ms less than the second round of the experiments (Table 6). This difference in the measurement of time is observed due to several factors such as the replacement and use of different type and length of physical networking ethernet cables, and due to the use of an alternative set of RPI hardware boards. Additionally, packet congestion during the normal operation of the switch affects and cause that minimal extra delay. However, that delay is considered negligible and inside normal operational levels.

8.6 Performance Analysis on CPU Host Load

Sections 8.4 and 8.5 shows that our novel scheme successfully achieve the CPU state migration on ARM based Bare metal instances. However, performance always is being an important factor in characterizing something efficient enough. Since our thesis and scheme focuses only on the CPU part, we also need to analyse and check the performance of the CPU. We mentioned in Section 6.3 that one of the main requirements is the utilization of a thin layer of a hypervisor that does not affect or degrade the performance of the system. A comparative analysis has been conducted by Xvisor developers between Xvisor with XEN and KVM as we mentioned in Chapter 6 which proves that Xvisor on ARM architecture occurs lower CPU overhead [269]. Backed up from this analysis, we also need to check that our novel scheme does not affect or comes in conflict with that. Therefore, we need to test and measure the CPU load of the host cores in the following four states:

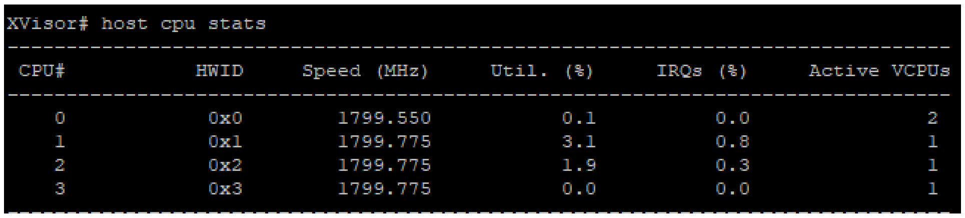
- once we boot the Xvisor hypervisor on RPI,
- once we boot the Guest instance,
- running a workload on the Guest instance
- when the live migration process takes place.

In that way we will identify and answer in the following questions:

- Is RPI a good choice for the edge?
- Is Xvisor hypervisor lightweight or there is CPU performance degradation?
- Does our novel scheme increase the CPU load during the live migration process?

Due to the existing limitations of Xvisor at the time that this experiment takes place, we cannot execute and perform a heavy workload benchmark application to test the CPU performance. Therefore, as alternative solution and in order to keep busy the processor we run multiple counters in the background on the Guest instance.

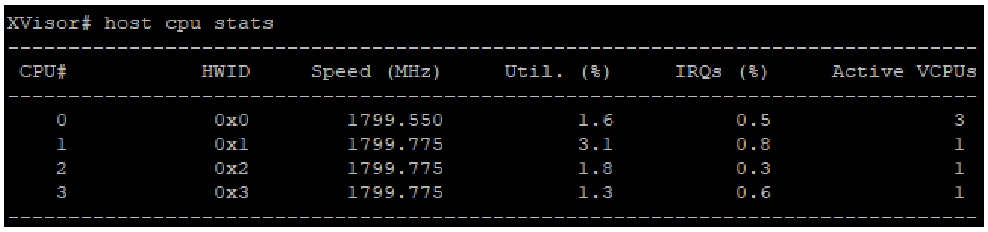
Figure 8.28 shows the load of the Host CPUs once the Xvisor boots in. From the output we can observe the current CPU load of each of the physical host CPUs as well as the number of the vCPUs are assigned on each physical CPU. The utilization is very low, which means that the processes running by Xvisor does not consume a significant amount of CPU at this stage.



```
XVisor# host cpu stats
--------------------------------------------------------------------------------
 CPU#           HWID     Speed (MHz)      Util. (%)      IRQs (%)      Active VCPUs
--------------------------------------------------------------------------------
   0             0x0       1799.550             0.1           0.0                 2
   1             0x1       1799.775             3.1           0.8                 1
   2             0x2       1799.775             1.9           0.3                 1
   3             0x3       1799.775             0.0           0.0                 1
--------------------------------------------------------------------------------
```

Figure 8-28 Host CPU load without a Guest instance

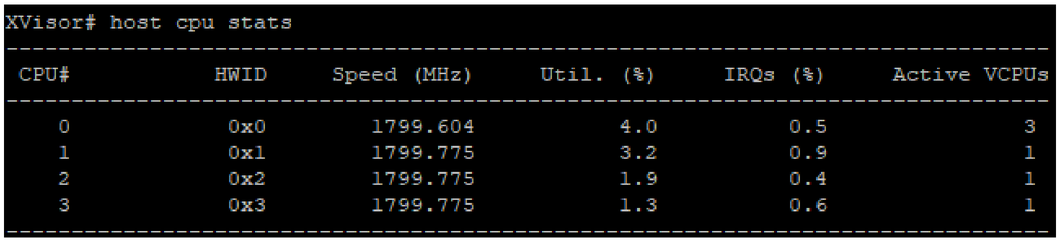
Figure 8.29 shows the load of the host CPUs after we have booted the Guest instance. As we can notice there is a small increase on all the host CPUs as well as an extra vCPU was assigned on the host physical CPU core with number 0. This is normal since several additional operations and processes run on the background related to both the Xvisor and the Guest instance. That extra vCPU is the one that was assigned on the Guest instance, while the second vCPU is not active yet. Once again, the CPU load is not too high.



```
XVisor# host cpu stats
--------------------------------------------------------------------------------
 CPU#           HWID     Speed (MHz)      Util. (%)      IRQs (%)      Active VCPUs
--------------------------------------------------------------------------------
   0             0x0       1799.550             1.6           0.5                 3
   1             0x1       1799.775             3.1           0.8                 1
   2             0x2       1799.775             1.8           0.3                 1
   3             0x3       1799.775             1.3           0.6                 1
--------------------------------------------------------------------------------
```

Figure 8-29 Host CPU load with a Guest instance

Then, we start the workload inside the Guest instance, and we measure the CPU load. In Figure 8.30 we can see that the CPU utilization of the first physical host CPU increases due to the processes running on the Guest instance. However, still the CPU utilization remains in normal levels.

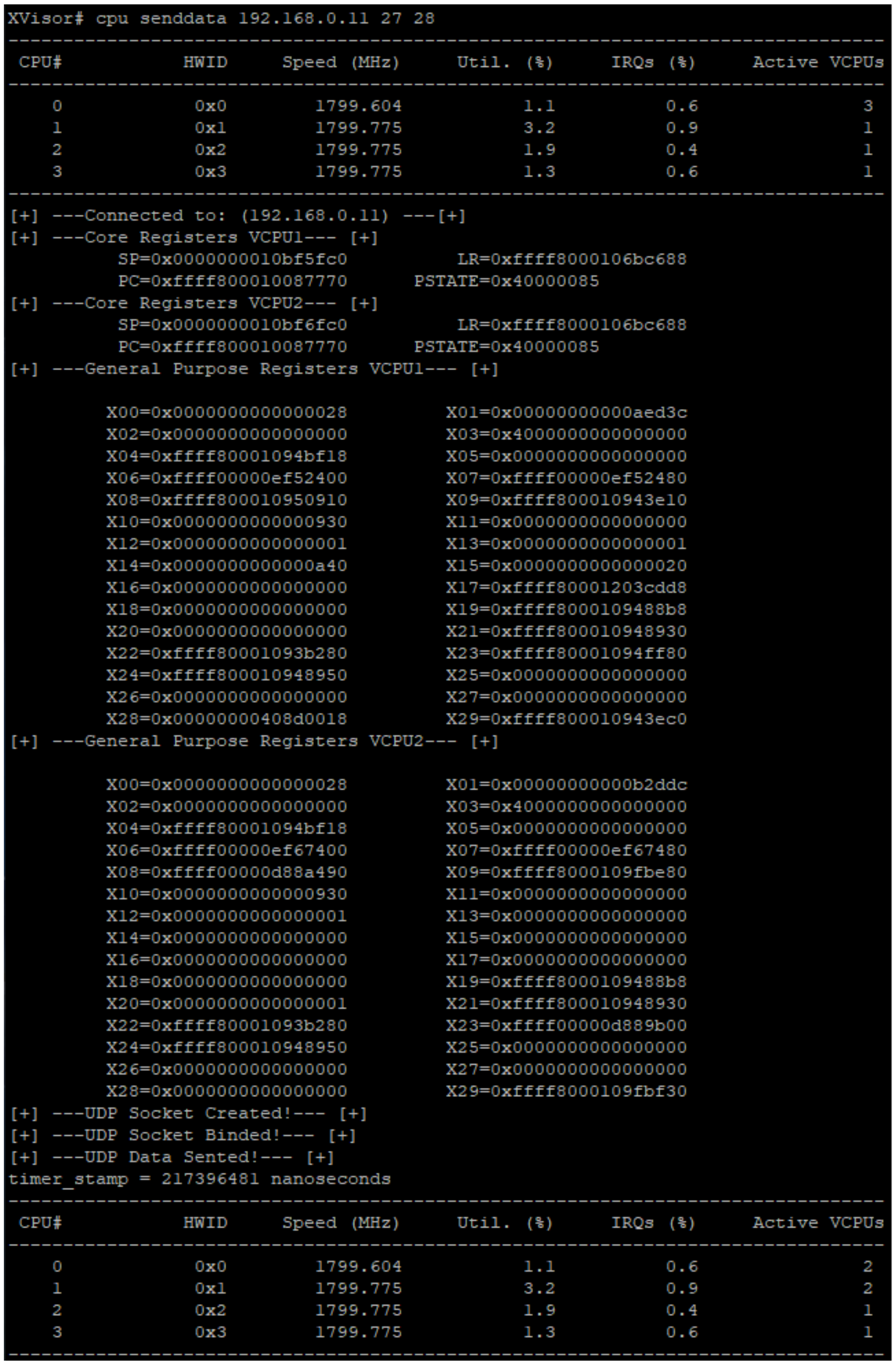


```
XVisor# host cpu stats
--------------------------------------------------------------------------------
 CPU#           HWID     Speed (MHz)      Util. (%)      IRQs (%)      Active VCPUs
--------------------------------------------------------------------------------
   0             0x0       1799.604             4.0           0.5                 3
   1             0x1       1799.775             3.2           0.9                 1
   2             0x2       1799.775             1.9           0.4                 1
   3             0x3       1799.775             1.3           0.6                 1
--------------------------------------------------------------------------------
```

Figure 8-30 Host CPU load with Guest workload

Finally, we trigger the CPU live migration process of the vCPUs that are assigned to the Guest instance, and we measure the load of the host CPUs during the migration process. Figure 8.31 shows the CPU load once the migration starts and at the completion of the process. We can see that there is neither performance degradation either CPU load spike when the migration takes place. Also, the active vCPUs before the migration took place was three, all assigned on physical CPU 0 and with the completion of the

CPU migration the Xvisor initiates and assigns a new vCPU to a separate physical host CPU. That’s a normal behaviour since the state of the vCPU transferred to the destination and in order to maintain the performance of the source host and not break or corrupt it initiates and assigns a new vCPU.



```
XVisor# cpu senddata 192.168.0.11 27 28
--------------------------------------------------------------------------------
 CPU#           HWID     Speed (MHz)      Util. (%)      IRQs (%)      Active VCPUs
--------------------------------------------------------------------------------
    0            0x0        1799.604           1.1           0.6               3
    1            0x1        1799.775           3.2           0.9               1
    2            0x2        1799.775           1.9           0.4               1
    3            0x3        1799.775           1.3           0.6               1
--------------------------------------------------------------------------------
[+] ---Connected to: (192.168.0.11) ---[+]
[+] ---Core Registers VCPU1--- [+]
          SP=0x0000000010bf5fc0          LR=0xffff8000106bc688
          PC=0xffff800010087770      PSTATE=0x40000085
[+] ---Core Registers VCPU2--- [+]
          SP=0x0000000010bf6fc0          LR=0xffff8000106bc688
          PC=0xffff800010087770      PSTATE=0x40000085
[+] ---General Purpose Registers VCPU1--- [+]

         X00=0x0000000000000028         X01=0x00000000000aed3c
         X02=0x0000000000000000         X03=0x4000000000000000
         X04=0xffff80001094bf18         X05=0x0000000000000000
         X06=0xffff00000ef52400         X07=0xffff00000ef52480
         X08=0xffff800010950910         X09=0xffff800010943e10
         X10=0x0000000000000930         X11=0x0000000000000000
         X12=0x0000000000000001         X13=0x0000000000000001
         X14=0x0000000000000a40         X15=0x0000000000000020
         X16=0x0000000000000000         X17=0xffff80001203cdd8
         X18=0x0000000000000000         X19=0xffff8000109488b8
         X20=0x0000000000000000         X21=0xffff800010948930
         X22=0xffff80001093b280         X23=0xffff80001094ff80
         X24=0xffff800010948950         X25=0x0000000000000000
         X26=0x0000000000000000         X27=0x0000000000000000
         X28=0x00000000408d0018         X29=0xffff800010943ec0
[+] ---General Purpose Registers VCPU2--- [+]

         X00=0x0000000000000028         X01=0x00000000000b2ddc
         X02=0x0000000000000000         X03=0x4000000000000000
         X04=0xffff80001094bf18         X05=0x0000000000000000
         X06=0xffff00000ef67400         X07=0xffff00000ef67480
         X08=0xffff00000d88a490         X09=0xffff8000109fbe80
         X10=0x0000000000000930         X11=0x0000000000000000
         X12=0x0000000000000001         X13=0x0000000000000000
         X14=0x0000000000000000         X15=0x0000000000000000
         X16=0x0000000000000000         X17=0x0000000000000000
         X18=0x0000000000000000         X19=0xffff8000109488b8
         X20=0x0000000000000001         X21=0xffff800010948930
         X22=0xffff80001093b280         X23=0xffff00000d889b00
         X24=0xffff800010948950         X25=0x0000000000000000
         X26=0x0000000000000000         X27=0x0000000000000000
         X28=0x0000000000000000         X29=0xffff8000109fbf30
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 217396481 nanoseconds
--------------------------------------------------------------------------------
 CPU#           HWID     Speed (MHz)      Util. (%)      IRQs (%)      Active VCPUs
--------------------------------------------------------------------------------
    0            0x0        1799.604           1.1           0.6               2
    1            0x1        1799.775           3.2           0.9               2
    2            0x2        1799.775           1.9           0.4               1
    3            0x3        1799.775           1.3           0.6               1
--------------------------------------------------------------------------------
```

Figure 8-31 Host CPU load during the CPU live migration

Summarizing in Figure 8.32 we observe that the CPU performance of the RPI remains on normal levels during the whole process and that the Xvisor hypervisor does not affect system's performance. We can notice that in Phase 1 where CPU load is extremely low while in Phases 2 and 3 CPU load is on regular levels. Furthermore, our scheme which takes action on Phase 4 does not affect CPU performance by adding extra overhead.

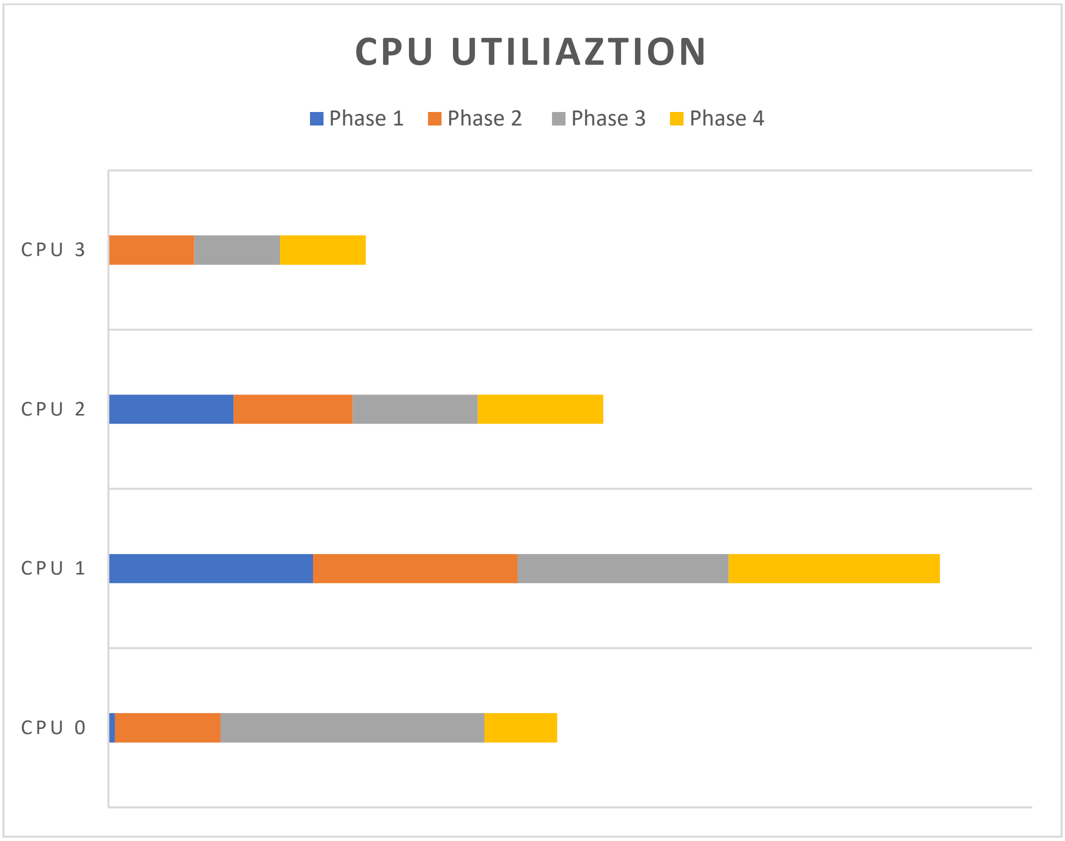


Figure 8-32 CPU load during the four states

8.7 Findings

In the previous section, we provided a demonstration of a CPU live migration process implemented on ARM based systems with the support of the Xvisor hypervisor. Xvisor takes advantage of the hardware-assisted virtualization characteristics and functionalities which allow a hypervisor to run on a higher, more privileged level than the Guest OS, which means we can gain access and retrieve a CPU state of both privileged and unprivileged CPU modes. The experiment as covered in this chapter, proves that our novel contribution successfully performs a CPU-only state migration via a UDP socket to a destination Linux system.

Although our definition of a CPU state is described as the content of the available registers of all the supported CPU modes, this may not be all that is needed in a live migration process in order to resume from the same point. Currently, there is no supported paper or research that specifies the strictly necessary number of groups of CPU registers that need to be migrated in order to maintain CPU state of a Guest instance between two hosts. However, taking into consideration the hardware-assisted approach on x86 architectures, where it manages content switch operations with the support of a memory-based structure called VMCS [59] that keeps the CPU state of both the host and Guest, we could define the potential total number of registers that are needed. Based on that, we evaluated the importance that the number of the migrated registers may have on the performance and completion time of the CPU migration.

Therefore, we performed our evaluation by considering a group of different numbers of CPU registers, measuring the time it takes for the completion of a CPU live migration. The outcome of the evaluation process, confirms and shows that defining the precise number of registers that compose a CPU state can affect the performance of a CPU state live migration process. ARM provides great flexibility and the freedom to choose what state and the number of registers that should be stored and then restored on a system. Therefore, on ARM we can configure and achieve a higher level of performance and reduction of the overall CPU migration phase providing the right number of registers.

9 Conclusions and Future Work

This chapter gives a conclusion of our thesis while we discuss additional improvements and challenges that can be fulfilled.

9.1 Thesis Conclusion

Virtual machine live migration finds extensive implementation on virtualized infrastructures, however, the demand for higher computing performance, network consistency, finer security and privacy acted as a catalyst for Bare metal instances on demand. A major limitation of Bare metal instances however is that it does not support live migration due to the lack of a virtualization layer. However, a few studies have now succeeded in addressing and developing a live migration scheme for Bare metal instances, some of those never found practical implementation while others are applicable only to x86 architectures. At the time that this thesis was written, no existing live migration schemes are available on ARM based systems. Without that availability, system continuity, service reliability and workflow load balancing on ARM based systems become extremely challenging tasks.

The shift from a cloud based, centralised architectural model to the Edge using Edge Computing paradigms brings Micro Data Centres to the fore. More and more practises in the research community explore the utilization and adoption of ARM based platforms to the edge as edge computation points. Inspired by this we suggest and explore the functionality of Micro DCs consisted by Bare metal ARM based systems powered by RPIs where the availability of live migration schemes becomes valuable tool for keeping in sync the state among that distributed architecture.

In order to develop and perform live migration on Bare metal instances, low level access with sufficient permissions and privileges to the hardware is needed in order to achieve direct communication with several hardware components such as memory, peripherals, network etc. Therefore, in our implementation we decided to use a thin hypervisor layer of through which we can gain access to the components and processes that we need, while allowing our live migration scheme to become portable finding availability on a wider list of systems without being limited to a single vendor and system. Our design takes advantage of the distinct exception levels that the ARM architecture provides, where the hypervisor runs on the highest exception level with the highest permissions to the underlying hardware infrastructure. Through that level we gain access to all the rest of the exception levels, CPU modes and hardware components that we need in order to compose and prepare the Guest state for migration.

Furthermore, our implementation allows us to read and store the CPU state at a given time, meaning that the programmer has the capability to decide and choose what state they want to capture. This can either be the CPU state of the Guest instance or the CPU state of some of the processes running on the host system.

In the literature there is no definition about what is considered as "state of a Guest instance", including the type and number of registers that make it up. When a VM live migration takes place the state of the Guest instance is stored in a file that is transferred from the source to the destination while when the same process takes place on x86 Bare metal instances the state is preserved through the high-speed data structure called VMCS that is designed to hold Guest state. A VMCS data structure is divided into 6 areas, one of which is Guest-state area that describes the Guest state. However, during the execution of a Bare metal live migration process, the entire VMCS structure is migrated which means that it consumes more storage, bandwidth and increases migration time in order to migrate the Guest state as well as other configuration parameters which are none highly importance in our design and implementation. Our approach takes advantage of the flexibility that ARM architecture gives to developers to choose the state and registers which want to save, and restore, allowing us to achieve higher performance, saving more bandwidth, while completing the CPU migration state in less time by focusing on the required registers that are necessary to properly reassembly Guest's state on destination system. Although our implementation has higher difficulty than other migration solutions since it requires a custom configuration, it is considered better approach due to the benefits that provides.

Our evaluation experiment proves and highlights that the size of the migrated state affects the migration time and consequently the total migration time. As we analysed and explained, the higher the number of migrated registers, the longer the migration time which opens a new subject for investigation and further discussion making important the exploration of the required registers that are needed to reassemble and compose the state of the Guest instance at the destination host after the live migration took place.

9.2 Future Work

Our novel live migration design as described in this thesis can be adopted by many use cases while being applied to a broader type of platforms, covering a wider range of ARM processor versions. However, the current CPU live migration scheme has some limitations that could be covered in future work. This section presents considerations for future work.

- Compatibility with other hypervisors

Although our design and implementation focus on CPU state migration utilizing the Xvisor hypervisor, is not limited and tied specific to that hypervisor. Our implementation could work generally on any hypervisor that meets the requirements with just few adjustments as well as could work on ARMv7 architecture. For example, in a case scenario of XEN hypervisor, although it does not meet the specifications based on the requirements as explained in Section 6.3.1, there is a special structure that defines the VCPU registers similar to Xvisor (Appendix C). The Figure 9.1 shows an example where we store both the 32bit and 64bit values of the registers into a buffer, like we did in our implementation based on Xvisor's structure. As we can see, we only adjust the naming format of the registers as declared and defined by XEN in the source code.

```
/*Put the values of register inside the array*/
   data_buffer[0] = regs->__DECL_REG(x0,r0);
   data_buffer[1] = regs->__DECL_REG(x1,r1);
   data_buffer[2] = regs->__DECL_REG(x2,r2);
   data_buffer[3] = regs->__DECL_REG(x3,r3);
```

Figure 9-1 Storing register values based on XEN VCPU structure

- Improve network reliability by using TCP

The current version of Xvisor (0.2.11) although supporting TCP as a transport protocol, does not support all TCP functionalities in order to initialize, send and receive data over a TCP channel. To achieve a fully functional TCP communication, we need to adjust and configure all the LwIP API calls related to TCP. The Xvisor developers are working on this task and this functionality will potentially be supported in a future release. Then we could modify our CPU live migration scheme to make use of a TCP channel instead of the UDP during the creation process of the socket. Working over a TCP channel increases the integrity and consistency of the migrated state through the network during the live migration process.

However, our implementation is intended to take place over a private network within a MicroDC which consists of two hosts where network traffic is low with no or limited external interventions, so, altering or affecting the data travelling from the source to destination is considered unlikely to occur. Adopting TCP over UDP will guarantee that migrated data will reach the destination host, arriving in the right order.

- Compatibility with ARMv7

The ARM architecture has many versions with several changes and core differences among them. Our live migration scheme is designed and developed based on the ARMv8 architecture, which makes it incompatible with previous ARM versions. However, we could implement backward compatibility with older versions such as ARMv7 or with the upcoming versions (ARMv9) by adding the right naming conversions of the registers as referred to by each processor developer's guide. Through that feature we could perform a live migration between systems of the same processor architecture or potentially a mix of them.

Two main differences between ARM versions are the naming formats of the supported registers and the range and type of available registers. In order to provide backward compatibility with older ARM versions such as ARMv7, we need to include those parameters. This can be done either by including separate functions delivered by a distinct menu for each architecture, for example, choosing when you want to perform and execute a CPU live migration between ARMv7 or ARMv8 or by discovering dynamically the underlying architecture and choosing to migrate the proper group of registers.

In both cases, we need to hardcode and cover the naming formats for both ARMv7 and ARMv8 architectures. Inter-compatibility among ARM versions gives us the ability to perform our CPU live migration scheme on a wider range of hosted platforms. For example, a RPI of ARMv7 architecture has 32 bits of registers while a RPI of ARMv8 architecture supports both 32 and 64bit of registers. So, in order to perform a CPU state migration on a RPI 2, pointers need to refer to the sample of code in Figure 9.1 where that structure defines the group of the supported architecture registers with the right naming definitions while on a RPI 3 model B, pointers need to refer to the sample of code as defined in Figure 9.2. As we can see, the variables type definition differs while the range of the registers varies too.

```
struct arch_regs {
        /* CPSR */
        u32 cpsr;
        /* Program Counter */
        u32 pc;
        /* R0 - R12 */
        u32 gpr[CPU_GPR_COUNT];
        /* Stack Pointer */
        u32 sp;
        /* Link Register */
        u32 lr;
} __packed;
```

Figure 9-2 ARMv7 architecture registers structure definition

```
struct arch_regs {
            /* X0 - X29 */
            u64 gpr[CPU_GPR_COUNT];
            /* Link Register (or X30) */
            u64 lr;
            /* Stack Pointer */
            u64 sp;
            /* Program Counter */
            u64 pc;
            /* PState/CPSR */
            u64 pstate;
} __packed;
```

Figure 9-3 ARMv8 architecture registers structure definition

- Implementing automation

The current proof-of-concept implementation requires many configuration changes to be performed manually in order to prepare the infrastructure and initiate the CPU live migration process, something that could lead to mistakes. Depending on the processor's architecture and execution state, the hypervisor can automatically gather and read the CPU registers which are assigned to the Guest instance that we need to migrate.

This is feasible since during the compilation of the source code of the Xvisor we define the number of the vCPU that we want to assign to our Guest instance. Furthermore, the Xvisor creates and assigns vCPUs to each of the background processes that remains constant throughout its operation. So, we can calculate the ID numbers of the vCPUs that belong to the Guest instance and pass them directly when the CPU live migration process is called. Through that process, we reduce the configuration time and the risk of a potential misconfiguration that could affect the CPU live migration process or even end in failure due to invalid data passing.

- Extend from Unicast to a Multicast network

Like most live migration schemes in the literature, similar, current implementations cover the performance of the CPU live migration process between two hosts as shown in Figure 9.1 at the left side. However, some use cases such as those we mentioned in Section 8.2 where the infrastructure consists of a cluster of RPIs, may need to perform a CPU live migration to more than a single board as the right side of Figure 9.3 illustrates. In order to perform live migration on such an architecture, we need to change the type of the socket. LwIP supports the configuration of a multicast group where we can attach and configure all the nodes in the network to be part of that multicast group. Furthermore, we need to make use of a valid multicast IP addressing scheme. Using a multicast network instead of a unicast could be efficient in cases where a bundle of RPIs work on the same task sharing processing power. A hardware failure could bring down the entire cluster so, migrating the state of the cluster to another cluster could be achieved through that process. This could be a great feature to Micro DCs where the demand of high compute performance leads to the formation of platforms in bundles work as a unit, in order to share workload computation, performing faster processing rates. Keeping all platforms in the bundle in sync after the migration process requires the support of a multicast network-based transmission.

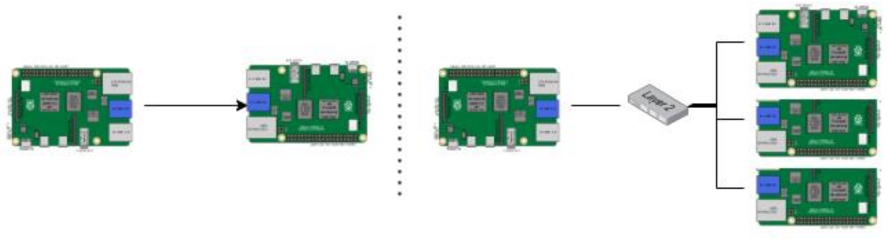


Figure 9-4 Live migration on a multicast network

9.3 Concluding Remarks

The design and development of the approach as presented in this thesis introduces a novel CPU live migration scheme applicable specifically on ARM based systems. Currently no other such scheme exists which is available on ARM. Our approach finds implementation through the Xvisor hypervisor covering a wide range of hardware systems and not limited only on Raspberry PI boards where our experiments took place. We conducted a series of experiments to evaluate the reliability of the process as well as to prove the flexibility that we have on reading registers from the CPU.

Our novel approach provides enhancements to modern Edge Computing networks especially in those that utilize clusters of single boards such as RPIs to offer higher availability and fault tolerance. By utilizing our scheme, the CPU state can be preserved, stored and transferred from a single RPI board to another over the network.

Another contribution of this thesis was to investigate the amount of data that constitutes the essential CPU state for a live migration. A CPU is described by several registers as we mentioned in Chapter 6. Some of them are responsible for handling the execution of the running applications and the communication with the underlying hardware resources while others are responsible for the general operation of the processor itself that are not critical in process of the reassembly process of the CPU state at the destination. Migrating the entire CPU footprint affects and slows down the live migration process. Therefore, it was essential to evaluate the performance of the CPU live migration process in relation to the number of registers that need to be transferred in the shortest possible time while maintaining the Guest state intact. Our results illustrate the effect of this on total migration time and highlight the gap that exists in the literature about the existence of a definition about what is considered as "Guest CPU state" and the need to establish it in order to achieve the best possible time for its transfer. This is very useful for lowering network overheads and gaining more available bandwidth since more the data that flows through the network, greater amount of bandwidth is required while higher latency is occurred.

References

[1] Ahmad, R., Gani, A., Ab. Hamid, S., Shiraz, M., Xia, F. and Madani, S., 2015. Virtual machine migration in cloud data centers: a review, taxonomy, and open research issues. The Journal of Supercomputing, 71(7), pp.2473-2515.

[2] Cloud Computing Characteristics and Services: A Brief Review. Available at: https://www.researchgate.net/publication/331731714_Cloud_Computing_Characteristics_and_Services_A_Brief_Review (Accessed: 26 December 2019).

[3] Docker Containers Versus Virtual Machine-Based Virtualization: Proceedings of IEMIS 2018, Volume 3 (no date). Available at: https://www.researchgate.net/publication/327389502_Docker_Containers_Versus_Virtual_Machine-Based_Virtualization_Proceedings_of_IEMIS_2018_Volume_3 (Accessed: 4 January 2020).

[4] AArch64 Virtualization Connect User Guide AArch64 Virtualization (2017). Available at: http://www.arm.com (Accessed: 23 February 2019).

[5] Acharya, S. and D'Mello, D. A. (2013) 'A taxonomy of live virtual machine (VM) migration mechanisms in cloud computing environment', Proceedings of the 2013 International Conference on Green Computing, Communication and Conservation of Energy, ICGCE 2013. IEEE, (Vm), pp. 809–815.

[6] Agarwal, A. and Raina, S. (2012) 'Live Migration of Virtual Machines in Cloud', International Journal of Scientific and Research Publications, 2(6). Available at: www.ijsrp.org (Accessed: 24 September 2018).

[7] Algarni, Abdullah & Ikbal, Mohammad & Alroobaea, Roobaea & Ghiduk, Ahmed & Nadeem, Farrukh. (2018). Performance Evaluation of Xen, KVM, and Proxmox Hypervisors. International Journal of Open Source Software and Processes. 9. 39-54. 10.4018/IJOSSP.2018040103.

[8] An Introduction to Edge Computing and A Real-Time Capable Server Architecture (no date). Available at: https://www.researchgate.net/publication/326441179_An_Introduction_to_Edge_Computing_and_A_Real-Time_Capable_Server_Architecture (Accessed: 3 August 2019).

[9] Arcserve | 1 Trends in Server Virtualization (2017). Available at: https://www.arcserve.com/wp-content/uploads/2016/07/virtualization-ebook.pdf (Accessed: 27 December 2018).

[10] ‘ARM ® Cortex ® -A Series Programmer’s Guide for ARMv8-A ARM Cortex-A Series Programmer’s Guide for ARMv8-A’ (no date).

[11] ARM ® Generic Interrupt Controller Architecture version 2.0 Architecture Specification (2008). Available at: https://www.cl.cam.ac.uk/research/srg/han/ACS-P35/zynq/arm_gic_architecture_specification.pdf (Accessed: 2 March 2019).

[12] Avelar, V. (no date) Cost Benefit Analysis of Edge Micro Data Center Deployments. Available at: http://www.edgeconnex.com/company/news-events/in-the-media/acg-research-produces-white-paper- (Accessed: 27 October 2019).

[13] Birje, M., Challagidad, P., Goudar, R. and Tapale, M., 2017. Cloud computing review: concepts, technology, challenges and security. International Journal of Cloud Computing, 6(1), p.32.

[14] Background: The ARM architecture’ (no date). Available at: https://www.arm.com/files/downloads/ARMv8_white_paper_v5.pdf (Accessed: 4 April 2017).

[15] Burns, P. (2014) ‘The IT benefits of Bare metal clouds’. Available at: http://static1.squarespace.com/static/567818bbe0327c06bcd8fe70/t/56f57a6a2eeb8139660676e6/1458928234355/The+IT+Benefits+of+Bare+Metal_Gigoam+Research.pdf.

[16] Buyya, R., Vecchiola, C. and Selvi, S. T. (2013) ‘Virtualization’, Mastering Cloud Computing, (1), pp. 71–109. doi: 10.1016/B978-0-12-411454-8.00003-6.

[17] Carlini, S. (no date) The Drivers and Benefits of Edge Computing Revision 0. Available at: http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p565.pdf (Accessed: 27 October 2019).

[18] Chierici, A. and Veraldi, R. (2010) ‘A quantitative comparison between xen and kvm’, in Journal of Physics: Conference Series. Institute of Physics Publishing. doi: 10.1088/1742-6596/219/4/042005.

[19] Choudhary, A. et al. (2017) ‘REVIEW Open Access A critical survey of live virtual machine migration techniques’, 6, p. 23. doi: 10.1186/s13677-017-0092-1.

[20] Clark, C. et al.Live Migration of Virtual Machines. Available at: https://www.cl.cam.ac.uk/research/srg/netos/papers/2005-migration-nsdi-pre.pdf (Accessed: 24 September 2018).

[21] Cloud Computing Tutorial for Beginners. Available at: https://www.guru99.com/cloud-computing-for-beginners.html (Accessed: 26 December 2019).

[22] Coursehero.com. 2020. Ch01. Cloud Computing - A Fresh Graduates Guide To Software Development Tools And Technologies 1 Chapter Cloud Computing CHAPTER AUTHORS Wong Tsz Lai | Course Hero. [online] Available at: <https://www.coursehero.com/file/14520142/Ch01-Cloud-Computing/> [Accessed 31 August 2020].

[23] Chevtchenko and Vale, n.d. A Comparison Of RISC And CISC Architectures. [online]. Available at: <http://ww2.deinfo.ufrpe.br/sites/ww2.deinfo.ufrpe.br/files/artigos_aoc/Artigo_2_final.pdf> [Accessed 31 August 2020].

[24] Dall, C. et al. (2016) 'ARM Virtualization: Performance and Architectural Implications', Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016, pp. 304–316. doi: 10.1109/ISCA.2016.35.

[25] Dall, C. et al. (no date) ARM Virtualization: Performance and Architectural Implications. Available at: http://www.cs.columbia.edu/~cdall/pubs/isca2016-dall.pdf (Accessed: 13 February 2019).

[26] Dall, C., Li, S.-W. and Nieh, J. (no date) 'Optimizing the Design and Implementation of the Linux ARM Hypervisor'. Available at: https://www.usenix.org/conference/atc17/technical-sessions/presentation/dall (Accessed: 25 July 2017).

[27] Dall, C. and Nieh, J., 2014. KVM/ARM. ACM SIGPLAN Notices, 49(4), pp.333-348.

[28] O'Callaghan, G. (2017). Digitalisation Executive summary and key results Outlook Digitalisation at Siemens Survey results and reference cases.

[29] Dijiang Huang, H. W. (2018) Edge Computing -Pushing the Boundary of Mobile Clouds. Available at: https://www.sciencedirect.com/topics/computer-science/edge-computing (Accessed: 4 September 2019).

[30] Ding, J.-H. et al. (no date) ARMvisor: System Virtualization for ARM. Available at: http://www.cs.nthu.edu.tw/~ychung/conference/ARMVisor.pdf (Accessed: 13 February 2019).

[31] Durairaj M (2014a) 'A Study On Virtualization Techniques And Challenges In Cloud Computing', INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH, 3(11). Available at: www.ijstr.org (Accessed: 29 August 2018).

[32] Durairaj M (2014b) ‘A Study On Virtualization Techniques And Challenges In Cloud Computing’, INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH, 3(11). Available at: www.ijstr.org (Accessed: 2 February 2020).

[33] Eder Betreuer, M., Kinkelin, H. and Netzarchitekturen, L. (no date) ‘Hypervisor-vs. Container-based Virtualization’. doi: 10.2313/NET-2016-07-1_01.

[34] Evans, D. (2012) The Internet of Everything How More Relevant and Valuable Connections Will Change the World. Available at: https://www.cisco.com/c/dam/global/en_my/assets/ciscoinnovate/pdfs/IoE.pdf (Accessed: 3 August 2019).

[35] Fathy, A., Abdelrazek, M. and Lücke, D. (no date) Exception and Interrupt Handling in ARM Architectures and Design Methods for Embedded Systems Summer Semester 2006. Available at: http://www.iti.uni-stuttgart.de/~radetzki/Seminar06/08_report.pdf (Accessed: 14 February 2019).

[36] Fedora 23 Virtualization Getting Started Guide Virtualization Documentation Fedora Documentation Project Virtualization Getting Started Guide Fedora 23 Virtualization Getting Started Guide Virtualization Documentation Edition 01 (2012). Available at: https://fedoraproject.org/wiki/ (Accessed: 3 February 2019).

[37] Fukai, T. et al. (2016) ‘OS-Independent Live Migration Scheme for Bare-Metal Clouds’, Proceedings - 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing, UCC 2015, pp. 80–89. doi: 10.1109/UCC.2015.23.

[38] Fukai, T., Shinagawa, T. and Kato, K., 2018. Live Migration in Bare-metal Clouds. IEEE Transactions on Cloud Computing, pp.1-1.

[39] Gavriil Kominos, C., Seyvet, N. and Vandikas, K. (no date) Bare-metal, Virtual Machines and Containers in OpenStack. Available at: https://www.ericsson.com/4a4354/assets/local/publications/conference-papers/bare_metal_virtual_machines_and_containers_in_openstack.pdf (Accessed: 5 September 2019).

[40] Golden, B. (2011) Virtualization for Dummies, 3rd HP Special Edition. Available at: http://www.hp.com/go/virtualization. (Accessed: 4 August 2018).

[41] Gouda, K. C. et al. (2014) 'Virtualization Approaches in Cloud Computing', International Journal of Computer Trends and Technology, 12. Available at: http://www.ijcttjournal.org (Accessed: 29 August 2018).

[42] GSMA (2018) Opportunities and Use Cases for Edge Computing in thE iot. Available at: www.gsma.com. (Accessed: 4 September 2019).

[43] GUIDELINE on SERVER CONSOLIDATION and VIRTUALISATION (no date). Available at: http://www.ncb.mu/English/Documents/Downloads/Reports and Guidelines/Guideline on Server Consolidation and Virtualisation.pdf (Accessed: 17 September 2018).

[44] Hassan, N. et al. (2018) 'The Role of Edge Computing in Internet of Things', IEEE Communications Magazine, 56(11), pp. 110–115. doi: 10.1109/MCOM.2018.1700906.

[45] Hennessey, J. et al. (2014) 'Hardware as a service -enabling dynamic, user-level Bare metal provisioning of pools of data center resources'. Available at: http://hdl.handle.net/2144/11221.

[46] Hu, W. et al. (2013) A Quantitative Study of Virtual Machine Live Migration. Available at: http://people.clarkson.edu/~lozhang/A Quantitative Study of Virtual Machine Live Migration.pdf (Accessed: 28 October 2018).

[47] Huth, A. and Cebula, J. (2011) 'The Basics of Cloud Computing'. Available at: https://www.us-cert.gov/sites/default/files/publications/CloudComputingHuthCebula.pdf (Accessed: 11 June 2018).

[48] IEEE Xplore Full-Text PDF: (no date). Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8759339 (Accessed: 9 February 2020).

[49] Industrial Internet Consortium (2018) Introduction to Edge Computing in IIoT. Available at: www.iiconsortium.org/IISF (Accessed: 3 August 2019).

[50] Iomart (2019) CloudSure Bare metal Cloud. Available at: www.iomart.com (Accessed: 5 September 2019).

[51] Iorga, M. et al. (no date) 'Fog Computing Conceptual Model: Recommendations of the National Institute of Standards and Technology'. doi: 10.6028/NIST.SP.500-325.

[52] International Journal of Recent Trends in Engineering and Research, 2017. Cloud Computing : Architecture, Challenges and Application. 3(6), pp.182-188.

[53] Joshi, M. R. and Tikar, B. V (2015b) International Journal of Computer Science and Mobile Computing Cloud-Computing its Services and Resent Trends, International Journal of Computer Science and Mobile Computing. Available at: www.ijcsmc.com (Accessed: 3 February 2019).

[54] Kim (2015) 'Prototype of Light-weight Hypervisor for ARM Server Virtualization Young-Woo Jung, Song-Woo Sok, Gains Zulfa Santoso, Jung-Sub Shin, and Hag-Young', Embedded Systems and Applications 215. Available at: http://worldcomp-proceedings.com/proc/p2015/ESA3346.pdf (Accessed: 14 August 2017).

[55] Korri, T. (2009) 'Cloud computing: utility computing over the Internet'. Available at: http://www.ndc3.net/resources/articles/Korri_2009.pdf (Accessed: 11 June 2018).

[56] Kumar, N., Kumar Kushwaha, S. and Kumar, A. (2014) Yamuna Expressway, Sector 17A, Greater Noida, UP-201306 India. Mtech-final year, Communication Engineering, Galgotias University. Plot No. 2,

[57] Kumar, R. and Charu, S. (2015) 'An Importance of Using Virtualization Technology in Cloud Computing', Global Journal of Computers & Technology, 1(2), pp. 56–60.

[58] Lee, H. (no date) Virtualization Basics: Understanding Techniques and Fundamentals. Available at: http://dsc.soic.indiana.edu/publications/virtualization.pdf (Accessed: 4 August 2018).

[59] Lettieri, G. (2015) 'Intel VMX technology', in. Available at: http://lettieri.iet.unipi.it/virtualization/2016/vn05.pdf (Accessed: 25 July 2017).

[60] Leymann, F. (no date) 'Cloud Computing: The Next Revolution in IT Institute of Architecture of Application Systems Cloud Computing: The Next Revolution in IT'. Available at: http://www.vde-verlag.de/buecher/537483/photogrammetric-week-09.html (Accessed: 11 June 2018).

[61] Linthicum, D. S. and Senf, V. (2014) Leveraging Bare metal clouds.

[62] Liu, G. (2012) 'Optimizing Live Migration of Virtual Machines', Bupt, 7161(May), pp. 1–14.

[63] M, H. G. (2013) 'PERFORMANCE ANALYSIS OF KERNEL-BASED VIRTUAL MACHINE', International Journal of Computer Science & Information Technology (IJCSIT), 5(1). doi: 10.5121/ijcsit.2013.5111.

[64] Malhotra L, Agarwal D and Jaiswal A (2014) 'Virtualization in Cloud Computing', J Inform Tech Softw Eng, 4, p. 136. doi: 10.4172/2165-7866.1000136.

[65] Mell, P. and Grance, T., 2020. The NIST Definition Of Cloud Computing. [online] Faculty.winthrop.edu. Available at: <http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf> [Accessed 31 August 2020].

[66] Microsystems, S. (2009) 'Introduction to Cloud Computing Architecture'. Available at: http://staff.polito.it/alessandro.mantelero/cloud_computing/Sun_Wp2009.pdf (Accessed: 11 June 2018).

[67] Mohsen, R. and Pinto, A. M. (2016) 'Theoretical foundation for code obfuscation security: A Kolmogorov complexity approach', in Communications in Computer and Information Science, pp. 245–269. doi: 10.1007/978-3-319-30222-5_12.

[68] Montvale, N. J. et al. (1973) Jones, A.K. Protection structures. Ph.D. Th., Carnegie-Mellon U., 1973. 30. Lampson, B.W. On reliable and extendable operating systems, 41. Shepherd, J. Principal design features of the multi-computer. ACM. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.4815&rep=rep1&type=pdf (Accessed: 5 August 2018).

[69] Nebbiolo Technologies Inc. (2016) Fog vs Edge Computing. Available at: https://www.nebbiolo.tech/wp-content/uploads/whitepaper-fog-vs-edge.pdf (Accessed: 4 September 2019).

[70] Nelson, M., Lim, B.-H. and Hutchins, G. (no date) Fast Transparent Migration for Virtual Machines. Available at: http://static.usenix.org/legacy/events/usenix05/tech/general/full_papers/short_papers/nelson/nelson.pdf (Accessed: 28 October 2018).

[71] Oludele, A. et al. (2014) 'On the Evolution of Virtualization and Cloud Computing: A Review', Journal of Computer Sciences and Applications, 2(3), pp. 40–43. doi: 10.12691/jcsa-2-3-1.

[72] Omote, Y., Shinagawa, T. and Kato, K. (2015) 'Improving Agility and Elasticity in Bare-metal Clouds', Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15, 50(4), pp. 145–159. doi: 10.1145/2694344.2694349.

[73] Openstack (2018) Cloud Edge Computing - Beyond the Data Center.

[74] Pasumarthy, S. C. (2015) Live Migration of Virtual Machines in the Cloud An Investigation by Measurements. Available at: www.bth.se (Accessed: 28 October 2018).

[75] Pearce, M. (2013) Virtualization: Issues, security threats, and solutions '39 pages', ACM Computing Surveys, 45(2), p. 17.

[76] Penneman, N. et al. (no date a) Formal virtualization requirements for the ARM architecture. Available at: https://users.elis.ugent.be/~brdsutte/research/publications/2013JSApenneman.pdf (Accessed: 11 February 2019).

[77] Perez-Botero, D., 2020. A Brief Tutorial On Live Virtual Machine Migration From A Security Perspective. [online] Semanticscholar.org. Available at: <https://www.semanticscholar.org/paper/A-Brief-Tutorial-on-Live-Virtual-Machine-Migration-Perez-Botero/b562a31a55998bab7fc416622e697844ae72f318> [Accessed 31 August 2020].

[78] Pierre, L. U. and Marie, E. T. (2014) 'THÈSE DE DOCTORAT CONJOINT TÉLECOM Architecture de sécurité de bout en bout et mécanismes d ' autoprotection pour les environnements Cloud'.

[79] Rad, P. et al. (2016) 'Benchmarking Bare metal Cloud Servers for HPC Applications', in Proceedings - 2015 IEEE International Conference on Cloud Computing in Emerging Markets, CCEM 2015.

[80] Ramachandra, G., Iftikhar, M. and Khan, F. A. (2017) 'A Comprehensive Survey on Security in Cloud Computing', in Procedia Computer Science. Elsevier B.V., pp. 465–472. doi: 10.1016/j.procs.2017.06.124.

[81] Rashid, A. and Chaturvedi, A. (2019) 'Cloud Computing Characteristics and Services A Brief Review', International Journal of Computer Sciences and Engineering. ISROSET: International Scientific Research Organization for Science, Engineering and Technology, 7(2), pp. 421–426.

[82] Reddy, Vuyyuru Krishna, Rathod, S. B. and Reddy, V Krishna (2014) 'Secure Live VM Migration in Cloud Computing: A Survey Self adaptive framework for secure VM migration over cloud computing View project Suresh B Rathod Sinhgad Technical Education Society Secure Live VM Migration in Cloud Computing: A Survey', Article in International Journal of Computer Applications, 103(2), pp. 975–8887.

[83] Rodríguez-Haro, F. et al. (2012) 'A summary of virtualization techniques', Procedia Technology, 3(February 2014), pp. 267–272.

[84] Rosenblum, M. (2004) 'The Reincarnation of Virtual Machines', Queue, 2(5), p. 34.

[85] Rishi Bhardwaj, E., 2020. A Choices Hypervisor On The ARM Architecture. [online] Citeseerx.ist.psu.edu. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary [Accessed 31 August 2020].

[86] Saleem, M. and Shah, G. (2017) Cloud Computing Virtualization, International Journal of Computer Applications Technology and Research. Available at: www.ijcat.com (Accessed: 29 August 2018).

[87] Salfner, F., Tröger, P. and Polze, A. (no date) Downtime Analysis of Virtual Machine Live Migration. Available at: https://citemaster.net/get/e61b2d78-b400-11e3-91be-00163e009cc7/salfner11downtime.pdf (Accessed: 10 February 2019).

[88] Sareen, P. (2013) Cloud Computing: Types, Architecture, Applications, Concerns, Virtualization and Role of IT Governance in Cloud, International Journal of Advanced Research in Computer Science and Software Engineering. Available at: www.ijarcsse.com (Accessed: 17 December 2018).

[89] Science, C., Kashyap, S. and Systems, E. (2014) 'An Enhanced Approach to Live Migration of Virtual Machines', (May).

[90] Scroggins, R. (2017) Emerging Virtualization Technology Emerging Virtualization Technology Emerging Virtualization Technology. Available at: https://globaljournals.org/GJCST_Volume17/3-Emerging-Virtualization-Technology.pdf (Accessed: 4 August 2018).

[91] Sharma, S. et al. (2016) 'Virtualization in Cloud Computing', 4(2), pp. 181–186.

[92] Simmon, E. (2017) Evaluation of Cloud Computing Services. Available at: https://www.nist.gov/sites/default/files/documents/2017/05/31/evaluation_of_cloud_computing_services_based_on_nist_800-145_20170427clean.pdf (Accessed: 18 June 2018).

[93] Sloss, A. N. (2001) Interrupt handling. Available at: https://ece.umd.edu/class/enee447.S2016/ARM-Documentation/ARM-Interrupts-1.pdf (Accessed: 2 March 2019).

[94] Sloss, A. N. 'ARM System Developer's Guide Designing and Optimizing System Software About the Authors'. Available at: http://eee.guc.edu.eg/Courses/Electronics/ELCT912 Advanced Embedded Systems/Lectures/ARM System Developer%27s Guide.pdf (Accessed: 7 June 2017).

[95] Stadtmueller, L. (2014) Bare metal Cloud: A Non-Virtualized Cloud Option for Performance-Sensitive Workloads Stratecast Perspectives &amp; Insight for Executives (SPIE) Volume 14, Number 2 Bare metal

Cloud: A Non-Virtualized Cloud Option for Performance-Sensitive Workloads 2. Available at: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf (Accessed: 5 September 2019).

[96] Sun, G. et al. (2016) 'A new technique for efficient live migration of multiple virtual machines', Future Generation Computer Systems, 55, pp. 74–86. doi: 10.1016/j.future.2015.09.005.

[97] Suzuki, A. and Oikawa, S. (2011) 'Implementing a simple trap and emulate VMM for the ARM architecture', in Proceedings - 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2011. IEEE, pp. 371–379. doi: 10.1109/RTCSA.2011.26.

[98] Svärd, P. (2012) Live VM Migration Principles and Performance. Available at: http://www.diva-portal.org/smash/get/diva2:707793/FULLTEXT02 (Accessed: 28 October 2018).

[99] Svorobej, S. et al. (2019) 'future internet Simulating Fog and Edge Computing Scenarios: An Overview and Research Challenges'. doi: 10.3390/fi11030055.

[100] The Challenges of Virtualization and the Popek and Goldberg Principles (no date). Available at: https://www.usenix.org/system/files/login/articles/105498-Revelle.pdf (Accessed: 19 February 2019).

[101] Tsz Lai, W., Trancong, H. and Goh, S. (2012) 'A Fresh Graduate's Guide to Software Development Tools and Technologies Chapter'.

[102] Turk, A. et al. (no date) 'An Experiment on Bare-Metal BigData Provisioning'.

[103] Union, I. T. (2009) Other reports in the series include: #1 Intelligent Transport System and CALM #2 Telepresence: High-Performance Video-Conferencing #3 ICTs and Climate Change #4 Ubiquitous Sensor Networks #5 Remote Collaboration Tools #6 Technical Aspects of Lawful Interc. Available at: https://www.itu.int/dms_pub/itu-t/oth/23/01/T23010000090001PDFE.pdf (Accessed: 11 June 2018).

[104] Varanasi, P. (2010) 'Implementing Hardware-supported Virtualization in OKL4 on ARM Prashant Varanasi Supervisor : Gernot Heiser', B of Sc. Thesis.

[105] Varanasi, P. and Heiser, G. (2011a) 'Hardware-supported virtualization on ARM', in Proceedings of the Second Asia-Pacific Workshop on Systems - APSys '11. New York, New York, USA: ACM Press, p. 1. doi: 10.1145/2103799.2103813.

[106] Varanasi, P. and Heiser, G. (2011c) 'Hardware-supported virtualization on ARM', in Proceedings of the Second Asia-Pacific Workshop on Systems - APSys '11. New York, New York, USA: ACM Press, p. 1. doi: 10.1145/2103799.2103813.

[107] Varanasi, P. and Heiser, G. (2011d) ‘Hardware-Supported Virtualization on ARM’, ApSys’11, p. 11. doi: 10.1145/2103799.2103813.

[108] Venkatesha, S., Sadhu, S. and Kintali, S. (2009) ‘Survey of Virtual Machine Migration Techniques’, Memory, (June), pp. 1–10.

[109] Virtualization Is the New Normal in Small Business | BizTech Magazine (no date). Available at: https://biztechmagazine.com/article/2013/11/virtualization-new-normal-small-business (Accessed: 7 August 2019).

[110] Yadav, A. K., Garg, M. L. and Ritika (2019) ‘Docker containers versus virtual machine-based virtualization’, in Advances in Intelligent Systems and Computing. Springer Verlag, pp. 141–150

[111] Zhang, F. (no date) Challenges and New Solutions for Live Migration of Virtual Machines in Cloud Computing Environments. Available at: https://d-nb.info/1160442126/34 (Accessed: 28 October 2018).

[112] Zhang, Q. et al. (no date) A Comparative Study of Containers and Virtual Machines in Big Data Environment.

[113] Zhang, Q., Cheng, L. and Boutaba, R. (2010) ‘Cloud computing: state-of-the-art and research challenges’, J Internet Serv Appl, 1, pp. 7–18.

[114] Zheng, J. (no date) Virtual Machine Live Migration in Cloud Computing. Available at: https://scholarship.rice.edu/bitstream/handle/1911/77591/thesis.pdf?sequence=1 (Accessed: 28 October 2018).

[115] PuTTY SSH and Telnet Client. Available at: https://www.chiark.greenend.org.uk/~sgtatham/putty/

[116] Komninos, A. et al. “Performance of Raspberry Pi microclusters for Edge Machine Learning in Tourism.” AmI (2019). At: https://www.semanticscholar.org/paper/Performance-of-Raspberry-Pi-microclusters-for-Edge-Komninos-Simou/b7464258deb91e358c19f4620f8a7c9f5a14699d

[117] C. Pahl, S. Helmer, L. Miori, J. Sanin and B. Lee, "A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters," 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), Vienna, 2016, pp. 117-124, doi: 10.1109/W-FiCloud.2016.36.

[118] Dall, C., & Nieh, J. (2014). KVM/ARM: the design and implementation of the linux ARM hypervisor. ASPLOS.

[119] C. G. Kominos, N. Seyvet and K. Vandikas, "Bare-metal, virtual machines and containers in OpenStack," 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), Paris, 2017, pp. 36-43, doi: 10.1109/ICIN.2017.7899247.

[220] Jaeseong Im, Jongyul Kim, Jonguk Kim, Seongwook Jin, and Seungryoul Maeng. 2017. On-demand virtualization for live migration in Bare metal cloud. In Proceedings of the 2017 Symposium on Cloud Computing.Association for Computing Machinery, New York, NY, USA, 378–389.

[221] J. Tang, D. Sun, S. Liu and J. Gaudiot, "Enabling Deep Learning on IoT Devices," in Computer, vol. 50, no. 10, pp. 92-96, 2017, doi: 10.1109/MC.2017.3641648.

[222] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. 2015. Improving Agility and Elasticity in Bare-metal Clouds. SIGPLAN Not. 50, 4 (April 2015), 145–159.

[223] UN DESA | United Nations Department of Economic and Social Affairs. 2018. 68% of the world population projected to live in urban areas by 2050, says UN | UN DESA | United Nations Department of Economic and Social Affairs. [online] Available at: <https://www.un.org/development/desa/en/news/population/2018-revision-of-world-urbanization-prospects.html> [Accessed 18 February 2021].

[224] Smith, S., 2021. ‘Internet of Things’ Connected Devices to Almost Triple to Over 38 Billion Units by 2020. [online] Juniperresearch.com. Available at: <https://www.juniperresearch.com/press/press-releases/iot-connected-devices-to-triple-to-38-bn-by-2020> [Accessed 18 February 2021].

[225] Calif, S.J. (2018). Global Cloud Index Projects Cloud Traffic to Represent 95 Percent of Total Data Center Traffic by 2021. [online] newsroom.cisco.com. Available at: https://newsroom.cisco.com/press-release-content?type=webcontent&articleId=1908858.

[226] Kaylie Gyarmathy (2020). Comprehensive Guide to IoT Statistics You Need to Know in 2019. [online] Vxchnge.com. Available at: https://www.vxchnge.com/blog/iot-statistics [Accessed 18 Feb. 2021].

[227] November 2019, 1st (2019). Digital Manufacturing and the IIoT – Success with a Single Platform. [online] IoT World Today. Available at: https://www.iotworldtoday.com/2019/11/01/digital-manufacturing-and-the-iiot/ [Accessed 18 Feb. 2021].

[228] Srivastava, S. (2017). IoT architecture: To run on the cloud or not? [online] CIO. Available at: https://www.cio.com/article/3231654/iot-architecture-to-run-on-the-cloud-or-not.html [Accessed 18 Feb. 2021].

[229] Symanovich, S. (2019). Norton. [online] Norton.com. Available at: https://us.norton.com/internetsecurity-iot-5-predictions-for-the-future-of-iot.html [Accessed 2021].

[230] Wilczek, M. (2018). IT governance critical as cloud adoption soars to 96 percent in 2018. [online] CIO. Available at: https://www.cio.com/article/3267571/it-governance-critical-as-cloud-adoption-soars-to-96-percent-in-2018.html.

[231] McKendrick, J. (2016). With Internet Of Things And Big Data, 92% Of Everything We Do Will Be In The Cloud. [online] Forbes. Available at: https://www.forbes.com/sites/joemckendrick/2016/11/13/with-internet-of-things-and-big-data-92-of-everything-we-do-will-be-in-the-cloud/?sh=12436aa14ed5 [Accessed 19 Feb. 2021].

[232] Raspberry Pi cloud cluster experiment," in Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom, 2013, vol. 2, pp. 170–175.

[233] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Riviere, "On using micro-clouds to deliver the fog," IEEE Internet Comput., vol. 21, no. 2, pp. 8–15, Mar. 2017.

[234] A. J. Ferrer, J. M. Marqués, and J. Jorba, "Ad-Hoc edge cloud: a framework for dynamic creation of Edge Computing infrastructures," in Proceedings - International Conference on Computer Communications and Networks, ICCCN, 2019, vol. 2019-July.

[235] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud PaaS architecture based on raspberry Pi clusters," in Proceedings - 2016 4th International Conference on Future Internet of Things and Cloud Workshops, W-FiCloud 2016, 2016, pp. 117–124.

[236] A. Javed, K. Heljanko, A. Buda, and K. Framling, "CEFIoT: A fault-tolerant IoT architecture for edge and cloud," in IEEE World Forum on Internet of Things, WF-IoT 2018 - Proceedings, 2018, vol. 2018-January, pp. 813–818.

[237] R. Scolati, I. Fronza, N. El Ioini, A. Samir, and C. Pahl, "A containerized big data streaming architecture for edge cloud computing on clustered single-board devices," in CLOSER 2019 - Proceedings of the 9th International Conference on Cloud Computing and Services Science, 2019, pp. 68–80.

[238] Nongnu.org. 2021. lwIP: Socket API. [online] Available at: <https://www.nongnu.org/lwip/2_0_x/group__socket.html> [Accessed 7 May 2021].

[239] Man7.org. 2021. socket(2) - Linux manual page. [online] Available at: <https://man7.org/linux/man-pages/man2/socket.2.html> [Accessed 7 May 2021].

[240] Intel. (2016). Intel ® 64 and IA-32 Architectures Software Developer’s Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide. In System (Vol. 3, Issue 253665). http://www.intel.com/design/literature.htm.

[241] Osman, S., Subhraveti, D., Su, G., & Nieh, J. (2002). The Design and Implementation of Zap : A System for Migrating Computing Environments Department of Computer Science 1 Introduction. SIGOPS Oper. Syst. Rev., 36(SI), 361–376.

[242] Arrow Business Communications. 2021. IoT devices 'to generate nearly 80 zettabytes of data' by 2025 - Arrow. [online] Available at: <https://www.arrowcommunications.co.uk/iot-devices-to-generate-nearly-80-zettabytes-of-data-by-2025/> [Accessed 13 May 2021].

[243] Abrahamsson, P., Helmer, S., Phaphoom, N., Nicolodi, L., Preda, N., Miori, L., Angriman, M., Rikkilä, J., Wang, X., Hamily, K., & Bugoloni, S. (2013). Affordable and energy-efficient cloud computing clusters: The Bolzano Raspberry Pi cloud cluster experiment. Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom, 2, 170–175.

[244] Bleeker, Ellen-Louise, Reinholdsson, M., Andersson ----Examiner, K., & Alfredsson , S. (2017). Creating a Raspberry Pi-Based Beowulf Cluster C-level thesis 15hp.

[245] Elkhatib, Y., Porter, B., Ribeiro, H. B., Zhani, M. F., Qadir, J., & Riviere, E. (2017). On using micro-clouds to deliver the fog. IEEE Internet Computing, 21(2), 8–15. https://doi.org/10.1109/MIC.2017.35

[246] Komninos, A., Simou, I., Gkorgkolis, N., & Garofalakis, J. (2019). Performance of Raspberry Pi microclusters for Edge Machine Learning in Tourism. Undefined.

[247] Mahmud, R., & Toosi, A. N. (2021). Con-Pi: A Distributed Container-based Edge and Fog Computing Framework for Raspberry Pis. http://arxiv.org/abs/2101.03533

[248] Shirer, M. and Murray, S., 2020. IDC Reveals 2021 Worldwide Digital Transformation Predictions; 65% of Global GDP Digitalized by 2022, Driving Over $6.8 Trillion of Direct DX Investments from 2020 to

2023. [online] IDC: The premier global market intelligence company. Available at: <https://www.idc.com/getdoc.jsp?containerId=prUS46967420> [Accessed 20 May 2021].

[249] Patel, A., 2017. Xvisor: eXtensible Versatile hypervISOR. [online] Xhypervisor.org. Available at: <http://xhypervisor.org/> [Accessed 20 May 2021].

[250] Michael A. Kozuch, Michael Kaminsky, and Michael P. Ryan. 2009. Migration without virtualization. In Proceedings of the 12th conference on Hot topics in operating systems (HotOS'09). USENIX Association, USA, 10.

[251] Z. Zhao, G. Min, W. Gao, Y. Wu, H. Duan and Q. Ni, "Deploying Edge Computing Nodes for Large-Scale IoT: A Diversity Aware Approach," in IEEE Internet of Things Journal, vol. 5, no. 5, pp. 3606-3614, Oct. 2018

[252] G. Premsankar, M. Di Francesco and T. Taleb, "Edge Computing for the Internet of Things: A Case Study," in IEEE Internet of Things Journal, vol. 5, no. 2, pp. 1275-1284, April 2018

[253] B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas and Q. Zhang, "Edge Computing in IoT-Based Manufacturing," in IEEE Communications Magazine, vol. 56, no. 9, pp. 103-109, Sept. 2018

[254] C. Martín Fernández, M. Díaz Rodríguez and B. Rubio Muñoz, "An Edge Computing Architecture in the Internet of Things," 2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC), 2018, pp. 99-102

[255] R. Das and S. Sidhanta, "LIMOCE: Live Migration of Containers in the Edge," 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2021, pp. 606-609

[256] P. Bellavista, A. Zanni and M. Solimando, "A migration-enhanced Edge Computing support for mobile devices in hostile environments," 2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC), 2017, pp. 957-962

[257] S. Wang, J. Xu, N. Zhang and Y. Liu, "A Survey on Service Migration in Mobile Edge Computing," in IEEE Access, vol. 6, pp. 23511-23528, 2018

[258] Miori, Lorenzo & Sanin, Julian & Helmer, Sven. (2017). A Platform for Edge Computing Based on Raspberry Pi Clusters. 153-159. 10.1007/978-3-319-60795-5_16.

[259] Richard McDougall and Jennifer Anderson. 2010. Virtualization performance: perspectives and challenges ahead. (December 2010), 40–56.

[260] Tarapra, Madhuri & Jadeja, Mitrarajsinh & John, Jeba Praba & Sridaran, R. (2017). Classification of Performance Degradation Issues In Virtualized Cloud Environments. 10.5281/zenodo.4384751.

[261] 7 ways we harden our KVM hypervisor at Google Cloud: security in plaintext: 2017. https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext. Accessed: 2021- 08- 25.

[262] AWS AMI Virtualization Types: HVM vs PV (Paravirtual VS Hardware VM): 2019. https://cloudacademy.com/blog/aws-ami-hvm-vs-pv-paravirtual-amazon/. Accessed: 2021- 08- 25.

[263] Azure Availability Zones | Build5Nines: 2018. https://build5nines.com/azure-availability-zones-public-preview/. Accessed: 2021- 08- 27.

[264] Azure availability set vs availability zone: 2021. https://www.pragimtech.com/blog/azure/availability-set-vs-availability-zone/. Accessed: 2021- 08- 29.

[265] Anita Choudhary, Mahesh Chandra Govil, Girdhari Singh, Lalit K. Awasthi, Emmanuel S. Pilli, and Divya Kapil. 2017. A critical survey of live virtual machine migration techniques. J. Cloud Comput.6, 1, Article 92 (December 2017), 41 pages.

[266] Dinesh, S., 2018. Kubernetes requests vs limits: Why adding them to your Pods and Namespaces matters | Google Cloud Blog. [online] Google Cloud Blog. Available at: <https://cloud.google.com/blog/products/containers-kubernetes/kubernetes-best-practices-resource-requests-and-limits> [Accessed 25 October 2021].

[267] Qualcomm. 2015. Qualcomm Technologies announces new specs for ARM-based data-center SoC | Qualcomm. Available at: <https://www.qualcomm.com/news/onq/2015/10/09/qualcomm-announces-new-specs-arm-based-data-center-soc> [Accessed 26 October 2021].

[268] Bhagyalakshmi and D. Malhotra, "A Critical Survey of Virtual Machine Migration Techniques in Cloud Computing," 2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC), 2018, pp. 328-332

[269] Patel A, Daftedar M, Shalan M, Watheq El-Kharashi M. Embedded Hypervisor Xvisor: A comparative analysis.

Appendix A

Source code configuration for CPU live migration performance

Each of the following sub-appendices contains blocks of source code as take place in Chapter 7 for the develop and implementation of a CPU live migration process.

A-1: Send CPU data source code function

A-2: Prepare network socket to send CPU data

A-3: Socket configuration

A-4: Reception script at destination host

A-5: Receive CPU data source code function

A-6: Adopt CPU data source code function

A-1: Send CPU data source code function

```
static int cmd_ping_senddata(struct vmm_chardev *cdev, int argc, char
**argv)
{
      //Timer initiation
      u64 timer_stamp = 0;
      u64 mult, start_tstamp, end_tstamp;
      start_tstamp = vmm_timer_timestamp();
      //Variables definition
      u8 ipaddr[4];
      int id1, id2;
      struct vmm_vcpu *vcpu;
      struct vmm_vcpu *vcpu1;
      struct vmm_vcpu *vcpu2;

      const char *str;
      irq_flags_t flags;

      //Input arguments validation
      if((argc < 1) || (argc > 3)) {
            cmd_ping_usage(cdev);
            return VMM_EFAIL;
      }
      //Conversions to the right format
      str2ipaddr(ipaddr, argv[0]);
      vmm_cprintf(cdev, "[+] ---Connected to: (%s) ---[+]\n", argv[0]);

      id1 = atoi(argv[1]);
      id2 = atoi(argv[2]);
      //Given vCPU IDs validation
      vcpu1 = vmm_manager_vcpu(id1);
      if (!vcpu1) {
            vmm_cprintf(cdev, "Failed to find vcpu\n");
            return VMM_EFAIL;
      }
      vcpu2 = vmm_manager_vcpu(id2);
      if (!vcpu2) {
            vmm_cprintf(cdev, "Failed to find vcpu\n");
            return VMM_EFAIL;
      }
      //Trigger of send data functionality
      netstack_prefetch_arp_mapping(ipaddr);
      netstack_send_udpdata(ipaddr, arm_regs(vcpu1), arm_regs(vcpu2));
      //End of timer
      end_tstamp = vmm_timer_timestamp();
      timer_stamp = (end_tstamp - start_tstamp);
      vmm_cprintf(cdev, "timer_stamp = %"RPId64" nanoseconds\n",
timer_stamp);
      return VMM_OK;
}
```

A-2: Prepare network socket to send CPU data

```
int netstack_send_udpdata(u8 *ripaddr, arch_regs_t *regs1, arch_regs_t
*regs2)
{
      //Variables declaration
      int s, i;
      unsigned data_buffer[50];
      struct sockaddr_in sa,ra;
      struct ip_hdr *iphdr;
      struct udp_hdr *udphdr;
      ip_addr_t to_addr, from_addr;
      size_t len = sizeof(struct udp_hdr) + data_buffer;
      /*Print size of the data structure */
```

```
        vmm_printf("***--%u-***\n",sizeof(data_buffer));
        /*Print Data - Registers */
        vmm_printf("[+] ---Core Registers VCPU1--- [+]\n");
        vmm_printf(" %11s=0x%016lx %11s=0x%016lx\n",
                  "SP", regs1->sp,
                  "LR", regs1->lr);
        vmm_printf(" %11s=0x%016lx %11s=0x%08lx\n",
                  "PC", regs1->pc,
                  "PSTATE", (regs1->pstate & 0xffffffff));

        vmm_printf("[+] ---Core Registers VCPU2--- [+]\n");
        vmm_printf(" %11s=0x%016lx %11s=0x%016lx\n",
                  "SP", regs2->sp,
                  "LR", regs2->lr);
        vmm_printf(" %11s=0x%016lx %11s=0x%08lx\n",
                  "PC", regs2->pc,
                  "PSTATE", (regs2->pstate & 0xffffffff));
        /*Load the values of register inside the array*/
        data_buffer[0] = regs1->sp;
        data_buffer[1] = regs1->lr;
        data_buffer[2] = regs1->pc;
        data_buffer[3] = (regs1->pstate & 0xffffffff);
        data_buffer[4] = regs2->sp;
        data_buffer[5] = regs2->lr;
        data_buffer[6] = regs2->pc;
        data_buffer[7] = (regs2->pstate & 0xffffffff);

        /* Prepare target address */
        IP4_ADDR(&to_addr, ripaddr[0],ripaddr[1],ripaddr[2],ripaddr[3]);

        /* Open RAW socket */
        if ((s = lwip_socket(AF_INET, SOCK_DGRAM, IP_PROTO_UDP)) < 0) {
              vmm_printf("%s: failed to open UDP socket\n", __func__);
```

```
            return VMM_EFAIL;
      }else
      {
            vmm_printf("[+] ---UDP Socket Created!--- [+]\n");
      }

      /* Prepare Sender socket address */
      memset(&sa, 0, sizeof(struct sockaddr_in));
      sa.sin_len = sizeof(sa);
      sa.sin_family = AF_INET;
      sa.sin_port = htons(SENDER_PORT);
      sa.sin_addr.s_addr = inet_addr(SENDER_IP);

      /*Bind the socket locally*/
      if (lwip_bind(s, (struct sockaddr *)&sa, sizeof(struct sockaddr_in) )
== -1)
      {
            vmm_printf("***--Bind to Port num %d failed*** \n",
SENDER_PORT);
            /* Close RAW socket */
            lwip_close(s);
      }else
      {
            vmm_printf("[+] ---UDP Socket Binded!--- [+]\n");
      }
      /*Prepare Receiver socket address */
      memset(&ra, 0, sizeof(struct sockaddr_in));
      ra.sin_len = sizeof(ra);
      ra.sin_family = AF_INET;
      ra.sin_port = htons(RECEIVER_PORT);
      ra.sin_addr.s_addr = inet_addr(RECEIVER_IP);

      int sent_data = lwip_sendto(s, data_buffer, sizeof(data_buffer), 0,
(struct sockaddr *)&ra, sizeof(ra));


      //Print Debugging messages
      if (sent_data < 0)
      {
            vmm_printf("***sent failed***\n");
            netstack_socket_free(s);
            lwip_close(s);
            return VMM_EFAIL;
      }else
      {
            vmm_printf("[+] ---UDP Data Sented!--- [+]\n");
      }


      netstack_socket_free(s);
      lwip_close(s);


}
VMM_EXPORT_SYMBOL(netstack_send_udpdata);
```

A-3: Socket configuration

```
      /* Open RAW socket */
      if ((s = lwip_socket(AF_INET, SOCK_DGRAM, IP_PROTO_UDP)) < 0) {
            vmm_printf("%s: failed to open UDP socket\n", __func__);
            return VMM_EFAIL;
      }else
      {
            vmm_printf("[+] ---UDP Socket Created!--- [+]\n");
      }
      /* Prepare Sender socket address */
```

```
	memset(&sa, 0, sizeof(struct sockaddr_in));
	sa.sin_len = sizeof(sa);
	sa.sin_family = AF_INET;
	sa.sin_port = htons(SENDER_PORT);
	sa.sin_addr.s_addr = inet_addr(SENDER_IP);

	/*Bind the socket locally*/
	if (lwip_bind(s, (struct sockaddr *)&sa, sizeof(struct sockaddr_in) )
== -1)
	{
		vmm_printf("***--Bind to Port num %d failed*** \n",
SENDER_PORT);
		/* Close RAW socket */
		lwip_close(s);
	}else
	{
		vmm_printf("[+] ---UDP Socket Binded!--- [+]\n");
	}

	/*Prepare Receiver socket address */
	memset(&ra, 0, sizeof(struct sockaddr_in));
	ra.sin_len = sizeof(ra);
	ra.sin_family = AF_INET;
	ra.sin_port = htons(RECEIVER_PORT);
	ra.sin_addr.s_addr = inet_addr(RECEIVER_IP);

	int sent_data = lwip_sendto(s, data_buffer, sizeof(data_buffer), 0,
(struct sockaddr *)&ra, sizeof(ra));
```

A-4: Reception script at destination host

```
#define BUFLEN 200
#define PORTSERVER 6002
#define PORTCLIENT 6002

int main(void)
{
    // Variables and structure definitions
    struct sockaddr_in serveraddr, clientaddr;
    int s, slen = sizeof(clientaddr), recv_len;
    unsigned buf[BUFLEN];
    //Creation of a Unix Socket
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
    perror("Cannot create socket");
    return 0;
    }
    //Reserve memory equal to an IP address size
    memset(&serveraddr, 0, sizeof(serveraddr));
    memset(&clientaddr, 0, sizeof(clientaddr));
    //Assign an IP address to the socket
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(PORTSERVER);
    clientaddr.sin_family = AF_INET;
    clientaddr.sin_addr.s_addr = inet_addr("192.168.0.10");
    clientaddr.sin_port = htons(PORTCLIENT);

    //Bind the socket
    if ( bind(s, (const struct sockaddr *)&serveraddr, sizeof(serveraddr))
< 0 )
    {
```

```
    perror("Bind Failed");
    return 0;
    }
    //Listening for incoming data
    while(1)
    {
        printf("[+] ---Waiting for data....[+]\n");
        if ((recv_len = recvfrom(s, buf, BUFLEN, 0, (struct sockaddr
*)&clientaddr, &slen)) < 0)
        {
        perror("recvfrom failed");
        }
        //Print on standard output the received data
        printf("[+] ---received :%d bytes ---[+]\n", recv_len);
            if (recv_len > 0 )
            {
            printf("[+] ---VCPU1 registers ---[+]\n ");
            printf("[+] ---received: SP = %016lx ---[+]\n",  buf[0]);
            printf("[+] ---received: LR = %016lx ---[+]\n",  buf[1]);
            printf("[+] ---received: PC = %016lx ---[+]\n",  buf[2]);
            printf("[+] ---received: PSTATE = %016lx ---[+]\n",  buf[3]);

            printf("[+] ---VCPU2 registers ---[+]\n");
            printf("[+] ---received: SP = %016lx ---[+]\n",  buf[4]);
            printf("[+] ---received: LR = %016lx ---[+]\n",  buf[5]);
            printf("[+] ---received: PC = %016lx ---[+]\n",  buf[6]);
            printf("[+] ---received: PSTATE = %016lx ---[+]\n",  buf[7]);


            }
    close(s);
    return(0);
    }
}
```

A-5: Receive CPU data source code function

```
int netstack_receive_udpdata(u8 port){

      unsigned data_buffer[5];
      struct sockaddr_in sa,ra;
      int s, err, i, slen = sizeof(sa);
      int recv_len;
      struct ip_hdr *iphdr;
      struct udp_hdr *udphdr;
      ip_addr_t to_addr, from_addr;
      size_t off, fromlen = sizeof(sa);

      vmm_printf("***--%s-***\n",__func__);
      /* Open RAW socket */
      if ((s = lwip_socket(AF_INET, SOCK_DGRAM, IP_PROTO_UDP)) < 0) {
            vmm_printf("%s: failed to open UDP socket\n", __func__);
            return VMM_EFAIL;
      }
      /* Set socket option */
      i = PING_RCV_TIMEO;
      lwip_setsockopt(s, SOL_SOCKET, SO_RCVTIMEO, &i, sizeof(i));
      /* Prepare Receiver socket address */
      memset(&ra, 0, sizeof(ra));
      ra.sin_len = sizeof(ra);
      ra.sin_family = AF_INET;
      ra.sin_port = htons(port);
      ra.sin_addr.s_addr = inet_addr(RECEIVER_IP);

      /*Bind the socket locally*/
      if (lwip_bind(s, (struct sockaddr *)&ra, sizeof(struct sockaddr_in))
< 0)
      {
            vmm_printf("***--Bind to Port num %d failed*** \n",
RECEIVER_PORT);
            /* Close RAW socket */
            lwip_close(s);
      }
      memset(&sa, 0, sizeof(sa));
      sa.sin_len = sizeof(sa);
      sa.sin_family = AF_INET;
      sa.sin_port = htons(SENDER_PORT);
      sa.sin_addr.s_addr = inet_addr(SENDER_IP2);

      /* Wait for reply */
      vmm_printf("***Waiting for data....***\n");
      off = 0;
      while (1)
      {

      if ((off = lwip_recvfrom(s, data_buffer, sizeof(data_buffer), 0,
                  (struct sockaddr*)&sa, &fromlen)) < 0)
                  {
                        vmm_printf("Nothing to received....\n");
                  }else
                  {
                  vmm_printf("***--%d--...***\n", off);
                  vmm_printf("***received: %02x--\n", data_buffer);

                  err = VMM_OK;}
      }
      vmm_printf("***Waiting for data ....1***\n");

            lwip_close(s);
            netstack_socket_free(s);
```

```
        lwip_close(s);
        return err;
}
VMM_EXPORT_SYMBOL(netstack_receive_udpdata);
```

A-6: Adopt CPU data source code function

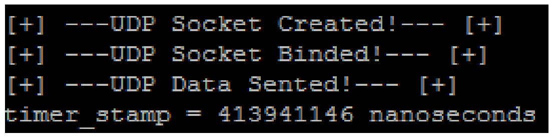
```
int netstack_adapt_registers(arch_regs_t *regs1, arch_regs_t *regs2) {
        unsigned data_buffer[50];

        vmm_printf("[+] ---Core Registers VCPU1--- [+]\n");
        vmm_printf(" %11s=0x%016lx %11s=0x%016lx\n",
                   "SP", regs1->sp,
                   "LR", regs1->lr);
        vmm_printf(" %11s=0x%016lx %11s=0x%08lx\n",
                   "PC", regs1->pc,
                   "PSTATE", (regs1->pstate & 0xffffffff));

        vmm_printf("[+] ---Core Registers VCPU2--- [+]\n");
        vmm_printf(" %11s=0x%016lx %11s=0x%016lx\n",
                   "SP", regs2->sp,
                   "LR", regs2->lr);
        vmm_printf(" %11s=0x%016lx %11s=0x%08lx\n",
                   "PC", regs2->pc,
                   "PSTATE", (regs2->pstate & 0xffffffff));

        /*Store the values of registers inside an array*/
        data_buffer[0] = regs1->sp;
        data_buffer[1] = regs1->lr;
        data_buffer[2] = regs1->pc;
        data_buffer[3] = (regs1->pstate & 0xffffffff);
```

```
      /*Load the values from the array into the registers*/
      regs2->sp = data_buffer[0];
      regs2->lr = data_buffer[1];
      regs2->pc = data_buffer[2];
      regs2->pstate = data_buffer[3];

      vmm_printf("[+] ---Core Registers VCPU1--- [+]\n");
      vmm_printf(" %11s=0x%016lx %11s=0x%016lx\n",
                "SP", regs1->sp,
                "LR", regs1->lr);
      vmm_printf(" %11s=0x%016lx %11s=0x%08lx\n",
                "PC", regs1->pc,
                "PSTATE", (regs1->pstate & 0xffffffff));

      vmm_printf("[+] ---Core Registers VCPU2--- [+]\n");
      vmm_printf(" %11s=0x%016lx %11s=0x%016lx\n",
                "SP", regs2->sp,
                "LR", regs2->lr);
      vmm_printf(" %11s=0x%016lx %11s=0x%08lx\n",
                "PC", regs2->pc,
                "PSTATE", (regs2->pstate & 0xffffffff));

      return VMM_OK;
}
VMM_EXPORT_SYMBOL(netstack_adapt_registers);
```

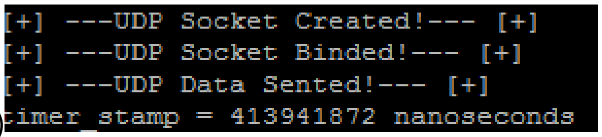
Appendix B

Evaluation results of a CPU live migration performance

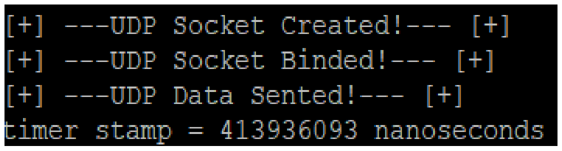
Time measurements of CPU live migration performance of 34 CPU registers

a) 

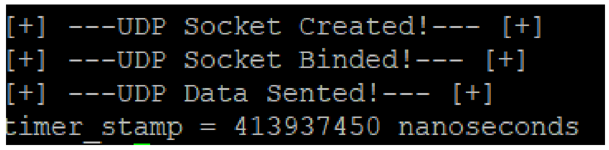
```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413941146 nanoseconds
```

b) 

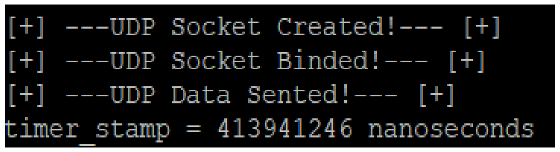
```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413941872 nanoseconds
```

c) 

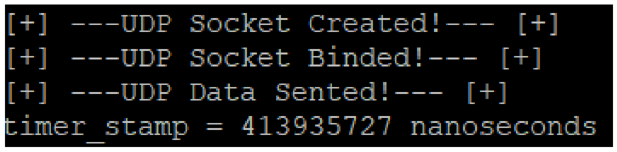
```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413936093 nanoseconds
```

d) 

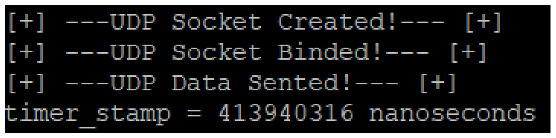
```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413937450 nanoseconds
```

e) 

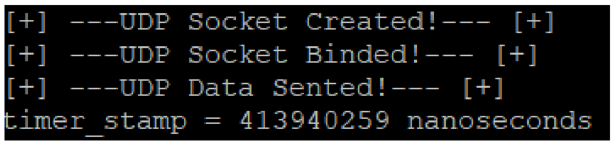
```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413941246 nanoseconds
```

f) 

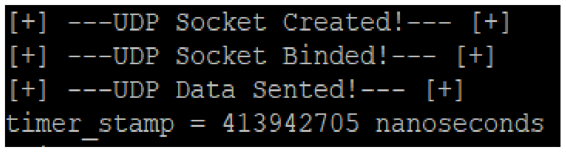
```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413935727 nanoseconds
```

g) 

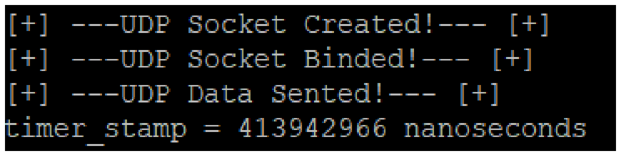
```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413940316 nanoseconds
```

h) 

```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413940259 nanoseconds
```

i) 

```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413942705 nanoseconds
```

j) 

```
[+] ---UDP Socket Created!--- [+]
[+] ---UDP Socket Binded!--- [+]
[+] ---UDP Data Sented!--- [+]
timer_stamp = 413942966 nanoseconds
```

Appendix C

Structure definition of ARM registers on XEN hypervisor system.

```
struct cpu_user_regs
{
  /*
   * The mapping AArch64 <-> AArch32
   *
   *      AArch64      AArch32
   */
```

```
__DECL_REG(x0,         r0/*_usr*/);
__DECL_REG(x1,         r1/*_usr*/);
__DECL_REG(x2,         r2/*_usr*/);
__DECL_REG(x3,         r3/*_usr*/);
__DECL_REG(x4,         r4/*_usr*/);
__DECL_REG(x5,         r5/*_usr*/);
__DECL_REG(x6,         r6/*_usr*/);
__DECL_REG(x7,         r7/*_usr*/);
__DECL_REG(x8,         r8/*_usr*/);
__DECL_REG(x9,         r9/*_usr*/);
__DECL_REG(x10,        r10/*_usr*/);
__DECL_REG(x11 ,       r11/*_usr*/);
__DECL_REG(x12,        r12/*_usr*/)
__DECL_REG(x13,        /* r13_usr */ sp_usr);
__DECL_REG(x14,        /* r14_usr */ lr_usr);
__DECL_REG(x15,        /* r13_hyp */ __unused_sp_hyp);
__DECL_REG(x16,        /* r14_irq */ lr_irq);
__DECL_REG(x17,        /* r13_irq */ sp_irq)
__DECL_REG(x18,        /* r14_svc */ lr_svc);
__DECL_REG(x19,        /* r13_svc */ sp_svc);
__DECL_REG(x20,        /* r14_abt */ lr_abt);
__DECL_REG(x21,        /* r13_abt */ sp_abt);

__DECL_REG(x22,        /* r14_und */ lr_und);
__DECL_REG(x23,        /* r13_und */ sp_und);
__DECL_REG(x24,        r8_fiq);
__DECL_REG(x25,        r9_fiq);
__DECL_REG(x26,        r10_fiq);
__DECL_REG(x27,        r11_fiq);
__DECL_REG(x28,        r12_fiq);
__DECL_REG(/* x29 */ fp, /* r13_fiq */ sp_fiq);
__DECL_REG(/* x30 */ lr, /* r14_fiq */ lr_fiq);
```

```
register_t sp; /* Valid for hypervisor frames */


/* Return address and mode */
__DECL_REG(pc,          pc32);          /* ELR_EL2 */
uint64_t cpsr;                          /* SPSR_EL2 */
uint64_t hsr;                           /* ESR_EL2 */
```