FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Study and Implementation of Modular Software Architectures based on Hypervisors for Automotive Electronic Control Units

Pedro Miguel Veiga Almeida e Silva

Mestrado em Engenharia Electrotécnia e de Computadores

Orientador: Prof. João Canas Ferreira

October 20, 2023

Abstract

In the ever-evolving world of automotive technology, the presence of electronics in a modern vehicle keeps rising with every new feature the automotive industry needs to develop and implement. As a result of this constant innovation, Electronic Control Units (ECUs) are becoming increasingly complex and expensive to produce. The difficulty resides in integrating hardware and software to create ECUs that are dependable, efficient and capable of managing many interconnected systems. Any malfunction in these complex devices can cause significant financial loss and, more importantly, imperil human lives. As the automotive industry evolves, ECUs must remain sophisticated enough to accommodate future features and functionalities emerging in the coming years. This forward-thinking strategy ensures that vehicles stay at the vanguard of technological advancements while maintaining the highest levels of safety. Extensive research, development, and testing are necessary to maintain the utmost performance, efficiency, and security levels to meet these stringent modern automotive ECU software requirements.

The primary objective of this thesis is to investigate the feasibility and practicality of implementing an open-source hypervisor-based solution for future Electronic Control Units in the automotive industry. With the constant evolution of automotive technology, the presence of electronics in vehicles is continuously increasing, necessitating more sophisticated and interconnected systems. As ECUs become increasingly complex and expensive to manufacture, exploring open-source hypervisor technology presents a promising avenue for addressing these challenges. By adopting a hypervisor-based approach, this research aims to develop a transversal solution that can efficiently and reliably manage multiple virtualized systems within a single ECU based on embedded systems. The study will also examine the potential benefits of selecting hypervisor features that hold significant potential for enhancing automotive Electronic Control Units. These features, when applied appropriately, can bring about various benefits to the automotive industry, including improved flexibility, scalability, and cost-effectiveness. Additionally, this dissertation will assess the implications of implementing this open-source approach on safety, security, and overall system performance. Ultimately, this research endeavors to contribute valuable insights to the automotive industry, potentially paving the way for the adoption of open-source hypervisor-based solutions in future ECUs.

Keywords: Hypervisors, Electronic Control Units, Virtualization, Embedded systems

Resumo

No mundo em constante evolução da tecnologia automóvel, a presença da eletrónica num veículo moderno continua a aumentar com cada nova funcionalidade que a indústria automóvel precisa de desenvolver e implementar. Como resultado desta inovação constante, as unidades de controlo eletrónico (ECU) estão a tornar-se cada vez mais complexas e dispendiosas de produzir. A dificuldade reside na integração de hardware e software para produzir ECUs que sejam não só fiáveis e eficientes, mas também capazes de gerir uma multiplicidade de sistemas interligados. Qualquer mau funcionamento destes dispositivos complexos pode causar perdas financeiras significativas e, mais importante ainda, pôr em perigo vidas humanas. As UEC devem manter-se suficientemente sofisticadas para acomodar futuras características e funcionalidades que possam surgir nos próximos anos, à medida que a indústria automóvel evolui. Esta estratégia de visão de futuro garante que os veículos permaneçam na vanguarda dos avanços tecnológicos, mantendo os mais elevados níveis de segurança. São necessários investigação, desenvolvimento e ensaios exaustivos para manter os níveis máximos de desempenho, eficiência e segurança, a fim de satisfazer os rigorosos requisitos modernos de software das UEC para automóveis.

O principal objetivo desta tese é investigar a viabilidade e a praticidade da implementação de uma solução baseada em hipervisor de código aberto para futuras unidades de controlo eletrónico na indústria automóvel. Com a constante evolução da tecnologia automóvel, a presença de eletrónica nos veículos está em constante aumento, exigindo sistemas mais sofisticados e interligados. Á medida que as ECUs se tornam cada vez mais complexas e dispendiosas de fabricar, a exploração da tecnologia de hipervisor apresenta uma via promissora para enfrentar estes desafios. Ao adotar uma abordagem baseada em hipervisor, esta investigação visa desenvolver uma solução transversal que possa gerir de forma eficiente e fiável vários sistemas virtualizados numa única ECU baseada em sistemas embarcados. O estudo examinará também os benefícios potenciais de uma seleção de características do hipervisor que têm um potencial significativo para melhorar as unidades de controlo eletrónico dos automóveis. Estas características, quando aplicadas adequadamente, podem trazer vários benefícios para a indústria automóvel, incluindo maior flexibilidade, escalabilidade e relação custo-eficácia. Além disso, esta dissertação avaliará as implicações da aplicação desta abordagem de código aberto na segurança e no desempenho global do sistema. Em última análise, esta investigação procura contribuir com conhecimentos valiosos para a indústria automóvel, abrindo potencialmente o caminho para a adoção de soluções baseadas em hipervisor de código aberto em futuras ECU.

Palavras-chave: Hypervisor, ECU, Virtualização, Sistemas embarcados

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador, o Professor João Canas Ferreira, pelas orientações e disponibilidade durante a elaboração desta dissertação e à Critical Techworks pela oportunidade e forma calorosa como me acolheram. Um agradecimento também ao Engenheiro João Monteiro por ter iniciado este processo comigo e ao Engenheiros Nikhil Santosh e João Garrido por terem continuado o seu trabalho.

Em segundo lugar, um grande agradecimento aos meus pais que me ajudaram em toda esta jornada a qual sem eles não era possível. Aos meus avós, um agradecimento do fundo do coração por sempre me apoiarem. À minha namorada e aos meus amigos, obrigado por todo o apoio neste percurso e pela ajuda não só durante a dissertação mas também durante os últimos 5 anos.

Pedro Almeida e Silva

"Learning is the only thing the mind never exhausts, never fears, and never regrets."

Leonardo da Vinci

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

ADAS	Advanced driver-assistance system
AGL	Automotive Grade Linux
API	Aplication Programming Interface
APU	Accelerated Processing Unit
ASIL	Automotive Safety Integrity Level
AWS	Amazon Web Services
BSP	Board Support Package
CAN	Controller Area Network
CPU	Central Processing Unit
CTW	Critical Techworks
ECU	Eletronic Control Unit
EG-VIRT	Automotive Grade Linux Virtualization Expert Group
EL	Exception Levels
FPGA	Field Programmable Gate Arrays
GPOS	General Purpose Operating System
HMI	Human Machine Interface
HW	Hardware
I/O	Inputs/Outputs
ID	Identification
IVI	In-Vehicle Infotainment
KVM	Kernel-based Virtual Machine
LXC	Linux Containers
OpenAMP	Open Asymmetric Multi Processing
OS	Operating System
QEMU	Quick Emulator
RAM	Random Access Memory
RPU	Real-time Processing Unit
Remoteproc	Remote Processor Framework
RPMsg	Remote Processor Messaging
RTOS	Real Time Operating System
RTS	Real Time System
SOC	System-on-a-Chip
SDK	Software Development Kit
SW	Software
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VCU	Vehicle Control Unit
VM	Virtual Machine
VMM	Virtual Machine Monitor

Chapter 1

Introduction

1.1 Context

In a modern vehicle, every component is designed to achieve the best performance and reliability while keeping production, development and maintenance costs in mind. The Electronic Computer Unit (ECU) is a component that, with the latest technology advancements, has taken a crucial role in the automotive industry. These components are responsible for monitoring and controlling nearly every function in a vehicle, from the In-Vehicle Infotainment (IVI) to the most straightforward system as a window controller. As a result, these components have proprietary software that is focused on optimizing the hardware performance and providing fail safes to face any error that occurs while in operation.

ECU software (ECU SW) typically consists of several subsystems with varying degrees of complexity and criticality. Most of these subsystems are shared despite the functional differences between each ECU, which naturally emphasizes the ideas of modularity, reusability, and isolation. These characteristics result in a SW that can be transversely applied to every modern vehicle system. ECU SW also has the task of handling the virtualization of the available hardware, a technique that consists of virtually separating the physical resources of a system between multiple users and has been widely used since the 1960s from gaming console software to data centers. As such, virtualization is arguably one of the best and most used options to implement these ECUs, so much so that most manufacturers are currently deploying it in the automotive industry [1]. Currently, BMW utilizes virtualization in their ECUs in the form of containers, which will be explained later in this document. As the demand for higher safety, isolation, performance and modularity requirements keeps rising, this thesis will explore a solution based in hypervisors for modern modular SW architectures that can accomplish those goals.

That being said, it is not that simple to implement a hypervisor solution in this context because, on the one hand, many hypervisor solutions are available, all with their own proprietary features. On the other hand, there may be constraints with hardware and software compatibility.

1.2 Motivation

This dissertation was proposed by Critical TechWorks, a joint venture created by BMW and Critical Software, aiming to explore innovative SW architectures for upcoming ECUs. With the evolution of autonomous driving and comfort features for the user, BMW realized that their current container-based virtualization utilized in their ECUs would not respond to the demands of security and re-utilization. As such, BMW opted to research a new SW architecture using hypervisors as a foundational technology for modern modular SW designs for the next ECU generations. This is the main focus of this dissertation, investigating the possible advantages and drawbacks of the hypervisor implementation, researching a solution based on open-source software and trying to implement a basic system in order to show the potential of hypervisor virtualization.

1.3 Objective

This dissertation aims to analyze current virtualization techniques used in the automotive industry and assess a possible solution that can implement a system using hypervisor-based technology, serving as a proof of concept to the next generations of ECU SW. Also, the feasibility of using hypervisors in situations of reusability and modularity will be discussed. In the end, a possible hypervisor solution will be proposed in order to showcase hypervisor features that may be useful in an automotive ECU scenario.

Additionally, it will explore the practicality of using hypervisors to enhance reusability and modularity. The study will culminate in proposing a concrete hypervisor solution, exemplifying its advantageous features within an automotive ECU context. Through this comprehensive investigation, the research endeavors to contribute valuable insights into the optimal virtualization approach for future automotive systems.

1.4 Dissertation structure

This dissertation is composed of an additional six chapters. In Chapter 2, there is a brief explanation of the theoretical concepts in this engineering field. Chapter 3 displays some projects investigating similar topics to this dissertation. In Chapters 4 and 5, the proposed solution for this dissertation gets presented (as well as the various demonstrations) and then implemented. Finally, in Chapters 6 and 7, the results from the demonstrations and a summary of the work executed are presented. The results and features of the evaluated hypervisor suggest that they can be an excellent alternative for implementation in automotive ECUs. Their robust performance, reliable virtualization capabilities, efficient resource utilization, and inherent security measures make them well-suited for automotive systems. Integrating hypervisors can lead to improved flexibility, scalability, and the ability to run multiple applications simultaneously, enhancing automotive computing and safety-critical systems.

Chapter 2

Background

This section presents the essential theoretical foundations required to understand the various topics discussed throughout this dissertation.

2.1 Virtualization

Virtualization is a technique that enables the creation of meaningful information technology services employing traditionally hardware-bound resources. It allows the developer to utilize the full potential of a physical computer by distributing its capabilities among several users or settings [2].

Two forms of operating system virtualization exist: container-based virtualization and hypervisors, from now on referenced as containers and hypervisors. These topics will be explained further in this document.

Virtualization offers several advantages, like:

- **Hardware consolidation:** In many practical cases, applications cannot be run in the same environment or on the same operating system. Without virtualization, the only solution would be to use separate machines to deploy each application. With the use of virtualization, there is a clear advantage in saving on hardware and energy costs and decreasing the need for physical space [3]. These advantages can benefit the automotive industry by reducing the wiring complexity and the number of ECUs scattered around the car, concentrating them into a single ECU (the optimal solution) [2].

- **Application Isolation:** Like the previous point, if strict isolation is needed between two applications, it cannot be achieved with only one machine. However, using virtualization, the emulation of two different environments can be possible, achieving the desired isolation. This is very important in the security of every application, guaranteeing that if a single application malfunctions or gets compromised in the case of a malicious attack, the whole system is not affected [2].

- **Mixed criticality:** This topic is a vital subject in many industry applications and will be addressed later in this document. Mixed criticality is the ability of a system to manage various workloads with different response times for hard real-time and soft real-time applications (this terms will be explained further on) [2].

In order to understand the next virtualization topic, it is important to introduce the topic of protection rings [3].

These protection rings are the procedures that safeguard data or errors depending on the security imposed while accessing computer system resources [3].

Protection rings have been attributed numbers from the most privileged to the least privileged. Starting at 0, this protection ring is the most privileged, interacting directly with the machines hardware. This ring also houses the system's kernel. By consequence, ring 3, usually called user mode, is the least privileged one and is where most applications are run [3].

In the case of virtualization, there is a hypervisor/virtual machine monitor (VMM) running in ring 0 since it needs access to the hardware. This causes a problem regarding the guest operating systems kernel because it also needs to reside in ring 0 and is unaware that the VMM exists. But with only one kernel per ring, an alternative was created [3].

There are some methods of virtualization to solve this problem based on the way the software communicates with the system hardware:

- **Full virtualization:** As seen in figure 2.1, full virtualization relies on techniques such as binary translation of instructions and input/outputs (I/O) requests to emulate the communication between the software and the hardware. This way, the applications or operating systems running are not aware of the virtualization layer and, therefore do not need to be modified [3].

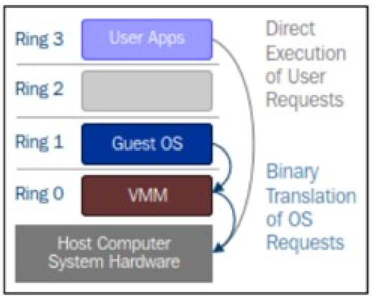

Ring 3
User Apps
Ring 2
Ring 1
Guest OS
Ring 0
VMM
Host Computer System Hardware
Direct Execution of User Requests
Binary Translation of OS Requests

Figure 2.1: Full virtualization architecture(source: [3])

- **Para-virtualization:** In para-virtualizatíon, the guest OS needs to be modified to communicate with the VMM through hypercalls. As explained in figure 2.2, the guest OS is aware that it is being virtualized and uses an Application Programming Interface (API) provided by the VMM to exchange privileged instruction calls [3].

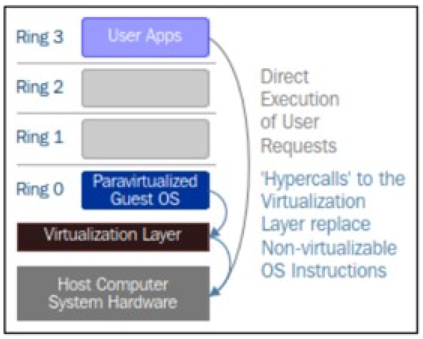

Ring 3
User Apps
Ring 2
Ring 1
Ring 0
Paravirtualized Guest OS
Virtualization Layer
Host Computer System Hardware
Direct Execution of User Requests
'Hypercalls' to the Virtualization Layer replace Non-virtualizable OS Instructions

Figure 2.2: Para-virtualization architecture (source: [3])

2.2 Containers

Container-based virtualization is a method for packaging a program and its dependencies and libraries into a single entity that can run anywhere the container engine is. In other words, a container engine abstracts the operating system so that the containers perceive that the whole OS is available to them. Therefore, one of the disadvantages is that the software running on the container needs to be compatible with the OS that it is running on. On the other hand, it doesn't have to boot a whole new OS when launching said container, which enables it to start in a few milliseconds. Another advantage of containers is their distribution, since it only needs the container image to run the processes, all a developer needs to do is build the image and send it to the desired machine running the same OS [4].

From now on, the method in which container function in a Linux machine works will be explained. These methods are relevant to this dissertation because of the current BMW ECU software that utilizes Linux as an OS.

Containers provide an isolated user space while sharing a common kernel. It uses two Linux tools called control groups (Cgroups) and kernel namespaces to allocate resources and isolation between containers [4].

Cgroups allow the user to allocate the desired resources to a group of processes and, comparable to Linux processes, a child Cgroup inherits his parent attributes. In this way, a user can limit CPU cores or memory amount a group of processes can use [4].

Kernel namespaces is a feature that allows a container to be isolated from other containers. The kernel passes global resources and group/user/process IDs into a namespace, allowing its separation. It can be possible to create namespaces with the identical process IDs as the host OS or other containers since they are only exclusive in the same namespace. In other words, there cannot be two equal process IDs inside the same namespace because all process IDs are relative to the namespace. In this way, one can achieve the desired isolation [4].

There are two types of containers: OS containers and application containers. OS containers run whole systems. For example, Sun introduced Solaris containers primarily as a mechanism to virtualize additional Solaris systems on top of a Solaris host. On the contrary, application containers run a single application, making application containers a clear choice when there are reduced resources [5].

The following sub-sections present two container engines that BMW uses in its current ECU software.

2.2.1 LXC

Linux-based containers (LXC) is a toolkit used to create OS containers in Linux. This toolkit was designed essentially to fulfill the configuration and deployment phase of the container technologies. In other words, the user does not need to read every documentation about containers to deploy it. This significantly reduces the learning curve of setting up containers [4].

2.2.2 Docker

The Docker engine first appeared in 2013, building on top of the objective of LXC to simplify the configuration and deployment of containers. Contrary to LXC, Docker creates, manages and deploys application containers through a command line tool, taking the burden of knowing how a toolkit works from the user. It also improves by developing a repository called Docker Hub, in which every user can download a container image and contribute to its development [4].

2.3 Hypervisors

Hypervisor-based virtualization is a technology that implements a VMM/hypervisor layer between the hardware and the desired guest operating systems. This way, the hypervisor is responsible for various tasks such as hardware virtualization, VM life cycle management, migrating of VMs, allocating resources in real-time, defining policies for virtual machine management. This layer also needs to control all the memory translation and I/O mapping.

This technology allows the user to have different operating systems running in the same machine which is a great advantage when deploying various applications that cannot all run on the same OS. The VMM is also responsible for allocating memory, CPU cores and every resource to every guest so it's a critical part of the system [3].

There are two types of hypervisors:

- **Type 1:** In figure 2.3, the type 1 hypervisor works directly on top of the hardware (they can also be called Bare Metal or Native hypervisors) and does not need an OS in between to access the hardware. This way, they offer many advantages, such as easier installation, smaller sizes, and less overhead [3].

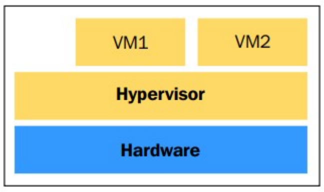


Figure 2.3: Type 1 hypervisor(source: [3])

- **Type 2:**In figure 2.4, the type 2 hypervisor needs to be installed on top of a host OS that allows numerous customizations and it is the host OS that is responsible for the hardware access. Some sources have stated that a type 2 hypervisor performs worse than a type 1 because of the performance overhead of having an extra OS layer [3].

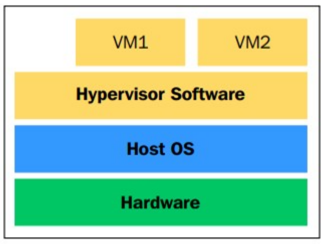


Figure 2.4: Type 2 hypervisor(source: [3])

There are numerous hypervisors on the market, ranging from open-source to closed-source or licensed hypervisors. The following subsection will explain the two open-source hypervisors chosen by Critical Techworks, SA (CTW).

2.3.1 KVM

KVM is an open-source hypervisor created in 2007 for the x86 architecture and later ported to ARM in 2012 [5]. As KVM was merged into Linux kernel many sources say that it is a Type 2 hypervisor [4] [2] while others say that in a scenario of stripping Linux of most of its components, only maintaining the bare essentials for the hypervisor, it can be considered Type 1 [6]. This fact is important because type 2 hypervisor usually have a more significant overhead than type 1. which can influence developers when choosing this hypervisor for their applications.

2.3.2 Xen

Xen is a Type 1 hypervisor created in 2003 as a research project by a team at the University of Cambridge [3]. Xen has the ability to support fully virtualized and para-virtualized guests in the x86 version, but on ARM only supports para-virtualization and ARM extensions [5].

2.4 Hypervisors and ARM architectures

As some BMW ECUs and, most importantly, the hardware on which the proposed solution will be implemented is ARM-based, it is essential to present a basic knowledge of how these hypervisors work with this architecture.

2.4.1 KVM on ARM processors

As KVM was first developed for x86, as previously mentioned in Section 2.3.1, and then ported to ARM, there were some compromises made in this adaptation [5]. The protection rings discussed in Section 2.1 are a feature of x86 architectures. ARM utilizes an identical feature named Exception levels (EL). Because of the architectural differences between these two platforms, KVM could not entirely reside in the kernel layer and therefore had to be split into two instances, Highvisor and Lowvisor. The Highvisor resides in the Linux Host Kernel and is responsible for most of the hypervisor functionalities. The Lowvisor resides on *ARMs Hyp mode* (a lower layer in relation to the host kernel layer as shown in figure 2.5) and handles isolation and hypervisor traps.

As previously referenced, this software works through the distinction of the hypervisor and the Virtual Machine, naming respectively Host and Guest. Host, as the name suggests, can be defined by the machine on which a virtual machine runs and guest is all the software related to the virtual machine.

This implementation has its advantage when it comes to portability because, provided that the ARM system is running a version of Linux higher than 3.9, the portability to other ARM machines does not require additional configurations. This is a good feature in the application case of the development and updates of ECUs [5].

On top of this, KVM requires userspace tools such as Quick Emulator (QEMU), Virtio and KVM tools residing in the host's user space to emulate guest hardware devices and instantiate virtual machines, taking advantage of the hardware virtualization extensions. This way, the system does not need to use a para-virtualization layer (Section 2.1) that would considerably increase its overhead [5].

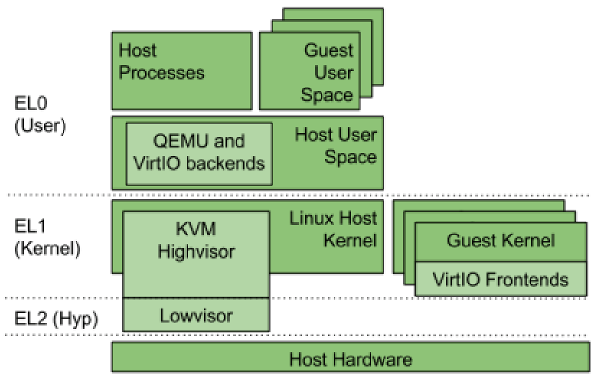


Figure 2.5: KVM in ARM [5]

2.4.2 XEN on ARM processors

Contrary to the KVM hypervisor, Xen does not use tools in the Host userspace layer, only residing in the hyp mode layer (the lowest software layer of the whole system) [5]. Xen's architecture is based on "domains", a system of partitions in which the guests get allocated, first creating a domain called Dom0 and the first guest gets allocated there. This domain is the more privileged one and its guest is responsible for managing other VMs placed in the lower privileged domain named *DomU*. This way, every communication between the VMs in DomU and the hardware needs to be handled by the Dom0, which can be a disadvantage. On the other hand, by residing in the hyp mode, Xen can utilize the virtualization extensions that ARM provides. So, Xen can perform better than KVM but, on the contrary, requires more configurations and fine-tuning in the migration to another ARM machine [5].

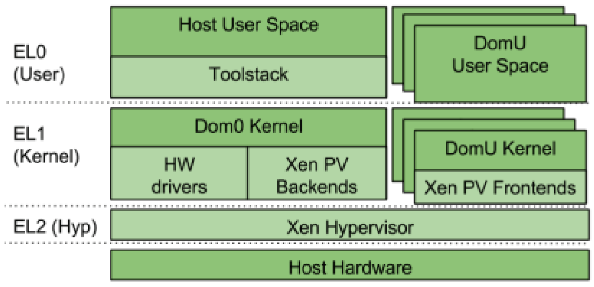


Figure 2.6: Xen in ARM [5]

2.5 Mixed Criticality

In industries with stringent safety requirements, such as aviation and automotive, mixed-criticality is an essential topic that needs to be addressed. Essentially, mixed-criticality can be defined as the seamless use/integration of applications/software with different levels of criticality in the same system. Criticality is evaluated based on the effect of function or component failure on the system's capacity to accomplish its tasks. As such, there must be a way to implement related requirements [7].

When discussing mixed-criticality, one of the most essential aspects for creating products that will be available to the public is its certification. There exists a wide range of certifications in the automotive industry [8] but, in the case of this dissertation, Automotive Safety Integrity Level (ASIL), defined by ISO26262, the standard that targets the safety of the electronic systems in a vehicle, is the certification that is most concerned to this topic [9]. ASIL has four levels regarding safety and how accident-prone a product is, with ASIL D being the level that designates the lowest failure rate (less than 1%). In the automotive case, the ASIL D level is usually required for the power train, telemetry, advanced driving assisted systems (ADAS), etc [9].

As BMW ECUs are essentially ARM-based, it is helpful to understand about a relevant ARM architecture feature regarding mixed criticality called ARM TrustZone.

2.5.1 ARM Trustzone

ARM TrustZone is a feature that provides two running modes: a Secure World that has higher privileges and is supposed to host the more safety-critical tasks in a Real-Time Operating System (RTOS) and a non-secure world that houses the general purpose operating system (GPOS). An example of a software that resides in the monitor mode of ARM TrustZone is *VOSYSMonitor* [10]. This low-level closed-source software acts as a lower-layer hypervisor that can run two hypervisors, one in the secure world and the other in the Normal world. It also has many advantages, such as its compliance with ASIL C standard. *VOSYSMonitor* achieves its mixed criticality goal through an interrupt managed by it, only letting the non-critical guest execute when the critical one is not in use [10].

Alternatively, there exists another software that utilizes the ARM trustzone feature. LTZVisor is a lightweight TrustZone-assisted hypervisor that allows the combination of two operating systems, one in the secure world (usually an RTOS) and another in the non-secure world (usually a general purpose operating system) [10]. One of the main differences between VOSYSMonitor and LTZVisor is that one is closed-source and open-source [10], respectively. This is an advantage to LTZVisor, considering the open-source preference in this dissertation.

One way of achieving mixed criticality maybe by implementing an RTOS. Therefore, it is appropriate to elaborate on the Real-Time Computing concept.

2.5.2 Real-Time Computing

A system can be classified by the time required to deliver the desired response. This way, Real-Time Computing is characterized by not only the logically correct response but also considering the moment it arrives, i.e., if it meets its deadline or not [11].

Real-time systems (RTS) can be defined by the impact of missing the deadlines:

- **Hard Real-Time systems:** These systems are characterized by the catastrophic consequence of missing their timing requirements. Such systems are more difficult to implement and certify. Example: Flight control systems, automotive telemetry systems, robotics etc [12].

- **Soft Real-Time systems:** In these systems, the response time is still important but not critical and missing its timing constraints can lead to performance degradation. Example: Banking system, multimedia, etc [12].

- **Firm Real-Time systems:** To these systems, the failure to comply to the deadlines is not critical, leading to the results being discarded [12].

Several RTOS exist, from which FREERTOS, SAFERRTOS and Zephyr RTOS are highlighted.

2.5.2.1 FREERTOS

This RTOS kernel was developed by Richard Barry around 2003 and then passed to the Amazon Web Services project (AWS) in 2017. It can support over 35 architectures, such as X86, MSP430, and most importantly, ARM. It has many features and advantages such as its low overhead, small size, use of C programming language, embedded scheduler and being open-source as the name suggests [13].

2.5.2.2 SAFERRTOS

SAFERRTOS is a project developed by the WITTENSTEIN group from FREERTOS, focusing on a hard real-time system. Therefore, it has ASIL D (the highest) certification by TÜV SÜD (German certification agency). Since its base is FREERTOS, it inherited all the functionalities and advantages of FREERTOS but, being a safety critical focused RTOS, it can be used for the most important aspects of the automotive industry such as throttle-by-wire, brake-by-wire etc [14].

2.5.2.3 Zephyr RTOS

Zephyr is an RTOS designed for embedded devices, from microcontrollers to servers. It is an open-source project supported by the Linux Foundation. Zephyr is lauded for its versatility in supporting multiple architectures and providing features such as preemptive multitasking, support for various communication protocols, and real-time scheduling. Zephyr is a real-time operating system that offers predictable and deterministic behavior, making it suitable for applications where timing is crucial. It is designed to be portable and supports a variety of processor architectures, such as ARM (Cortex-M, Cortex-R, and Cortex-A), ARC, MIPS, Nios II, RISC-V, Xtensa, SPARC, x86, and x86-64. This RTOS includes a comprehensive set of device drivers, protocol stacks, and libraries, which facilitate the development process for a variety of peripherals and functionalities[15].

As a general-purpose OS, Linux is the most widely available OS used in embedded systems and data centers. Yocto project is an open-source project that facilitates the creation of custom Linux distributions for embedded systems.

2.5.2.4 Yocto project

The Yocto project is a powerful and flexible open-source initiative that enables developers to create customized Linux distributions tailored for embedded systems. The Yocto project enables engineers to develop effective and streamlined Linux images tailored for specific hardware platforms or use cases by utilizing tools, recipes, and layers. The BitBake build tool, which orchestrates the intricate build process and analyzes metadata and recipes to produce the intended target picture, is at the heart of the project. This cross-compilation-capable project enables developers to work productively on their host PCs while targeting diverse architectures.

Custom Linux distribution development is further streamlined by the availability of numerous layers and detailed documentation, which decreases time-to-market and improves overall product quality. Its scalability and agility make it a top choice for various applications, from consumer electronics and automotive systems to industrial automation and IoT devices. The Yocto Project, which offers a strong foundation for creating compelling, dependable, and scalable Linux-based solutions, continues to be a crucial tool in the toolbox of embedded systems developers even as technology advances[16].

Chapter 3

State-of-the-art

This chapter describes and analyses use cases extracted from the existing literature applied to multiple contexts that employ technologies relevant to the current work. This research aimed to discover the technologies used in the proposed solutions and their performance analysis. With this knowledge, the proposed solution in this dissertation can be fully justified by the previous attempts described in this chapter.

3.1 Automotive Grade Linux

Nowadays, more and more industries are adopting virtualization as the primary way of software design for its versatility and resource abstraction. In the case of the automotive industry, Automotive Grade Linux (AGL) is an exciting project developed by The Linux Foundation that focuses on the applications of software standardization in this field.[2]

AGL was launched in 2012 by The Linux Foundation, with its original application in In-Vehicle Infotainment (IVI). However, it was later adapted to all sorts of other components like instrument clusters, telemetry, Human Machine Interfaces (HMI), etc. Since 2018, the Automotive Grade Linux Virtualization Expert Group (EG-VIRT) was created with the task of implementing virtualization into this project. [2]

AGL sets three main goals for its virtualization architecture solution: modularity, openness and mixed-criticality. [2]

In [2], the researchers propose an interesting architecture based on separating the applications into different execution environments and classifying those environments into Non Critical Execution Environments and Critical Execution Environments. Furthermore, this division also enables the implementation of different communication buses, one critical and another non-critical. This way, the mixed-criticality currently being demanded for ECUs can be achieved.

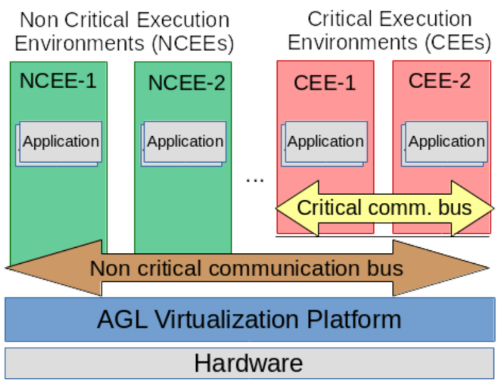

Non Critical Execution Environments (NCEEs)
Critical Execution Environments (CEEs)
NCEE-1
NCEE-2
CEE-1
CEE-2
Application
Application
Application
Application
...
Critical comm. bus
Non critical communication bus
AGL Virtualization Platform
Hardware

Figure 3.1: AGL proposed architecture [2]

The team finally concluded that the next step in this research is the development of a communication bus solution between Execution Environments.

There have been several other studies and projects that have tried to tackle the issue of combining virtualization and mixed-criticality.

3.2 Virtualization solutions integrating modularity, mixed-criticality and re-usability

This section will present solutions that introduce hypervisor virtualization mixed with modularity, mixed-criticality and re-usability demands into an automotive context.

3.2.1 DriveOS

DriveOS is a system created not only as a possible solution to an automotive ECU software but also as a proof of concept in this field. This system basically acts like the VOSYSMonitor mentioned above, although it can also work on X86 hardware. In summary, DriveOS acts like a separation kernel, separating a Real-Time OS from a hypervisor partitioning system. This way, the system can handle time-critical tasks and tasks that require a complex OS and graphical capability. Another topic that is also assured by DriveOS is the strict isolation between the RTOS and the other services implemented in Yocto , only communicating with well-defined channels set up by the developer. The project provides an adaptable collection of tools and a venue where embedded developers from across the world can discuss technologies, software stacks, settings, and best practices that may be used to produce Linux images for embedded devices. This feature is essential in the case of an outside attack that can compromise not only the system itself but also the safety of the users. This isolation also provides the advantage of designing channels of communication with different levels of importance, allowing the system to comply with the goal of mixed-criticality[1]

In [1], the researchers implemented this system and compared it against a Linux system and a barebones Linux system in a series of time-critical and non-time-critical tasks. The researchers concluded that DriveOS performed much better than any Linux version, being that the end-to-end delay of, for example, a USB-CAN-dependent controller loop amounted to 12 times more in Linux than in DriveOS. As a result, it can be said that, as a proof of concept, DriveOS is an exciting solution to keep in mind when designing ECU SW solutions. [1]

3.2.2 ARAMiS II Project:

When researching mixed-criticality, ARAMiS II is one of the most well-funded and supported by the automotive industry. This project is the continuation of the ARAMiS Project[17] that was developed between 2011 and 2015 with the objective of researching the application of multi-core architectures in safely-critical applications. Due to the success of the first project, on 01.10.2016, ARAMiS II was started, focusing on the development and refinement of development tools and platforms to facilitate the creation and use of safety-critical software on multi-core embedded platforms. This project also works closely with different manufacturers in the automotive industry, such as AUDI, DENSO, FAG, in the development of control units for chassis systems over power-train, new control systems for drive systems in electric vehicles and much more.[17]

3.2.3 MultiPARTES FP7 project

MultiPARTES is a Spanish project coordinated by IKERLAN S. COOP and several European universities, supported by the European Commission.[18] This project objective is achieving the integration of mixed-criticality into virtualized embedded systems. As a result, MultiPARTES clearly defines some of the main features that need to be accomplished, such as spatial isolation, temporal isolation, predictability, security, static resource allocation, fault isolation, and management and confidentiality. To achieve these goals, the team proposes using a modified version of XtratuM hypervisor, an open-source, bare-metal hypervisor developed for embedded real-time systems.[7]

3.2.4 Hypervisor for consolidating real-time automotive control units

Developing a new architecture and structure of virtualization and applying it to a state-of-the-art hardware platform is the optimal way of implementing this kind of technology. However, this is not a common practice in the automotive industry. Most times, developers need to implement new features in hardware not designed to support it and are required to support legacy software in new architectures.

In [9], the researchers focused on not only bringing virtualization to a system that did not support it but also supporting legacy software such as AUTOSAR. Due to its ASIL-D certification, the Infineon AURIX family TC29x tri-core controller was chosen [1]. To implement two heterogeneous applications, an engine ECU (E-ECU) and a vehicle control unit (VCU), the researchers chose a prototype version of the hypervisor developed by ETAS GmbH named RTA-HVR. This hypervisor is tailored to handle hard real-time requirements and hardware that does not have virtualization support. After allocating two CPU cores to the E-ECU and one CPU core to the VCU, the team developed a startup sequence of the hypervisor. Subsequently, each VM developed a mechanism for handling traps and interrupts, Input/Output access, etc.

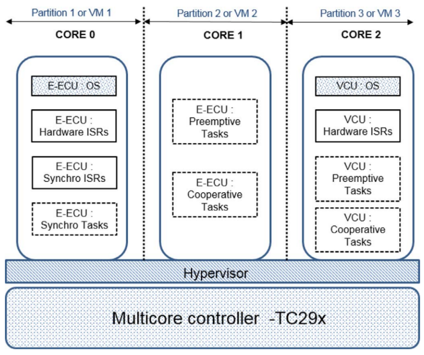


Figure 3.2: Software distribution between Hardware[9]

In this topic, [1] can be very useful when developing the proposed solution, as it describes with great detail every feature designed to achieve the best-case scenario of mixed-criticality. During the testing phase, the team focused on the hypervisor overhead and its influence on hardware interruptions. It was concluded that even though the hypervisor layer caused an overhead, the real-time demands of the system were fulfilled.

3.3 Comparison of current technologies

After researching the use cases in this engineering field, the current technologies stated above can be beneficial in elaborating a solution in this dissertation. In most of these use cases, the solutions presented are mostly closed-source; the software solutions cannot be used in this thesis. However, it can consider all the concerns and features of these solutions and implement them with open-source software. With that, the DriveOS project proposes an innovative architecture separating the general-purpose tasks and real-time tasks. This separation may be the best way of achieving mixed criticality. Another method described was the use of hypervisors that handle directly real-time tasks. This is a different philosophy from the kernel separation, but it can be easier to establish communications between general-purpose guests and real-time guests. In the end, the one goal these projects all strive to achieve is the integration of mixed-criticality in each solution.

Chapter 4

Problem Statement and Proposed Solution

This chapter will focus on exploring the problem at hand and envisioning a solution to address it.

4.1 Problem Statement

In summary, a hypervisor, also referred to as a virtual machine monitor, is software that creates and operates virtual machines. It enables multiple operating systems to operate in their own virtualized environments on a single hardware host.

As stated in the previous chapters, they have many advantages regarding security, costs and efficiency, software consolidation and future-proofing.

However, it is essential to note that using hypervisors presents obstacles. These include the potential performance impact of virtualization, the need for real-time performance in safety-critical systems, and the difficulty of testing and validating such systems.

After studying the topics described in Sections 2 and 3 and researching similar use cases, it can be said that the use of hypervisors in the automotive industry is not a new topic, with a lot of researchers trying to adapt it and create functional solutions to the problem at hand.

That being said, many of these researchers never fully commit to an agreement on a hypervisor solution. This dissertation aims to contribute to this field of engineering by taking into account all the previous attempts and developing a well-founded study on the viability and feasibility of using hypervisors in modern applications. Likewise, a solution that can serve as a proof of concept for future ECU software will also be proposed.

This dissertation will also focus on demonstrating the capability of hypervisor that can be useful in an automotive environment and also test the system capabilities in terms of overhead, comparing it in some cases to a native system.

4.2 Proposed solution

In this way, the proposed solution needs to attend to the various requirements previously mentioned in the previous chapters with the intent of demonstrating the hypervisor's ability to replace the current ECU software deployment method that is containers. With that, a solution was developed, in cooperation with CTW, to tackle the problem at hand. The chosen hardware was a Xilinx Zynq Ultrascale+ MPSoC ZCU106[19] that can accurately represent ECU hardware as embedded systems become more and more powerful and capable of handling big data loads. On the hypervisor front, Xen was chosen for its type 1 open-source characteristic and because it is the official hypervisor supported by Xilinx, the board manufacturer. Four demonstrations were devised to test and demonstrate some of the hypervisor capabilities[20].

The primary objective of the first demonstration is to showcase the network capabilities of the hypervisor. This involves engineering a solution that simulates communication between different components and assesses its performance. As a first test, the network card of the board was allocated to a separate domain, distinct from Dom0. This allocation illustrates the hypervisor's capability to manage device allocation effectively, highlighting its potential for efficient resource utilization and system management. Next, to demonstrate inter-domain communication within Xen, a virtual network interface was allocated in each Domain, connected to a virtual network bridge functioning as the local network switch. This setup was further linked to the physical network interface, establishing a seamless communication link between the virtual and physical realms. The experiment primarily focused on Xen's networking configuration, where a multi-domain environment was found with intricate networking connections. This allowed for a comprehensive exploration of Xen's networking capabilities and its potential to facilitate complex communication scenarios between domains.

More concretely, as seen in Figure 4.1, Xen was setup running on the ZCU106 with a Dom0 control domain and a Dom1 and Dom2 as DomUs. An internal network was created named xenbr0 that connects the three domains and the network card (eth0) attached to the Dom0.

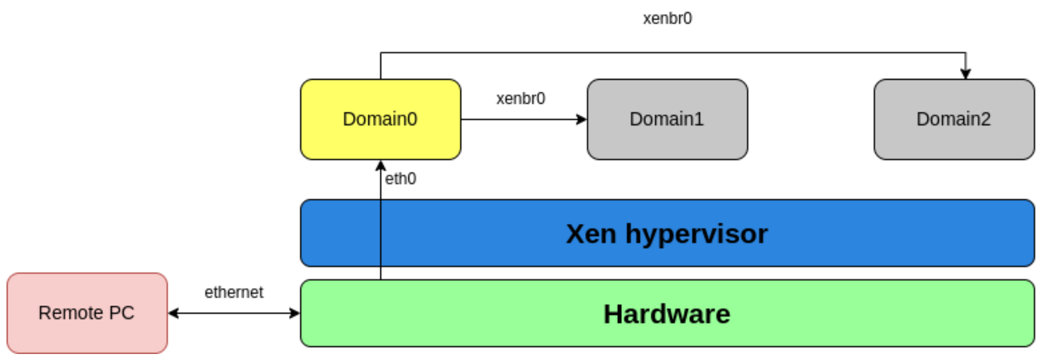


Figure 4.1: Diagram of the Xen network setup

There was a desire to compare it to a native system, so a Linux system was also implemented on the board to simulate a normal device. For the testing of these setups, five measurements were taken:

- **Remote PC to Linux:** This test was taken for baseline purposes.
- **Remote PC to Xen Dom0:** In this test, the performance degradation can be measured because of the xen overhead.
- **Xen Dom0 to Xen Dom1:** This test was made to check the routing performance of the xenbr0 device.
- **Xen Dom1 to Xen Dom2:** This test was made to access a performance comparison with Xen Dom0 to Xen Dom1.
- **Xen Dom1 to Remote PC:** This test was made not only to demonstrate the capability of routing data from Dom0 to Dom1 but also to measure its performance, as it is theoretically the worst performer.

The second demonstration is intended to demonstrate a communication method between domains called Event Channels and shows that domains can be configured with specific shared memory addresses that only the configured domains can access.

A compelling demonstration of the functionality of event channels and shared memory between domains in Xen can be very interesting when condensing two or more ECUs into one system. Event channels are crucial for inter-domain communication, serving as a signaling mechanism for Xen's virtual machines. They play a crucial role in the notification system, notifying domains of events such as data availability and shared memory updates. Shared memory, on the other hand, is an effective method for data exchange between distinct domains. It enhances efficacy by avoiding the resource-intensive necessity of constant data copying between the sender and receiver domains. In the experiment, multiple domains were constructed, and a complex communication system was established using shared memory and event channels.

In Figure 4.2, it can be seen the concrete design of this implementation.

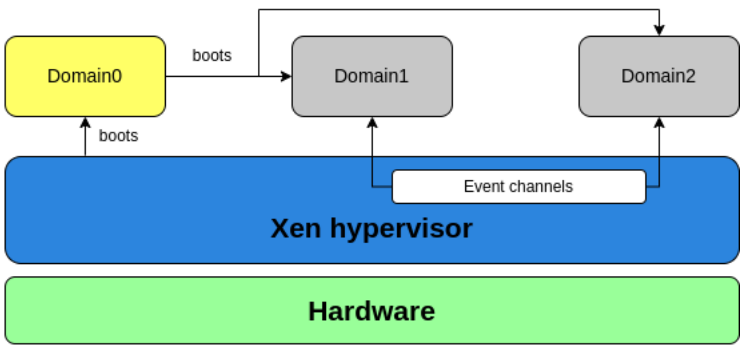


Figure 4.2: Diagram of the Xen event channel setup

For the testing, several tests were made comparing various times the domain2 notified the domain1. The measurements were taken for a period of 20 seconds, testing a 1, 0.1, 0.01, and 0.001 seconds periodicity. There were also three different tests made with this time setup. In the first test, There was only a simple notification from one Domain to another. The second test involves a data write by domain2 to the shared memory and the notification to Domain1, which reads the shared memory address. The last test is more CPU-intensive, in which Domain2 writes to matrices on the shared memory and the notification to Domain1, which reads the shared memory address and computes the data by multiplying it.

Another variable introduced into the tests was the setups with Yoctolinux and Zephyr to verify the performance difference in running a full OS or an RTOS.

Finally, a concluding test was conducted to determine the minimum period between event channels. This test aimed to assess the system's responsiveness and efficiency of event handling. By analyzing the minimum period, valuable insights were gained into the system's ability to handle events swiftly and accurately, providing essential information for optimizing performance and real-time responsiveness.

The third demonstration focused on testing Xen's capability to launch Dom0less domains, meaning the ability to start domains in parallel with Dom0 without requiring Dom0 to be fully loaded first. This feature holds significant importance for the automotive industry, particularly in critical systems that demand rapid boot times when the user starts the car. The experiment involved measuring the time it takes to launch Dom0less domains as well as creating a domain through Dom0. By comparing these launch times, the study aimed to assess the efficiency of Dom0less domain deployment and its potential to enhance automotive systems' boot time performance significantly.

Figure 4.3 shows a diagram of the proposed system.

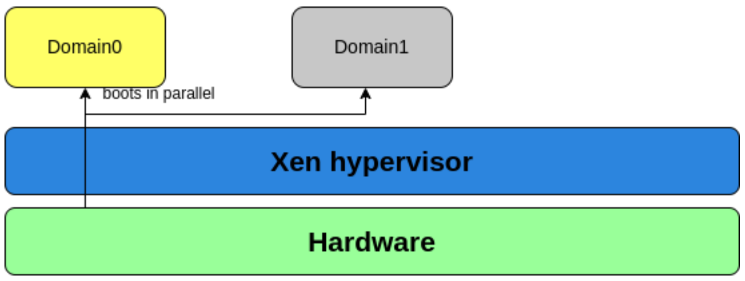


Figure 4.3: Diagram of the Xen dom0less setup

For the fourth and final demonstration, Open Asymmetric Multi-Processing (OpenAMP) was integrated with Xen to exploit the capabilities of a Remote Processing Unit (RPU). OpenAMP, a framework that enables operating systems and application software to take full advantage of multi-processor systems, was viewed as a game-changer when combined with Xen's hypervisor capabilities. An RPU, which is typically found in heterogeneous system-on-chips (SoCs), adds another layer of complexity and processing capability to this system. The experiment focused on implementing Xen and OpenAMP in a multi-core, multi-OS environment where high-performance computation was required, with the RPU playing a crucial role.

In this dissertation, the OpenAMP framework will be implemented in conjunction with the Xen hypervisor with the Remoteproc (Remote Processor Framework) and RPMsg (Remote Processor Messaging) available on the Dom0. Doing this will again prove the versatility of hypervisor software, as they can be combined with other addons such as this. Then, a matrix multiplication demonstration will be tested. Its operations will be explained later. In Figure 4.4, a high-level diagram was created to explain this setup better.

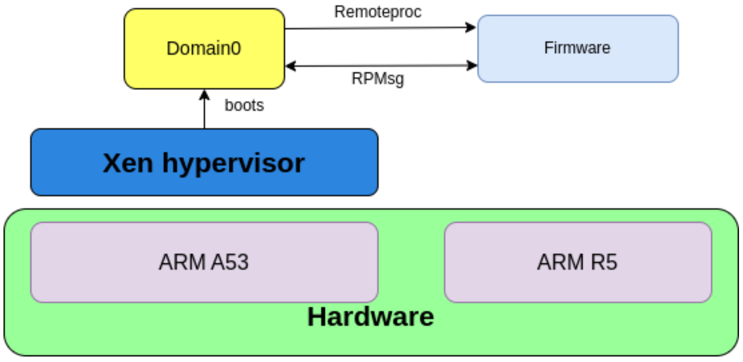


Figure 4.4: Diagram of the Xen OpenAMP setup

Chapter 5

Implementation

This chapter will dive into the setup of the four demonstrations. Firstly there will be a section explaining some of the peculiarities of Xen and details that need to be explained for the demonstrations, then four sections that will demonstrate the work behind all the system configurations but also mentioning the technologies and software used to achieve said configurations.

5.1 General setup

In section 2.4.2 an introduction of the Xen hypervisor was made, however, a more practical in-depth explanation is in order. Xen is a type 1 hypervisor, meaning that it boots directly on top of the hardware. Also, from now on the term domain will be used to reference a virtual machine as this is the term used by this specific hypervisor.

As seen in Figure 5.1, Xen classifies domains into two categories: Domain 0 (Dom0) and Domain U (DomU). Dom0 is the first domain started by the Xen hypervisor on boot and has special privileges like direct hardware access and the ability to manage DomU instances. DomUs are the unprivileged domains that are created and managed by Dom0, and do not have direct access to the physical hardware.

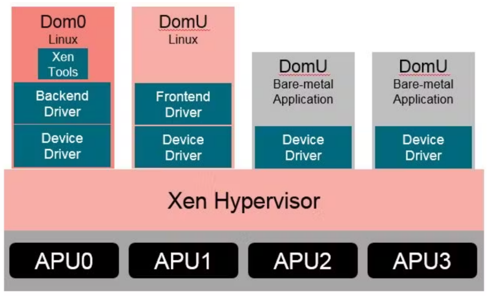


Figure 5.1: High level Xen diagram. [21]

To build the hypervisor, dom0 kernel and root files, the software Petalinux was used. PetaLinux tools is a software development kit (SDK) made by Xilinx, a prominent manufacturer of programmable logic devices to simplify the workflow for hardware and software developers of

Xilinx Field Programmable Gate Array (FPGA) and System on Chip (SoC). The framework also facilitates integration with third-party applications and libraries and supports application development with SDKs for C/C++. The compatibility with the Yocto Project enables PetaLinux to take advantage of a larger ecosystem of tools and layers. Using PetaLinux, custom Linux distributions, kernel configuration, boot-loader setup, generation of root filesystems were made to accommodate every setup requirement [22]. As seen in Figure 5.2 and in Annex A.2, the SDK needs a file that contains the hardware information generaed by another Xilinx program called Vivado. In this project the file used was the one provided in the board support package (BSP) available in the Xilinx download repository. The standard build for Xen and dom0 can be found in Annex A.

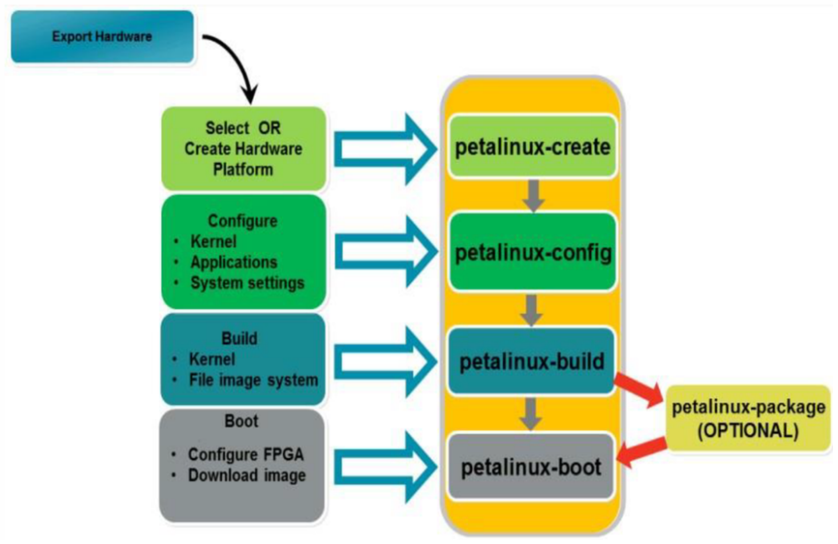


Figure 5.2: Petalinux compilation process. [23]

In order to allocate the right hardware for the system, Device Tree Blob files were used for the different configurations. Device Tree Blob (DTB) is a compact binary representation of a device tree, a data structure that describes the hardware configuration of a particular computer system. Device trees are frequently used in systems running the Linux kernel, particularly in the context of embedded systems and systems-on-a-chip (SoCs), where the hardware configuration can vary widely and is not always automatically detected by the operating system. The device tree depicts the hardware components (such as CPUs, memory, and peripherals) in a system, as well as their configuration details, such as memory addresses and interrupt lines. This information is crucial to the operation of the kernel, which must know how the hardware is organized and where to locate each component.The Device Tree Source (DTS) is a Device Tree source file that is legible by humans. This source file is transformed by the Device Tree Compiler into the binary Device Tree Blob. During launch, the bootloader passes the DTB to the kernel. The kernel can then interpret the system's hardware configuration using the device tree structure [24].

In order to boot Xen, the bootloader needs the ".scr" file to load the desired kernel, dtb, etc. to the the correct memory addresses. A tool called ImageBuilder was used to generate that file. ImageBuilder is a tool that generates a U-Boot script that can be used to automatically install all binaries and launch the entire system quickly. Given a collection of binaries including Xen, Dom0, and a number of Dom0-less DomUs, ImageBuilder and calculates all loading addresses [25]. A standard configuration file to boot Xen is shown in Listing 5.1 .

```
MEMORY_START="0x0"
MEMORY_END="0x80000000"

DEVICE_TREE="xen.dtb"
XEN="xen"
DOM0_KERNEL="Image"
DOM0_CMD="console=hvc0 earlycon=xen earlyprintk=xen clk_ignore_unused root=/dev/
    mmcblk0p2 rw rootwait"
DOM0_MEM=1024

UBOOT_SOURCE="boot.source"
UBOOT_SCRIPT="boot.scr"
```

Listing 5.1: Config for ImageBuilder

Another configuration that needs to be made for convenience is the boot arguments. If the bootloader arguments are not changed the procedure to start Xen is present in Annex A.4 and needs to be inserted at every boot of the board. By doing the procedure described at the end of Annex A.4, we configure the boot sequence to execute the commands automatically in the mmc_boot parameter.

After booting Xen and Dom0, the Dom0 console can be accessed by conecting a USB cable to the UART port on the board. In order to boot a new domain, the xl toolstack compiled by Petalinux tools into the root filesystem of dom0 was used. The xl toolstack provides a comprehensive set of commands to control virtually every aspect of Xen's operation. The xl toolstack can create and destroy domains, list current domains, configure domain settings, and monitor the performance of domains. It can also manage virtual machine migration, both live and non-live [26]. The basic command use of the xl toolstack to boot a new domain is shown in Listing 5.2.

```
#Comand to create a new domain.
#The -c flag means that will automaticaly bind to its console
$ xl create domain.cfg -c
```

Listing 5.2: Creating a new domain

The file shown in Listing 5.3 is the " .cfg" file for the new domain. It is a configuration file used to designate domain parameters. When creating a new domain, these files are typically written in a Python-based syntax and are interpreted by the xl command-line tools.

There are a couple of standard parameters to create a new domain. This is a ".cfg´´ file used to boot a Linux domain with one virtual CPU, 1024 MiB of RAM and a virtual disk with the rootfs.ext4 file system mounted [27]:

```
name = "dom1"
vcpus = 1
memory = 1024
kernel = "/home/root/domains/Image"
disk = ['file:/home/root/domains/rootfs.ext4,xvda,w']
extra = "root=/dev/xvda rw console=hvc0
```

Listing 5.3: The domain.cfg file

With this setup and explanations done, the next sub-chapters will be explained some of the specific setups and test methods needed for the four demonstrations.

5.2 Xen networking demonstration

As previously said on section 4.1, this demonstration will show the network capabilities of Xen and test the overhead/performance degradation that entitles. To better explain the setup of the demo, Figure 5.3 demonstrates the connections made in Xen.

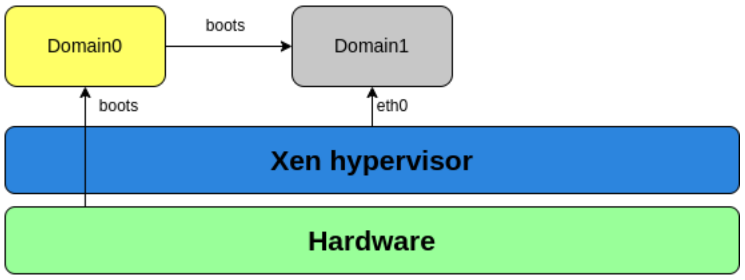


Figure 5.3: Diagram of the Xen device assignment setup

Firstly, to achieve the result in Figure 5.3, the dtb files needs an alteration to signal the Xen hypervisor not to attach the network card eth0 to the dom0 on boot. This way, the dtb file needs to be processed by the dtc package to convert into a dts file that can be human read as in Listing 5.4 .

```
#Comand to create a dts file of a dtb file
$ dtc -I dtb -O dts  xen.dtb > xen.dts
```

Listing 5.4: Creating a dts file from a dtb file.

Then, accessing the dts file, the change to allow the network card not to be assigned can be written as can be seen in Listing 5.5.

```
    ethernet@ff0e0000 {
      compatible = "xlnx,zynqmp-gem\0cdns,zynqmp-gem\0cdns,gem";
      status = "okay";
      interrupt-parent = <0x04>;
      interrupts = <0x00 0x3f 0x04 0x00 0x3f 0x04>;
      reg = <0x00 0xff0e0000 0x00 0x1000>;
      clock-names = "pclk\0hclk\0tx_clk\0rx_clk\0tsu_clk";
      #address-cells = <0x01>;
      #size-cells = <0x00>;
      iommus = <0x0e 0x877>;
      power-domains = <0x0c 0x20>;
      resets = <0x0f 0x20>;
      clocks = <0x03 0x1f 0x03 0x6b 0x03 0x30 0x03 0x34 0x03 0x2c>;
      phy-handle = <0x10>;
      pinctrl-names = "default";
      pinctrl-0 = <0x11>;
      phy-mode = "rgmii-id";
      xlnx,ptp-enet-clock = <0x00>;
      local-mac-address = [ff ff ff ff ff ff];
      phandle = <0x79>;
+     xen,passthrough;

      ethernet-phy@c {
        reg = <0x0c>;
        ti,rx-internal-delay = <0x08>;
        ti,tx-internal-delay = <0x0a>;
        ti,fifo-depth = <0x01>;
        ti,dp83867-rxctrl-strap-quirk;
        phandle = <0x10>;
      };
    };
```

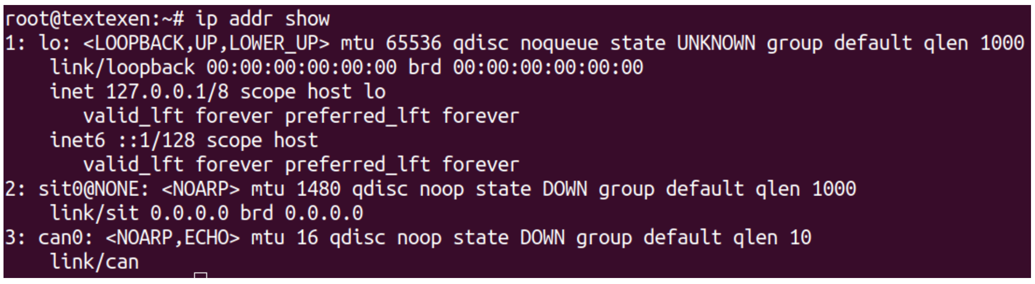
Listing 5.5: Change made to the dts file

After this, the reverse transformation of the file needs to be made.

```
#Comand to create a dtb file of a dts file
$ dtc -I dts -O dtb  xen.dts > xen.dtb
```

Listing 5.6: Creating a dtb file from a dts file

Following this, when the system is initiated it is clear that the network card does not appear in dom0 as can be seen in Figure 5.4.



```
root@textexen:~# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1000
    link/sit 0.0.0.0 brd 0.0.0.0
3: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN group default qlen 10
    link/can
```

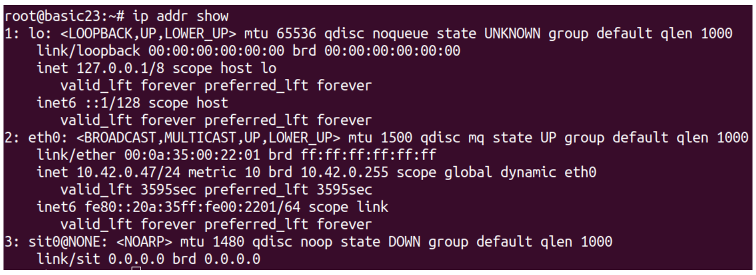
Figure 5.4: Networking devices of dom0 [28].

Then, to setup correctly the network card in the new domain, some parameters need to be added to the .cfg file as seen in Listing 5.7. To give some context to these new parameters, the "dtdev" variable specifies the host device tree nodes to pass-through to this guest. The "device_tree" corresponds to the dtb file that needs to be passed to the domain. In Annex B, there is the full dts file corresponding to the "passthrough_eth.dtb" file [29]. Also, the new .cfg file must contain the interrupt request (IRQ) from the new device that needs to be added to the domain. It’s a hardware signal sent to the processor that temporarily halts a running program and allows a special program, an interrupt handler, to run instead. It’s a way for a device to get the attention of the processor. In this case, the IRQ number is determined by consulting the dts file presented in Annex B , "interrupts = <0x0 0x3f 0x4 0x0 0x3f 0x4>" is the line that contains the address for the interrupts and the desired address is 0x3f which is equal to 63. After that, it needs to be added 32 for the base address of the interrupt handler so the final number comes to 95. In the end, "iomem" represents the specific hardware I/O memory pages.

```
 name = "dom1"
 vcpus = 1
 memory = 1024
 kernel = "/home/root/domains/Image"
 disk = ['file:/home/root/domains/rootfs.ext4,xvda,w']
 extra = "root=/dev/xvda rw console=hvc0"
+dtdev = [ "/axi/ethernet@ff0e0000" ]
+device_tree = "/etc/xen/passthrough_eth.dtb"
+irqs = [ 95 ]
+iomem = [ "0xff0e0,1" ]
```

Listing 5.7: Creating a dtb file from a dts file.

After creating the domain, it is possible to see in Figure 5.5 that the new domain is connected to the desired network card. This way, it was demonstrated one of the many features that can be useful when developing an ECU because, in some situation, it is best to isolate a domain from a connection to an external source.



```
root@basic23:~# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:0a:35:00:22:01 brd ff:ff:ff:ff:ff:ff
    inet 10.42.0.47/24 metric 10 brd 10.42.0.255 scope global dynamic eth0
       valid_lft 3595sec preferred_lft 3595sec
    inet6 fe80::20a:35ff:fe00:2201/64 scope link
       valid_lft forever preferred_lft forever
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1000
    link/sit 0.0.0.0 brd 0.0.0.0
```

Figure 5.5: Networking devices of dom1.

Another important feature that a hypervisor needs to have to be used in the automotive industry is support for the connection between domains. One way is by creating a virtual network. In Figure 5.6, there is example of a detailed view of the setup with the backend domain being dom0.

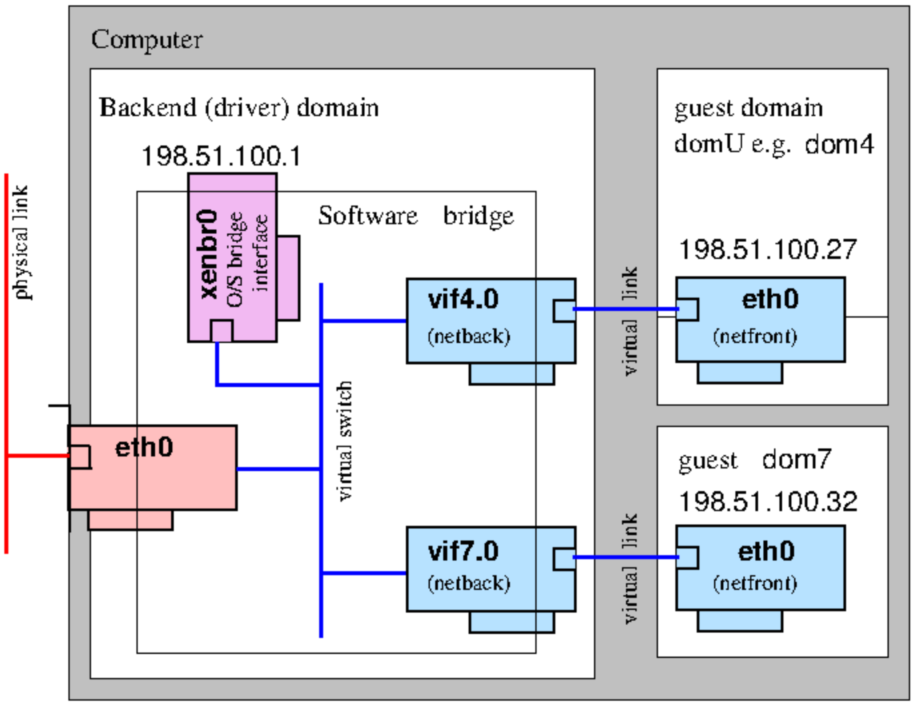


Figure 5.6: Detailed Xen network diagram [30]

To execute this configuration, the network bridge must first be created in dom0 with the script created for this purpose listed in Listing 5.8 . Also, the script contains some commands to enable the forwarding of packets to the other domains that connect to this network bridge. In this way, the other domains will also be available to communicate using the network port.

```
#!/bin/sh
brctl addbr xenbr0
ifconfig xenbr0 192.168.0.1 up
sudo sysctl -w net.ipv4.ip_forward=1
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
sudo iptables -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -i xenbr0 -o eth0 -j ACCEPT
```

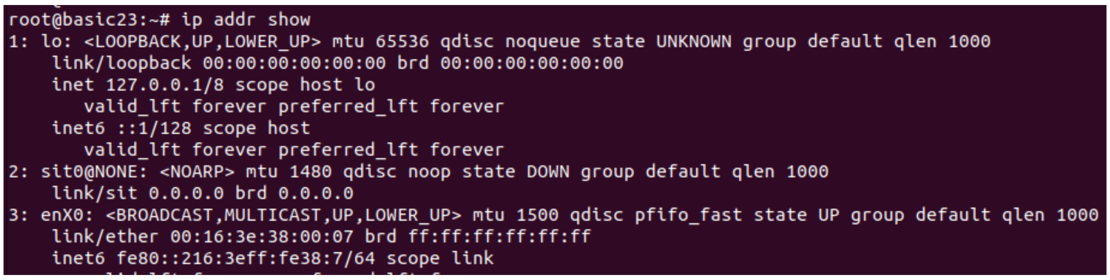
Listing 5.8: Script to create a new bridge with name xenbr0.

Then, the .cfg file of the domains that need to be attached to that bridge need to have the parameter in Listing 5.9 .

```
vif=['bridge=xenbr0']
```

Listing 5.9: Script to create a new bridge with name xenbr0.

This allows the domain to boot with a virtualized network attached, as shown in Figure 5.7.



```
root@basic23:~# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1000
    link/sit 0.0.0.0 brd 0.0.0.0
3: enX0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:16:3e:38:00:07 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::216:3eff:fe38:7/64 scope link
```

Figure 5.7: Dom1 shared network device.

When the domains are fully boot up, it is necessary to configure their IP addresses with the commands present in Listing 5.10.

```
#Command executed in Dom1
ifconfig enX0 192.168.0.2 up
#Command executed in Dom2
ifconfig enX0 192.168.0.3 up
```

Listing 5.10: Script to create a new bridge with name xenbr0

Finally, after this setup has been made, it was necessary to measure the performance of this configuration in order later to establish if the overhead created by the hypervisor can negatively affect the decision of converting the automotive ECUs into a hypervisor based solution.

To measure the performance several tests were made using different tools:

- **Ping:** The Ping command transmits an echo request, and if the remote host detects the target system, it responds with an echo reply. From the response, both the distance (number of steps) to the remote system and the conditions in-between can be determined (packet loss and time to respond) [31].

- **Scp:** The scp (secure copy) command is a powerful tool used in Unix-based operating systems to securely transfer files and directories between a local host and a remote host. It functions over SSH (Secure Shell) protocol, ensuring encrypted and secure data transmission. With the scp command, users can easily copy files from one system to another, either within the same network or across different networks. The basic syntax of the command includes the source and destination, denoting the file or directory to be copied and the target location where it should be placed. This straightforward and efficient utility has become an essential part of system administrators and developers toolkits, facilitating seamless and secure file transfers in various scenarios [32].

- **Neper:** Neper is a Linux networking performance utility that supports multithreading and multi-flows by default. This tool can generate workloads using epoll and accurately capture statistics. Currently, Neper supports six distinct workloads, including tcp rr, tcp stream, udp rr, udp stream, psp stream, and psp rr, which simulate various varieties of network traffic. Neper establishes T threads (workers) for each test, generates F flows (e.g., TCP connections), and distributes the F flows uniformly across the T threads. Each thread is equipped with an epoll to manage multiple flows [33].

- **iPerf3:** iPerf3 is a network performance measurement and optimization tool that actively measures the utmost bandwidth achievable on IP networks. It has the ability to tune numerous timing, buffer, and protocol parameters (TCP, UDP, SCTP with IPv4 and IPv6). This tool is especially useful for determining whether the bandwidth between any two networked computers is sufficient to support the transmission of an application [34].

Next, the setup and test method of the second demonstration will be described.

5.3 Xen event channel and shared memory demonstration

This demonstration has the purpose of showing the ability of the Xen hypervisor the setup a shared memory zone in the systems memory that can be accessed on by the domain that have it configured. Another feature that will be displayed is the event channels in Xen.

Xen provides event channels as a fundamental primitive for delivering asynchronous notifications. They facilitate efficient message passing mechanisms, particularly for communication between domains, drivers and function fundamentally as hardware interrupts.

In the Xen context, an event is indicated by the transition of a single bit from 0 to 1. When an event occurs, the hypervisor layer calls the notified domain to inform them of its arrival (setting the bit). Thereafter, subsequent notifications are masked until this portion is again deleted. Guests must verify the bit's value after re-enabling event delivery to prevent missing notifications. Notably, event notifications can be masked by setting a flag, which is equivalent to disabling interrupts and can be used to guarantee the atomicity of specific visitor kernel operations [35].

In Linux, there are three types of events that can be mapped to event channels:

VIRQs: Per-processor events typically used for timers.

IPIs: Inter-processor interrupts.

PIRQs: Physical hardware interrupts.

Event channels are also essential for inter-Xen domain communication as can be seen in the next chapters.

In terms of the structure and operation of event channels, the Xen public API provides several structures and macros. One of this structures is the Xen hypercalls. Similar to a syscall in an operating system context, a hypercall in Xen represents a software trap from a domain to the hypervisor. A hypercall, like a system call, is synchronous. When a domain makes a hypercall, it requests that the hypervisor execute privileged operations, such as pagetable updates. In contrast to a system call, the return path from the hypervisor to the domain utilizes event channels, which are collections of asynchronous notifications. In this context of event channels, the hypercall used is

HYPERVISOR_event_channel_op. The first parameters of this hypercal is the event channel operations the domain pretends to do. This includes operations like EVTCHNOP_bind_interdomain, EVTCHNOP_bind_virq, EVTCHNOP_bind_pirq, EVTCHNOP_close, EVTCHNOP_send, and EVTCHNOP_status among others [35].

With this short explanation concluded, there needs to be some configuration before access to shared memory in domains.

First, there needs to be an addition in the boot dtb file. In the memory field, there needs to be a parameters that reserves some addresses to be used as shared memory. Listing 5.11 shows an example of adding 4096 bytes of memory starting from address 0x70000000.

```
memory {
    device_type = "memory";
    reg = <0x00 0x00 0x00 0x7ff00000 0x08 0x00 0x00 0x80000000>;
  };
+ reserved-memory {
+   #address-cells = <0x02>;
+   #size-cells = <0x02>;
+   ranges;
+   xen-shmem@70000000 {
+     compatible = "xen,shared-memory-v1";
+     reg = <0x00 0x70000000 0x00 0x1000>;
+   };
+   }
```

Listing 5.11: Changes to dts file to add a shared memory reserved zone.

Then, in the .cfg file of each domain, the address written on the dtb file also needs to be there like in Listing 5.12. There also needs to be a dtb file (Annex C) passthrough that needs to be included.

```
   name = "dom1"
   vcpus = 1
   memory = 1024
   kernel = "/home/root/domains/Image"
   disk = ['file:/home/root/domains/rootfs.ext4,xvda,w']
   extra = "root=/dev/xvda rw console=hvc0"
+  device_tree = "/home/root/domains/passthrough1.dtb"
+  iomem = [ "0x70000,1@0x70000,memory" ]
```

Listing 5.12: Changes to .cfg file to add a shared memory zone

Having explained the configurations needed to enable the shared memory, it will now be explained the testing method of the event channel notification method.

Firstly, the plan was to build a simple C file that used the Xen library available in the root filesystem of the domains that needed to communicate. After using the gcc compiler available on the board to compile the file for the right architecture the file was launched. Unfortunately, after a lot of experimenting, the only results obtained every time the program tried to create an event channel were the ones displayed in Listing 5.13 .

```
    (XEN) d1v0 Rn should not be equal to Rt except for r31
    (XEN) d1v0 unhandled Arm instruction 0x3d800000
    (XEN) d1v0 Unable to decode instruction
    (XEN) traps.c:2059:d1v0 HSR=0x00000092000046 pc=0x00000040001acc gva=0 gpa
   =0000000000000000
```

Listing 5.13: Log from executing the userpace program.

It was later concluded that the userspace of Linux executes in the exception layer 3 (EL3) and the hypercalls can only be executed in the EL1, because the hypervisor is at EL2 so the hypercalls need to be executed on the EL with lower privilege that the hypervisor. Baremetal applications also run in the EL1 as can be seen later on. With this, another strategy was devised. As the kernel space of Linux is executed in El1, using kernel modules was the only solution for this problem. Modules for the Linux kernel are sections of code that can be imported and unloaded on demand. These modules augment the functionality of the kernel without requiring a system restart. They provide the flexibility to add or remove features or drivers as necessary.

The majority of device drivers are implemented as kernel modules. They can be configured as either loadable or built-in. Loadable modules are compiled as distinct object files that can be dynamically imported or unloaded from the kernel at runtime, whereas built-in modules are included in the kernel [36].

The Linux operating system stores kernel modules in the /usr/lib/modules/kernel release directory. The “uname -r” command provides information about the current kernel release version. To observe which kernel modules are presently installed, use the “lsmod” command. In addition, the “modinfo” module name command can be used to display information about a specific module [36].

To create and build a kernel module the commands in Listing 5.14 were used. The petalinux-create command creates the recipe for that module. The main.c file is the one that needs to be edited. After that, using the “petalinux-build” command it will compile and mount the module into the root filesystem [22].

```
# Command to create the new module
   petalinux-create -t modules --name dom1module --enable
# Command to compile the module and to insert it into the root fylesystem
   petalinux-build -c rootfs
```

Listing 5.14: Creating a new kernel module using Petalinux tools

With this explanation, it was possible to elaborate the files in Annex D.

The kernel module for dom1 starts by getting the node where the shared memory is. Next, it allocates the event channels made for the three tests and binds them with their respective function.The fourth event channel is just to print the resulting information in the end of each test. Then, it copies the local ports into the shared memory for the other domain to know where to bind and copies the string "go" so that the second domain knows that the event channels have been created and can bind them.

In the kernel module for dom2 it starts the same way by getting the node where the shared memory is but then hangs until it can read the "go" string from the shared memory. After that, it binds the event channels to the corect port and starts the tests.

The first test is a basic notification during 20 seconds. This test was executed with the periodicity of 1 second, 0.1 seconds, 0.01 seconds and 0.001 seconds.

The second test involves a copy of a integer to the shared memory and the receiving domain to read it during 20 seconds. This test was executed with the periodicity of 1 second, 0.1 seconds, 0.01 seconds and 0.001 seconds.

The third test involves a bigger interaction with the shared memory as it copies two matrices of 2x2 to the shared memory and the receiving domain reads the memory and multiplies the matrices during 20 seconds. This test was executed with the periodicity of 1 second, 0.1 seconds, 0.01 seconds and 0.001 seconds.

After observing the results of these tests, it was decided that it would be good to also add the ability of using a another type of OS to compare its performance. For that, the Zephyr RTOS [15] was chosen.

After executing the setup procedure in Annex E, the 2 domains were created but now using the Xen library of Zephyr. The ".cfg" file of the Zephyr domains, presented in Listing 5.15, also need to change slightly as Zephyr only has a binary kernel and does not need a root filesystem .

```
name="zephyr1"
kernel="/home/root/zephyr/zephyr.bin"
vcpus=1
memory=16
gic_version="v2"
on_crash="preserve"
device_tree = "/home/root/domains/passthrough1.dtb"
iomem = [ "0x70000,1@0x70000,memory" ]
```

Listing 5.15: Zephyr domain ".cfg" file.

Several combinations of tests were made to test if there was any impact in using a full OS. These were the combinations made:

- **Dom1 using Linux and Dom2 using Linux**
- **Dom1 using Linux and Dom2 using Zephyr**
- **Dom1 using Zephyr and Dom2 using Linux**
- **Dom1 using Zephyr and Dom2 using Zephyr**

When using Linux it was also tested if a load on the system had any impact in the performance of the event channels. This way, the program in Annex F was developed. In summary, the program executes a while function to pin the CPU to 100 % during a percentage of a second. This way, if the purpose was to pin the CPU on 25 % utilization, the program executes the while function during 0.25 seconds and then sleeps for 0.75 seconds. This averages in a load of 25 % every second. The number can be change to accommodate the needs of the test. The event channels were tested with a load of 25 %, 50 %, 75 % and 100 % on the dom1 and the dom2.

In order to know how fast these event channels could be, the system was tested to its limit. To do that, a file was created that decreased the time between notifications by multiplying it by 0.1, essentially giving 10 % of the previously tested time. The setup tested was the Zephyr in Dom1 and Zephyr in Dom2, as that would theoretically give the best performance. That file is available in Annex H.

Another test was made to see what would be the reaction of the system to a normal configuration without the previous change in Listing 5.14.

5.4 Xen Dom0less demonstration

This demonstration, aims to show the Xen capability of executing a parallel boot of domains at the boot time of dom0 using the Xen's dom0less feature.

Xen Dom0-less is a feature of the Xen hypervisor that offers a novel approach to virtualization-based static partitioning for the development of mixed-criticality solutions. This feature enables numerous domains to commence at launch time directly from the Xen hypervisor without the need for a privileged Dom0, which is typically required in Xen systems for domain management.

Non-dominant functionality reduces launch times significantly as Xen user-space utilities such as xl and libvirt become optional. It accomplishes this by extending the existing device tree-based Xen launch protocol to include additional domain-specific information. The binaries, such as kernels and ramdisks, are installed by the bootloader (u-boot) and advertised to Xen by means of new device tree bindings. To achieve the dom0less boot, the Imagebuilder config file needs to be changed as shown in Listing 5.16.

```
MEMORY_START="0x0"
MEMORY_END="0x80000000"

DEVICE_TREE="xen-openamp.dtb"
XEN="xen"
DOM0_KERNEL="Image"
DOM0_CMD="console=hvc0 earlycon=xen earlyprintk=xen clk_ignore_unused root=/dev/
    mmcblk0p2 rw rootwait"
DOM0_MEM=1024

+NUM_DOMUS=1
+DOMU_KERNEL[0]="domu/dom0less.bin"

UBOOT_SOURCE="boot.source"
UBOOT_SCRIPT="boot.scr"
```

Listing 5.16: Imagebuilder config file.

The file used for this new domain was made in Vitis IDE, a software tool developed by Xilinx. Vitis IDE is designed to facilitate the development of high-performance applications on Xilinx platforms. As a proof of concept, a small application was created just to compare the boot time of the dom0less domain and the boot time of a domain using the xl framework. The application code can be seen in Annex G .

The application code basically invokes a hypercall that signals the Xen console to print the string inside the brackets.

In order to boot the third domain as soon as the xl framework is available,the script in Listing 5.17 was made to execute the command at boot time.

```bash
#!/bin/bash
xl create -c dom2.cfg
chmod +x run_example.sh #Command to make the script executable
```

Listing 5.17: Script to create the new domain.

Then, the script path was written in a systemd service file to run the script at boot time. Listing 5.18 contains the contents of the systemd service file.

```
[Unit]
Description=Run script at boot

[Service]
ExecStart=<path to script>

[Install]
WantedBy=multi-user.target
```

Listing 5.18: Service file to execute the script

Then the service was configured to execute at boot time with te command "systemctl enable <service name>.service"

The boot time was recorded using the minicom timestamp console on the remote PC connected through the UART0 port on the board. This way, the time between turning on the physical board and the domain output could be measured.

5.5 OpenAMP integration with Xen

For the last demonstration, the OpenAMP framework [37] was configured to work with the Xen hypervisor. OpenAMP (Open Asymmetric Multi-Processing) is a communication framework designed to facilitate efficient communication and collaboration between various processing elements in a heterogeneous system, such as multiple cores or processors. It enables these processing elements to share information and coordinate duties, facilitating their interaction. One CPU is designated as the master and is responsible for administering and orchestrating communication, while the other cores function as slaves. Between processing elements, the framework employs a combination of shared memory and messaging protocols to exchange data. This design improves system performance and resource utilization because each core can focus on its specialized duties while also utilizing the capabilities of other cores when necessary. OpenAMP has applications in a variety of disciplines, including embedded systems, IoT devices, and high-performance computing, where optimizing processing capabilities and inter-core communication is essential for attaining optimal system performance [37].

OpenAMP uses two tools to achieve its goal, Remoteproc and RPMsg. Remote Processor (Remoteproc) and Remote Processor Messaging (RPMsg) are closely related. In a heterogeneous system, they collaborate to facilitate communication and administration between multiple processors.

Remote Processor (Remoteproc) is a feature of the Linux kernel that permits the administration and control of remote processors in a heterogeneous computing environment. It provides a standard interface for booting, loading, and managing the lifecycle of remote processors from the principal (host) processor, which is typically Linux. The principal processor acts as the master, while the remote processors function as slaves. Remoteproc simplifies the management of multiple remote processors by providing a unified interface for their control and interaction. It allows the primary processor to initiate firmware installation, start and halt remote processors, and effectively manage their resources. This makes it simpler for developers to leverage the potential of heterogeneous multi-processing, in which processors with varying capabilities can seamlessly collaborate for enhanced performance and resource efficiency [38].

Complementing Remoteproc, RPMsg (Remote Processor Messaging) is a lightweight, high-performance interprocessor communication protocol. It functions as the communications layer between the primary and remote processors, facilitating the efficient exchange and transmission of data. RPMsg enables processors to send and receive messages, enabling applications running on one processor to communicate transparently with applications running on another processor. This communication is conducted via shared memory buffers, minimizing overhead and enabling interactions with low-latency and high-throughput. RPMsg enables a variety of communication patterns, ranging from simple message transmission to more complex shared memory and remote procedure calls, thereby providing a flexible method of inter-processor communication [39].

With this explanation done, to use OpenAMP in the previous Xen setups it needs to be added to the root filesystem. To do this, the Petalinux tools were used once again. The commands in Listing 5.19 were executed to add OpenAMP to the Xen setup.

```
#In the project folder
petalinux-config -c rootfs

Filesystem Packages --->
-> Petalinux Package Groups
  -> packagegroup-petalinux-openamp
```

Listing 5.19: Adding OpenAMP framework to root filesystem

The device tree also needs to be changed to accommodate the shared memory and the kernel remoteproc.

For that, the first line in Listing 5.20 was appended to the file "project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi" and the second line was appended to the file "project-spec/meta-user/recipes-bsp/device-tree/device-tree.bbappend".

```
/include/ "zynqmp-openamp.dtsi"

SRC_URI:append = "file://openamp.dtsi"
```

Listing 5.20: Changes to the device tree

After this, the "petalinux-build" command was executed so that the Petalinux tools cross-compiled all the previous changes.

For the matrix calculation demonstration, two files were used, one for the master in the APU and another for the slave firmware to be deployed in the RPU. These two files are included in Annex I. To cross-compile the RPU slave firmware, the Vitis IDE was used. To cross-compile the master application, the Petalinux tools were once again used with the commands in Listing 5.21.

```
 petalinux-create -t apps --template install -n <app_name> --enable

 petalinux-build
```

Listing 5.21: Creating a new application using Petalinux tools

This demonstration starts by setting the firmware file as the remoteproc firmware and then starting the remote RPU, which can be seen in Listing 5.22 .

```
#Set the file that the remoteproc will search for in folder /lib/firmware
root@textexen:~# echo image_matrix_multiply > /sys/class/remoteproc/remoteproc0/
#Command to start the firmware
root@textexen:~# echo start > /sys/class/remoteproc/remoteproc0/state
main():128 Starting application...
0 L7 registered generic bus
1 L7 init_system():160 c_buf,c_len = 0x3ed201bc,4096
2 L6 platform_ioot@textexen:~# rm_create_proc()
3 L6 platform_create_proc():103 rsc_table, rsc_size = 0x3ed20000, 0x100
4 L7 zynqmp_r5_a53_proc_init():73 metal_device_open(generic, poll_dev, 0x25f8)
5 L7 platform_create_proc():113 poll{name,bus,chn_mask} = poll_dev,generic,0
    x1000000
6 L7 zynqmp_r5_a53_proc_mmap():138 lpa,lda= 0x3ed20000,0xffffffff
7 L7 zynqmp_r5_a53_proc_mmap():150 mem= 0x3898
8 L7 zynqmp_r5_a53_proc_mmap():154 tmpio= 0x38d8
9 L7 zynqmp_r5_a53_proc_mmap():138 lpa,lda= 0x3ed40000,0xffffffff
10 L7 zynqmp_r5_a53_proc_mmap():150 mem= 0x3920
11 L7 zynqmp_r5_a53_proc_mmap():154 tmpio= 0x3960
12 L6 platform_create_proc():142 Initialize remoteproc successfully.
13 L6 platform_create_rpmsg_vdev():202 creating remoteproc virtio rproc 0x3ed20178
14 L6 platform_create_rpmsg_vdev():210 initializing rpmsg shared buffer pool
15 L6 platform_create_rpmsg_vdev():215 initializing rpmsg vdev
16 L6 app():106 Waiting for events...
```

Listing 5.22: Loading the firmware on the RPU console output.

As the firmware starts, it initiates the remoteproc framework and maps the shared memory previously added to the device tree. Then, it creates the devices needed to use for the RPMsg communication between the processors. Finally, it waits for the remote processor to notify the RPU so it execute the callback function.

To execute the rest of the application, the matrix-multiplication.bin file needs to be executed in Linux, which will be shown in the next chapter.

Chapter 6

Results and performance analysis

This chapter describes the results obtained by the setups previously mentioned in chapter 5 for the four demonstrations.

6.1 Xen networking demonstration

For the first demonstration, the first step was to test the connection between the domains and the external PC needed to be tested to validate the systems connection. In order to do that, the ping command was between all the connection . Furthermore, as the ping framework can also measure the round trip time of packets, the test was executed for 100 packets for all connection like in Listing 6.1.

```
ping -c 100 <ip address of the destination>
```

Listing 6.1: Ping command executed

Table 6.1 displays the results.

Table 6.1: Ping results in milliseconds

	Native Linux			Domain0		
	Min	Avg	Max	Min	Avg	Max
remote pc	0.108	0.132	0.207	0.114	0.139	0.243
dom1	X	X	X	0.146	0.207	0.400
dom2	X	X	X	0.135	0.202	0.366
	Domain1			Domain2		
	Min	Avg	Max	Min	Avg	Max
remote pc	0.352	0.692	0.916	X	X	X
dom2	0.231	0.301	0.704	X	X	X

Evaluating the results, it can be said that, switching from a native Linux setup to a Dom0 in Xen when pinging a remote PC, there is a small increase of 5 % in the average result, not representing a significant change. On the contrary, the Dom1 to the remote PC resulted in an increase of 424 % and 398 % in the average value compared to the native Linux and Dom0 setup respectively. Regarding the virtual network, it is acceptable to affirm that the system behaved as expected as the values of the connection between Dom1 and Dom2 are higher than the values between the dom0 to dom1 connection and the dom0 to dom2 connection do to the fact that the xenbr0 is created by dom0.

Next, a test was made using the scp framework. This test was made as a real life emulation of a secure transfer of files between machines. To generate the file the command in Listing 6.2 .

```
dd if=/dev/urandom of=testfile bs=1M count=1000
```

Listing 6.2: Test file generation

The scp results are summarized in Table 6.2.

Table 6.2: Results of scp test in megabits per second

Connections	Speed
Remote PC to native Linux	29.3
Remote PC to Dom0	14.5
Dom0 to Dom1	18.5
Dom1 to Dom2	10.6

With these results it can be said that the overhead of Xen took a 50 % hit on the performance comparing it to native Linux. The same thing happened on the Dom0 to Dom1 and Dom1 to Dom2.

Even though the scp test already gave transfer speed results, the problem is that it has a lot of overlays attached to it as password encryption and handshakes. For that, other tools were used to understand the behavior of the system regarding TCP and UDP packets as will probably happen in the real world.

After that, the Neper tool was used to measure how the various connections handled more advanced payloads like TCP and UDP packets. With that, the Neper tool was initiated as a server in the Remote PC with the command "tcp_rr". Next, the client was initiated in the Native Linux, Dom0 and Dom1 with the command "tcp_rr -c -H <Server ip>". These procedures were repeated for every connection. Furthermore, the testing modes used were:

- **TCP_rr:** generates request/response workload (similar to HTTP or RPC) over TCP
- **TCP stream:** generates bulk data transfer workload (similar to FTP or scp) over TCP
- **UCP_rr:** generates request/response workload (similar to HTTP or RPC) over UDP
- **UCP stream:** generates bulk data transfer workload (similar to FTP or scp) over UDP

Tables 6.3 and 6.4 were elaborated to easily present the important result that is the throughput. The throughput of the rr tests are meant to be interpreted as transactions per second. The stream test results are in megabit per second.

Table 6.3: Neper tool rr results

	Native Linux	Dom0 to PC	Dom1 to Dom0
TCP_rr test	9088.75	8530.42	5275.31
UDP_rr test	9098.23	8992.83	8334.66
	Dom1 to PC	Dom2 to Dom1	
TCP_rr test	4908.95	4169	
UDP_rr test	6127.58	5698	

Like the ping test already pointed, the best performer is clearly the native Linux due to the non existence of a hypervisor layer causing overhead. Another interesting result is the good performance of the inter domain connections posing as a viable way of communication and fast file transfer between domains. The worst performer, the Dom2 to Dom1 connection, is due to being a connection dependent on Dom0 so, it has to make the redirection from Dom2 to Dom0 and then from Dom0 to Dom1.

As a last test, the “iperf3” tool was used to also measure the Transmission Control Protocol and User Datagram Protocol (TCP and UDP) transfer speeds. The tests were executed in a 10 second frame. The results of this test can be seen in Annex J. As the previous tests also confirmed, there is a small overhead when transferring TCP packets using Dom0 instead of native Linux, with a bigger overhead when transferring using Dom1 to the remote PC. A result to note is the good performance obtained in the inter-domain test of Dom0 to Dom1 transfer, with a worst result in the Dom1 to Dom2 as expected. In the UDP test the results are pretty uniform because of the lower payload this standard has.

6.2 Xen event channel and shared memory demonstration

This demonstration has the objective of validating the event channels method of communication between two domains and then test their performance. In order to do that, the programs already described in Chapter 5 were deployed used to achieve the results.

Firstly, the combination of a Linux domain as a receiver and a Linux domain as a sender were tested. After installing the kernel on the receiving domain, the allocated event channels can be seen by switching to the Xen console by pressing Ctrl+A three times. Then, with the Xen console available, by pressing E the event channels of the system get presented, as can be seen in Listing 6.3 . Normally, a new Linux domain creates five event channels when it boots, all connected to Dom0. Those event channels are used for management such as sending a shutdown command from Dom0. In the Dom1 field it can be seen that the port 6,7,8,9 have been created and available to be connected by other domains. Every event channel has some flags:

- **"s":** This flag represents the status of the event channel. The meaning of different status values can vary depending on the hypervisor’s specific implementation and configuration.

Table 6.4: Neper tool stream results

	Native Linux		Dom0 to PC		Dom1 to Dom0	
	Server	Client	Server	Client	Server	Client
tcp stream	927.27	929.84	444.56	446	808.13	447.11
udp stream	954.91	X	582.44	X	310.98	X

	Dom1 to PC		Dom2 to Dom1	
	Server	Client	Server	Client
tcp stream	425.24	225.63	666.77	268.06
udp stream	239.11	X	306.92	X

- **"n":** This flag indicates the number of notifications that have occurred on the event channel. It represents the count of notifications since the last time the event channel was checked or processed.
- **"x":** This flag indicates the number of times the event channel was unmasked (enabled) without receiving a notification. It represents the count of "spurious" wake-ups or attempts to process the event channel without actual events occurring.
- **"d":** This flag denotes the domain ID or domain number associated with the event channel. In Xen, each virtual machine is assigned a unique domain ID, and this flag identifies the domain to which the event channel is connected.
- **"p":** This flag represents the port number associated with the event channel. The port number is an identifier for the specific event channel and is used to distinguish different event channels within a domain.

```
(XEN) *** Serial input to Xen (type 'CTRL-a' three times to switch input)
(XEN) 'e' pressed -> dumping event-channel info
(XEN) Event channel information for domain 0:
(XEN) Polling vCPUs: {}
(XEN)     port [p/m/s]
(XEN)        1 [0/0/  -   ]: s=3 n=0 x=0 d=0 p=4
(XEN)        2 [0/0/  -   ]: s=5 n=0 x=0 v=2
(XEN)        3 [0/0/  -   ]: s=5 n=0 x=0 v=3
(XEN)        4 [0/0/  -   ]: s=3 n=0 x=0 d=0 p=1
(XEN)        5 [0/0/  -   ]: s=3 n=0 x=0 d=1 p=1
(XEN)        6 [0/0/  -   ]: s=3 n=0 x=0 d=1 p=2
(XEN)        7 [0/0/  -   ]: s=3 n=0 x=0 d=2 p=1
(XEN)        8 [0/0/  -   ]: s=3 n=0 x=0 d=2 p=2
(XEN)        9 [0/0/  -   ]: s=3 n=0 x=0 d=1 p=3
(XEN)       10 [0/0/  -   ]: s=3 n=0 x=0 d=1 p=4
(XEN)       11 [0/0/  -   ]: s=3 n=0 x=0 d=1 p=5
(XEN)       12 [0/0/  -   ]: s=3 n=0 x=0 d=2 p=3
(XEN)       13 [0/0/  -   ]: s=3 n=0 x=0 d=2 p=4
(XEN)       14 [0/0/  -   ]: s=3 n=0 x=0 d=2 p=5
```

```
(XEN) Event channel information for domain 1:
(XEN) Polling vCPUs: {}
(XEN)     port [p/m/s]
(XEN)        1 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=5
(XEN)        2 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=6
(XEN)        3 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=9
(XEN)        4 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=10
(XEN)        5 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=11
(XEN)        6 [0/0/   -   ]: s=2 n=0 x=0 d=2
(XEN)        7 [0/0/   -   ]: s=2 n=0 x=0 d=2
(XEN)        8 [0/0/   -   ]: s=2 n=0 x=0 d=2
(XEN)        9 [0/0/   -   ]: s=2 n=0 x=0 d=2
(XEN) Event channel information for domain 2:
(XEN) Polling vCPUs: {}
(XEN)     port [p/m/s]
(XEN)        1 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=7
(XEN)        2 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=8
(XEN)        3 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=12
(XEN)        4 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=13
(XEN)        5 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=14
```

Listing 6.3: Event channels of the system

When installing the kernel module in the sender domain, the event channels get connected and the sender starts the tests. In Listing 6.4 , the connected event channels can be seen.

```
(XEN) *** Serial input to Xen (type 'CTRL-a' three times to switch input)
(XEN) 'e' pressed -> dumping event-channel info
(XEN) Event channel information for domain 0:
(XEN) Polling vCPUs: {}
(XEN)     port [p/m/s]
(XEN)        1 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=4
(XEN)        2 [0/0/   -   ]: s=5 n=0 x=0 v=2
(XEN)        3 [0/0/   -   ]: s=5 n=0 x=0 v=3
(XEN)        4 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=1
(XEN)        5 [0/0/   -   ]: s=3 n=0 x=0 d=1 p=1
(XEN)        6 [0/0/   -   ]: s=3 n=0 x=0 d=1 p=2
(XEN)        7 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=1
(XEN)        8 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=2
(XEN)        9 [0/0/   -   ]: s=3 n=0 x=0 d=1 p=3
(XEN)       10 [0/0/   -   ]: s=3 n=0 x=0 d=1 p=4
(XEN)       11 [0/0/   -   ]: s=3 n=0 x=0 d=1 p=5
(XEN)       12 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=3
(XEN)       13 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=4
(XEN)       14 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=5
(XEN) Event channel information for domain 1:
(XEN) Polling vCPUs: {}
(XEN)     port [p/m/s]
(XEN)        1 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=5
(XEN)        2 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=6
```

```
(XEN)         3 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=9
(XEN)         4 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=10
(XEN)         5 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=11
(XEN)         6 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=6
(XEN)         7 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=7
(XEN)         8 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=8
(XEN)         9 [0/0/   -   ]: s=3 n=0 x=0 d=2 p=9
(XEN) Event channel information for domain 2:
(XEN) Polling vCPUs: {}
(XEN)     port [p/m/s]
(XEN)         1 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=7
(XEN)         2 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=8
(XEN)         3 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=12
(XEN)         4 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=13
(XEN)         5 [0/0/   -   ]: s=3 n=0 x=0 d=0 p=14
(XEN)         6 [1/1/   -   ]: s=3 n=0 x=0 d=1 p=6
(XEN)         7 [1/1/   -   ]: s=3 n=0 x=0 d=1 p=7
(XEN)         8 [1/1/   -   ]: s=3 n=0 x=0 d=1 p=8
(XEN)         9 [1/1/   -   ]: s=3 n=0 x=0 d=1 p=9
```

Listing 6.4: Event channels of the system

With this, the three tests were executed for this first Linux-Linux setup. The time between each event channel activation was measured by the receiver. In the Figures 6.1, 6.2, 6.3 and 6.4, a bar graphic was elaborated to compare the average timestamps of this setup but with various workloads. This way, it was able to infer that independently of the load placed on both the Linux domains, the resulting timestamps never oscillated enough to conclude that the workload of the domains has an effect on the performance of the event channels.

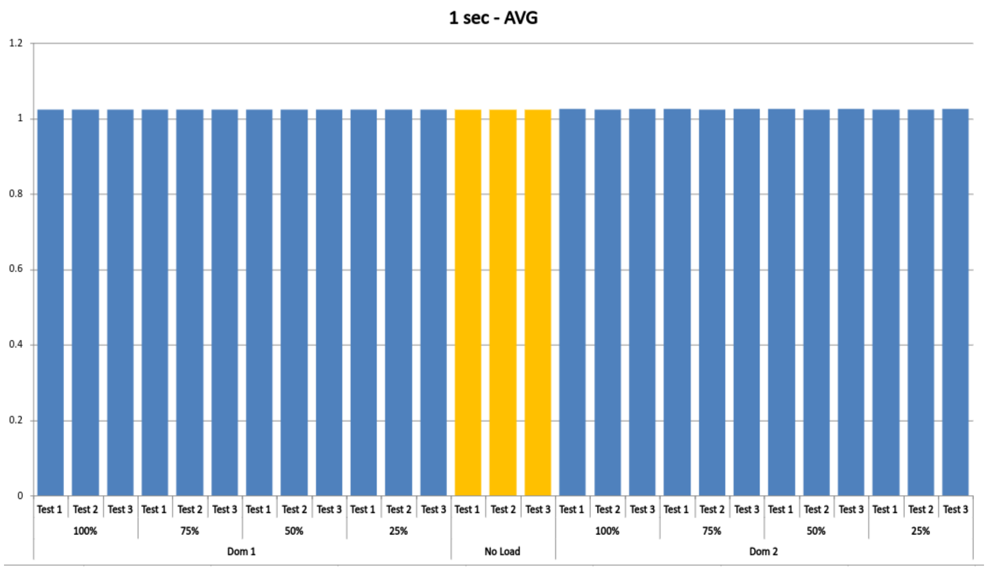


Figure 6.1: Event channel results with 1 sec period.

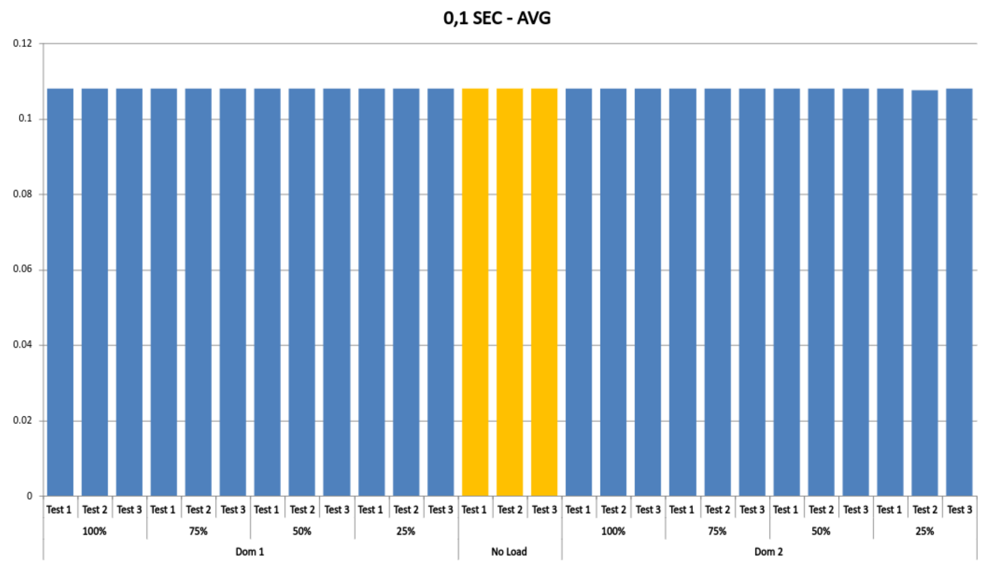


Figure 6.2: Event channel results with 0.1 sec period

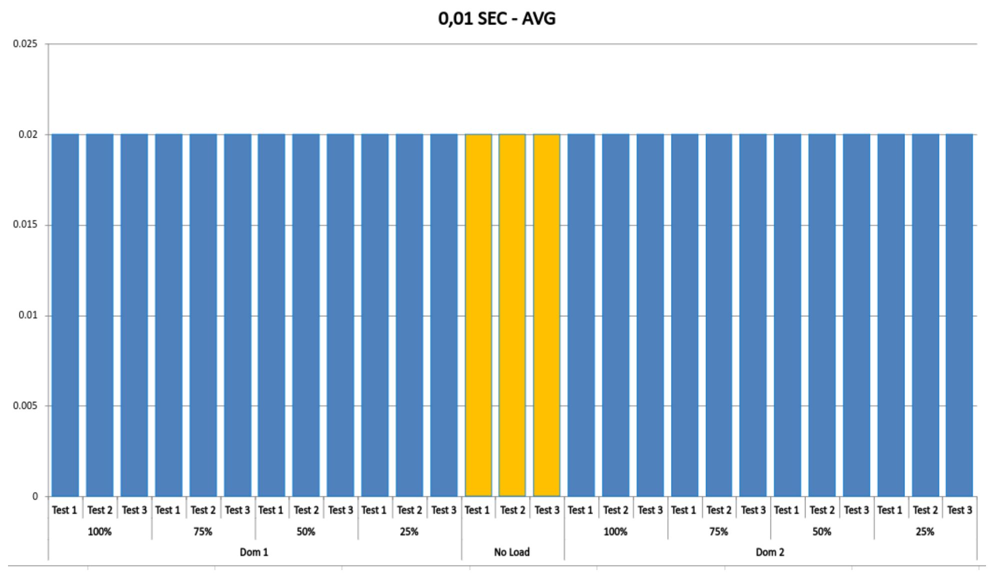


Figure 6.3: Event channel results with 0.001 sec period

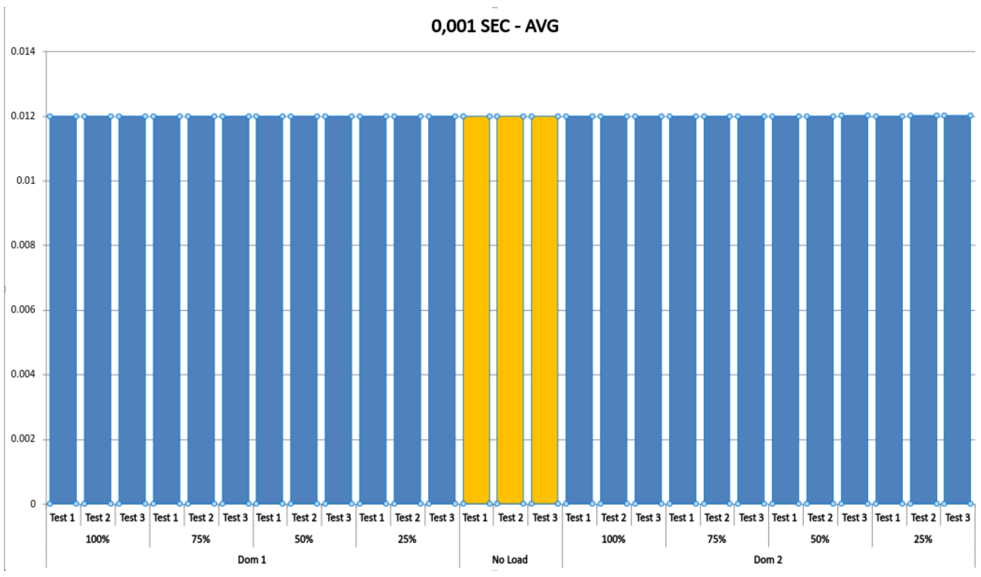


Figure 6.4: Event channel results with 0.0001 sec period

These findings also raised another question, of knowing which domain was the bottleneck in this operation, the sender or the receiver. For that, the tests were run again but with an alteration to the main program. Now, every time the sender ended a test it notified the fourth event channel so that the receiver domain printed the statistics of the received notification amount. With these results, Table 6.5 was elaborated. The horizontal axis corresponds to the Dom2 and the vertical axis corresponds to the Dom1.

Table 6.5: Results of counting the notifications of each domain

		Zephyr		Linux	
		Sent	Received	Sent	Received
Zephyr	Test 1	22220	22220	2873	2873
	Test 2	22220	22220	2873	2873
	Test 3	22220	22220	2873	2873
Linux	Test 1	22220	22220	2873	2873
	Test 2	20383	20383	2873	2873
	Test 3	20383	20383	2873	2873

Through this results it can be concluded that the real bottleneck in these tests was clearly the sender domain as every time the sender was able to send a notification the receiver was able to execute the callback function. Also, when the sender domain was setup with Linux, the results were clearly the worse, only notifying about 13 % of the theoretical 22220 notifications. Another fact that corroborates this conclusion is that the sender achieved the theoretical number of sent notifications even when switching from test1 (only notifying the other domain) to test2 (copying a integer to the shared memory and then notifying) when using zephyr as a sender.

This clearly indicates that calling the hypercall and notifying the event channel poses a bigger overhead than only having only the callback function connected to a interrupt managed by the vcpu in the Linux environment.

Having this knowledge, it was also important to know the limits of these event channels. The log in Listing 6.5 describes the results of the test.

```
....EXECUTING THE FIRST TEST....
Testing 1000 micro
....Notified 10000 times....
Testing 100 micro
....Notified 100000 times....
Testing 10 micro
....Notified 100001 times....
Maximum in test1 is 100 micro
....ENDING THE FIRST TEST....
....EXECUTING THE SECOND TEST....
Testing 1000 micro
....Notified 10000 times....
Testing 100 micro
....Notified 100000 times....
Testing 10 micro
....Notified 100001 times....
Maximum in test1 is 100 micro
....ENDING THE SECOND TEST....
....EXECUTING THE THIRD TEST....
Testing 1000 micro
....Notified 10000 times....
Testing 100 micro
....Notified 100000 times....
Testing 10 micro
....Notified 100001 times....
Maximum in test1 is 100 micro
....ENDING THE THIRD TEST....
```

Listing 6.5: Log from the limit test

As seen in Listing 6.5 the maximum period that event channels can be used is limited to 100 microseconds even with the shared memory operations of test2 and test3.

Finally, the system reaction of not having the memory configured in the ".cfg" file was also tested. This simulates some other domain trying to access a memory region that it should not.

```
root@textexen:~# xl create -c zephyr/zepyr_conf.cfg
Parsing config from zephyr/zepyr_conf.cfg
(XEN) null.c:353: 3 <-- d3v0
(XEN) d3v0: vGICD: unhandled word write 0x000000ffffffff to ICACTIVER4
(XEN) d3v0: vGICD: unhandled word write 0x000000ffffffff to ICACTIVER8
(XEN) d3v0: vGICD: unhandled word write 0x000000ffffffff to ICACTIVER12
(XEN) d3v0: vGICD: unhandled word write 0x000000ffffffff to ICACTIVER16
(XEN) d3v0: vGICD: unhandled word write 0x000000ffffffff to ICACTIVER20
```

```
(XEN) d3v0: vGICD: unhandled word write 0x000000ffffffff to ICACTIVER0
(XEN) traps.c:2059:d3v0 HSR=0x00000093040046 pc=0x00000040007114 gva=0x70000004 gpa
    =0x00000070000004
*** Booting Zephyr OS build zephyr-v3.4.0-1127-gea2aac6e5131 ***
Successfully allocated event channel with port: 3
--- 2 messages dropped ---
Successfully bound callback function to event channel
[00:00:00.008,000] <err> os: ELR_ELn: 0x0000000040007114
[00:00:00.008,000] <err> os: ESR_ELn: 0x0000000096000000
[00:00:00.008,000] <err> os:   EC:  0x25 (Data Abort taken without a change in
    Exception level)
[00:00:00.008,000] <err> os:   IL:  0x1
[00:00:00.008,000] <err> os:   ISS: 0x0
[00:00:00.008,000] <err> os: FAR_ELn: 0x0000000070000004
[00:00:00.008,000] <err> os: TPIDRRO: 0x010000004002b6b0
[00:00:00.008,000] <err> os: x0:  0x0000000070000004  x1:  0x000000004003988c
[00:00:00.008,000] <err> os: x2:  0x0000000000000004  x3:  0x0000000000000000
[00:00:00.008,000] <err> os: x4:  0x0000000000000003  x5:  0x00000000ffffffc8
[00:00:00.008,000] <err> os: x6:  0x0000000000000000  x7:  0x0000000000000040
[00:00:00.008,000] <err> os: x8:  0x0000000000000001  x9:  0xffffffffffffffff
[00:00:00.008,000] <err> os: x10: 0x000000004000c557  x11: 0x0000000040039860
[00:00:00.008,000] <err> os: x12: 0x000000004000c318  x13: 0x0000000040039820
[00:00:00.008,000] <err> os: x14: 0x0000000200007fed  x15: 0x0000000000000000
[00:00:00.008,000] <err> os: x16: 0x0000000000000001  x17: 0x0000000000000005
[00:00:00.008,000] <err> os: x18: 0xffffffffffffffff  lr:  0x000000004000138c
[00:00:00.008,000] <err> os: >>> ZEPHYR FATAL ERROR 0: CPU exception on CPU 0
[00:00:00.008,000] <err> os: Current thread: 0x4001a660 (main)
[00:00:00.012,000] <err> os: Halting system
```

Listing 6.6: Log from wrong memory access test

As seen in Listing 6.6, the Xen hypervisor gives a trap print in the console. That happens because the zephyr app has the shared memory 0x70000000 address configured in the boot procedure and, as it tries to access it as the system boots, the hypervisors traps that communication. After that, when the system tries to access the memory address it crashes because it was not correctly setup at boot.

6.3 Xen Dom0less demonstration

In this demonstration, the system was initiated with the configuration made in Chapter 4. The log file was written in Listing 6.7 with some omissions do to its size.

```
[2023-07-28 11:10:21]Xilinx Zynq MP First Stage Boot Loader
<.....>
[2023-07-28 11:10:27] (XEN) Xen version 4.16.1-pre
<.....>
[2023-07-28 11:10:27] (XEN) *** LOADING DOMAIN 0 ***
<.....>
```

```
[2023-07-28 11:10:27] (XEN) *** LOADING DOMU cpus=1 memory=80000KB
<.....>
[2023-07-28 11:10:28] (XEN) *** Serial input to DOM0 (type 'CTRL-a' three times to
    switch input)
[2023-07-28 11:10:28] (XEN) null.c:353: 0 <-- d0v0
[2023-07-28 11:10:28] (XEN) null.c:353: 2 <-- d1v0
[2023-07-28 11:10:28] (XEN) Freed 344kB init memory.
[2023-07-28 11:10:28] (d1) Hello World
<.....>
[2023-07-28 11:10:34] Welcome to PetaLinux 2022.2_release_S10071807 (honister)!
<.....>
[2023-07-28 11:10:46] (d2) Hello World
<.....>
[2023-07-28 11:10:47]
[2023-07-28 11:10:47] PetaLinux 2022.2_release_S10071807 textexen hvc0
[2023-07-28 11:10:47]
[2023-07-28 11:10:47] textexen login: root (automatic login)
[2023-07-28 11:10:47]
[2023-07-28 11:10:53] root@textexen:~#
```

Listing 6.7: Dom0less boot log

From the log file there is three important values to analyze for this demonstration:

- Dom1 boot message timestamp
- Dom2 boot message timestamp
- Dom0 console timestamp

Analysing the timestamps, the first log will be the reference for turning on the switch on the board. Next, it can be seen that the Dom1 message appears only 7 seconds after turning the board on. The Dom2 message only appears 18 seconds after the Dom1 message and 25 seconds after turning the board on. Finally, the Dom0 console becomes available only 32 seconds after the boards boot. The time for Dom2 message and the Dom0 console can also change depending on the Dom0 root filesystem size and amount of services it needs to boot so, these times can be even worse.

So, if this system gets deployed in a consumer product of the automotive industry, as the consumer unlocks the car the system would turn on and without the dom0less feature, even the most critical systems would take a minimum of 25 seconds to boot wich is not possible in today's world. With dom0less, the boot time can be shortened by 72 %, which makes these systems much more viable.

6.4 OpenAMP integration with Xen

In this last demonstration, the purpose is to show the OpenAMP framework functioning in conjunction with the Xen hypervisor. After executing the setup mentioned in the Chapter 4, the master binary application needs to be executed in the Linux OS running on the APU. Listing 6.8 is the result of running the binary app.

```

root@textexen:~# matrix-test

 Matrix multiplication demo start

Master>probe rpmsg_char
+ lsmod
Module                    Size  Used by
rpmsg_char               16384  0
mali                    229376  0
uio_pdrv_genirq          16384  0
xen_netback              53248  0
xen_blkback              36864  0
xen_gntalloc             16384  0
xen_gntdev               28672  2
dmaproxy                 16384  0
+ modprobe rpmsg_char
lookup_channel():265 using dev file: virtio0.rpmsg-openamp-demo-channel.-1.1024
bind_rpmsg_chrdev():161 open /sys/bus/rpmsg/devices/virtio0.rpmsg-openamp-demo-
    channel.-1.1024/driver_override
bind_rpmsg_chrdev():184 write virtio0.rpmsg-openamp-demo-channel.-1.1024 to /sys/
    bus/rpmsg/drivers/rpmsg_chrdev/bind
get_rpmsg_chrdev_fd():204 opendir /sys/bus/rpmsg/devices/virtio0.rpmsg-openamp-demo
    -channel.-1.1024/rpmsg
get_rpmsg_chrdev_fd():214 open /dev/rpmsg_ctrl1
main():332 rpmsg_create_ept: rpmsg-openamp-demo-channel[src=0,dst=0x400]
checking /sys/class/rpmsg/rpmsg_ctrl1/rpmsg0/name
svc_name: rpmsg-openamp-demo-channel
.
open /dev/rpmsg0
main():352 matrix_mult(1)

 Master : Linux : Input matrix 0

 6   4   3   7   0   0
 6   6   7   1   6   8
 8   0   1   8   5   7
 7   1   5   7   7   1
 9   5   1   9   5   6
 3   2   0   6   9   3

 Master : Linux : Input matrix 1

 6   6   9   5   9   7
 6   9   7   7   8   4
 6   7   7   3   4   5
 6   6   2   0   5   7
 6  0  9  8  7  1
 1  5  9  2  3  0
0: write rpmsg: 296 bytes
read results

 Master : Linux : Printing results
 120  135  117  67  133  122
 164  185  273  157  201  114
 139  138  203  97  172  122
 163  133  191  115  178  134
 180  190  240  135  223  156
 123  87  161  107  145  80
End of Matrix multiplication demo Round 0
```

Listing 6.8: Log from APU console after executing the application

As can be seen in Listing 6.8, the application starts by loading the RPMsg kernel module necessary. Then, it configures the device previously created by the RPU firmware and binds a device driver to it. After that, it opens the device to send the information it needs to send to the RPU firmware, generates the two matrices and writes them to the inter-processor device. It then waits until the full resulting matrix has been written by the RPU and prints the result.

In conclusion, it was proven that the OpenAMP framework can be integrated with the Xen hypervisor. The integration of the OpenAMP framework with the Xen hypervisor provides significant resource management, isolation, and communication benefits for multiprocessor systems. By integrating OpenAMP with Xen, there is not only the possibility of creating other domains in the same CPUs but also the ability of offloading real-time applications to other CPUs (in the case of the hardware used in this dissertation a real-time CPU) thereby improving system scalability and workload distribution. In addition, the isolation capabilities of Xen assure robust partitioning and fault tolerance between domains, thereby enhancing system dependability and security. This combination enables a flexible and dynamic system architecture, permitting efficient hardware resource utilization, seamless processor-to-processor communication, and the coexistence of diverse operating systems, making it ideal for complex embedded systems and heterogeneous multi-core platforms as are automotive ECUs.

Chapter 7

Conclusion and Future work

This chapter will summarize the contents of the previous chapters and provide the concluding verdict regarding the value of developing this solution. In the Future Work section, new ideas and considerations for the application of the work produced in this dissertation will be made.

7.1 Conclusion

This dissertation started with a view of the existing technologies used for this line of work: hypervisors. The main objective stated was to develop a study into hypervisors, devise a solution and test said solution in some aspects that would be relevant to the automotive industry. In chapter 4, the proposed solution using a Xen hypervisor and some demonstrations that showed the capabilities of this hypervisor were presented. These demonstrations proved that the Xen hypervisor has the capability of being a great alternative to the current architecture for automotive ECUs. In the networking demonstration, several features were displayed, such as the capability of Xen to attach hardware devices to domains and isolate them from other domains. This security characteristic is fundamental in the automotive industry. Another discovery made were the performance numbers of the various networking connections made using Xen, which not only proved the flexibility of this software but also that the performance overhead that a hypervisor causes is not that serious in the domain with a network card directly attached. In the event channel and shared memory demonstration, the security characteristics were again reinforced as the results of the last test in which a domain tried to access memory that was not meant to give the best outcome in terms of security. It was also shown that event channels can be a very fast communication process for inter-domain communication. In the Dom0less demonstration, another helpful feature for the automotive industry was displayed as the domain configured to boot in parallel with the Dom0 booted much faster than waiting for the whole xl framework to initiate. The last demonstration proved that the OpenAMP framework could work in conjunction with the Xen hypervisor, opening the door for several interesting setups regarding real-time computing and resource management.

This solution paves the way for the development of a complex system and shows that a hypervisor solution has the capabilities for successful development for the automotive industry. Therefore, it justifies the endeavor to develop a more complex system.

7.2 Future work

This dissertation investigated the viability of hypervisor technology for automotive ECUs. Nevertheless, several promising topics for future research can contribute to practical advances and real-world applications in automotive computing.

7.2.1 Testing Real-time Capability with OpenAMP and Expanded Hardware

To validate the practicality of hypervisor-based ECUs in real-time systems, future work should consist of extensive experimentation with OpenAMP on more hardware platforms. Incorporating real-time scenarios, such as safety-critical applications, will aid in evaluating the hypervisor's ability to adhere to stringent timing constraints.

7.2.2 Optimizing Xen Configuration for Enhanced Performance

Additional research can be conducted by modifying Xen's configuration parameters, such as vCPUs (virtual Central Processing Units), memory allocation, and CPU schedulers. Experimenting with various configurations can shed light on maximizing resource utilization, minimizing overhead, and attaining optimum performance for automotive ECUs.

7.2.3 Practical Implementation of ECU Software

Implementing a prototype of an actual ECU software system should be the following step for validating the efficacy of the hypervisor approach in real-world scenarios. This implementation may include automotive-specific applications and features. The practicality and dependability of the hypervisor-based ECU must be determined by measuring the system's performance under various conditions, such as dynamic workloads and real-time constraints.

References

[1] Soham Sinha and Richard West. Towards an Integrated Vehicle Management System in DriveOS. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.

[2] The Linux Foundation. The Automotive Grade Linux Software Defined Connected Car Architecture, 2018. URL: https://www.automotivelinux.org/wp-content/uploads/sites/4/2018/06/agl_software_defined_car_jun18.pdf.

[3] Humble Devassy Chirammal, Prasad Mukhedkar, and Anil Vettathu. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.

[4] Michael Eder. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 1, 2016.

[5] Moritz Raho, Alexander Spyridakis, Michele Paolino, and Daniel Raho. KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing. In *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–8, 2015. doi:10.1109/AIEEE.2015.7367280.

[6] RedHat. What is KVM?, 2022. URL: https://www.redhat.com/en/topics/virtualization/what-is-KVM.

[7] Salvador Trujillo, Alfons Crespo, and Alejandro Alonso. MultiPARTES: Multicore Virtualization for Mixed-Criticality Systems. In *2013 Euromicro Conference on Digital System Design*, pages 260–265, 2013. doi:10.1109/DSD.2013.37.

[8] Indeed Editorial Team. 7 Important Certifications for Automotive Companies. URL: https://www.indeed.com/career-advice/career-development/automotive-industry-certifications.

[9] Arun Kumar Sundar Rajan, Armin Feucht, Lothar Gamer, Idriz Smaili, et al. Hypervisor for consolidating real-time automotive control units: Its procedure, implications and hidden pitfalls. *Journal of Systems Architecture*, 82:37–48, 2018.

[10] Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 2021.

[11] Hermann Kopetz. *Real-Time Systems*. Springer New York, NY, 2011.

[12] Kasim M. Al-Aubidy. Classification of Real-Time Systems. University Lecture, 2011.

[13] Market leading rtos (real time operating system) for embedded systems with internet of things extensions. URL: https://www.freertos.org/.

[14] SAFERTOS, the safety certified RTOS - available pre-certified to IEC 61508. URL: https://www.highintegritysystems.com/safertos/.

[15] Zephyr Project Documentation 2023, Jul 2023. URL: https://docs.zephyrproject.org/latest/introduction/index.html.

[16] The Yocto Project Overview and Concepts Manual. URL: https://docs.yoctoproject.org/overview-manual/intro.html.

[17] ARAMIS II project. URL: http://www.aramis2.com/project/.

[18] Multi-cores Partitioning for Trusted Embedded Systems, 2022. URL: https://cordis.europa.eu/project/id/287702.

[19] Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit. URL: https://www.xilinx.com/products/boards-and-kits/zcu106.html.

[20] Xilinx Wiki/Open Source Projects/XEN Hypervisor, Jun 2021. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1947959300/Overview.

[21] AI at the Edge: Xen Hypervisor on the ZCU102. URL: https://www.hackster.io/whitney-knitter/ai-at-the-edge-xen-hypervisor-on-the-zcu102-067333.

[22] PetaLinux Tools Documentation: Reference Guide (UG1144). URL: https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide.

[23] Xilinx Support. URL: https://support.xilinx.com/s/article/1066813?language=en_US.

[24] Device Tree (dtb) - postmarketOS. URL: https://wiki.postmarketos.org/wiki/Device_Tree_(dtb).

[25] ViryaOS / imagebuilder · GitLab. URL: https://gitlab.com/ViryaOS/imagebuilder.

[26] xl - Xen management tool, based on libxenlight. URL: https://xenbits.xen.org/docs/unstable/man/xl.1.html.

[27] xl.cfg - xl domain configuration file syntax. URL: https://xenbits.xen.org/docs/unstable/man/xl.cfg.5.html.

[28] Xen Networking - Xen. URL: https://wiki.xenproject.org/wiki/Xen_Networking.

[29] Xilinx. xen-passthrough-device-trees/device-trees-2021.2/zcu102/ethernet@ff0e0000.dts at master · Xilinx/xen-passthrough-device-trees. URL: https://github.com/Xilinx/xen-passthrough-device-trees/blob/master/device-trees-2021.2/zcu102/ethernet%40ff0e0000.dts.

[30] Xen Networking. URL: https://wiki.xenproject.org/wiki/Xen_Networking.

[31] Linux ping command. URL: https://www.tutorialspoint.com/linux-ping-command.

[32] SCP Linux Command – How to SSH File Transfer from Remote to Local_2021, Sep 2021. URL: https://www.freecodecamp.org/news/scp-linux-command-example-how-to-ssh-file-transfer-from-remote-to-local/.

[33] google. GitHub - google/neper: neper is a Linux networking performance tool., May 2023. URL: https://github.com/google/neper.

[34] Vivien GUEANT. iPerf - The TCP, UDP and SCTP network bandwidth measurement tool. URL: https://iperf.fr/.

[35] Event Channel Internals - Xen. URL: https://wiki.xenproject.org/wiki/Event_Channel_Internals.

[36] Kernel module - ArchWiki. URL: https://wiki.archlinux.org/title/Kernel_module.

[37] Libmetal and OpenAMP User Guide (UG1186). URL: https://docs.xilinx.com/r/en-US/ug1186-zynq-openamp-gsg/OpenAMP.

[38] Index of /doc/Documentation/remoteproc.txt. URL: https://www.kernel.org/doc/Documentation/remoteproc.txt.

[39] Remote Processor Messaging (rpmsg) Framework — The Linux Kernel documentation. URL: https://docs.kernel.org/staging/rpmsg.html.

Appendix A

Xen hypervisor build process and hardware deployment

This procedure was executed in the following environment:

• Host PC with Linux OS • Petalinux version 2022.2 already installed

The first step is initializing the Petalinux tools:

```
source /<path to petalinux tools folder>/settings.sh
```

A.1 Create a Petalinux Project

Next, the project needs to be created in the desired directory. The template used in this case is zynqMP as it is the model of the fpga used:

```
petalinux-create -t project --template zynqMP -n xen
```

The project needs the .xsa file. The file used in this dissertation was the one provided in the board support package available in Xilinx's website.

A.2 Hardware and Rootfs configuration

```
cd <project directory>/xen
petalinux-config --get-hw-description <PATH-TO-XSA-FILE>
```

Then, Xen needs to be enabled in the root filesystem:

```
petalinux-config -c rootfs

Petalinux Package Groups  ---> packagegroup-petalinux-xen   --->  [*] packagegroup-
    petalinux-xen
Filesystem Packages ---> console ---> tools ---> xen ---> [*] xen-tools
```

A.3 Device tree changes

After this, the device tree needs to be edited in order to add extra Xen related configurations. The file is:

```
cd project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi
```

It needs to look like this:

```
/include/ "system-conf.dtsi"
/include/ "zynqmp-xen.dtsi"
/ { };
```

Another file also needs to be edited:

```
project-spec/meta-user/recipes-bsp/device-tree/device-tree.bbappend
```

The first lines should look like this:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:${SYSCONFIG_PATH}:"

SRC_URI:append = " file://config file://system-user.dtsi"
SRC_URI:append = " file://zynqmp-xen.dtsi"
```

A.4 Build and hardware deployment

In the end, execute the build command:

```
petalinux-build
```

After the Petalinux tools compile the build, the generated files need to be transferred to an SD card that has two patitions, one for the boot files and another one for the root files. The boot partition needs to be approximately 5GB to accommodate all the boot files and domains that may need to be booted in dom0less mode.

In the projects directory, these files need to be copied to the boot partition:

```
cp linux/xen/xen.dtb /<path to media devices>/boot/
cp linux/xen/Image /<path to media devices>/boot/
cp linux/xen/xen_boot_sd.scr  /<path to media devices>/boot/
cp linux/xen/xen  /<path to media devices>/boot/
```

Then the root filesystem needs to be copied to the root partition:

```
sudo tar xvf images/linux/rootfs.tar.gz -C /<path to media devices>/root
```

After inserting the memory card in the board and powering it on the bootloader will come up:

```
Xilinx Zynq MP First Stage Boot Loader
Release 2022.2   Oct  7 2022  -  04:56:16
NOTICE:  BL31: v2.6(release):xlnx_rebase_v2.6_2022.1_update3-18-g0897efd45
 NOTICE:  BL31: Built : 03:55:03, Sep  9 2022
```

```
U-Boot 2022.01 (Sep 20 2022 - 06:35:33 +0000)
CPU:   ZynqMP
Silicon: v3
Board: Xilinx ZynqMP
DRAM:  4 GiB
PMUFW:    v1.1
PMUFW no permission to change config object
EL Level: EL2
Chip ID:  zu7e
NAND:  0 MiB
MMC:   mmc@ff170000: 0
Loading Environment from FAT... *** Warning - some problems detected reading
   environment; recovered successfully
OK
In:    serial
Out:   serial
Err:   serial
Net:   FEC: can't find phy-handle
No ethernet found.
scanning bus for devices...
SATA link 0 timeout.
SATA link 1 timeout.
AHCI 0001.0301 32 slots 2 ports 6 Gbps 0x3 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
starting USB...
No working controllers found
Hit any key to stop autoboot:  0
```

Hit any key to stop the autoboot. Then insert this into the console:

```
load mmc 0:1 0xC00000 xen_boot_sd.scr; source 0xC00000
```

This will load the xen_boot _sd.scr file into the correct memory address.

To do this automatically just insert this command in the bootloader:

```
setenv mmc_load="load mmc 0:1 0xC00000 xen_boot_sd.scr; source 0xC00000"
```

Appendix B

Dts file with Ethernet passthrough

File for the ethernet pasthrough dtb. This file is composed by the ethernet device of the system to enable it in the new domain:

```
/dts-v1/;

/ {
  #address-cells = <0x2>;
  #size-cells = <0x2>;
  passthrough {
    compatible = "simple-bus";
    ranges;
    #address-cells = <0x2>;
    #size-cells = <0x2>;
    misc_clk {
      #clock-cells = <0x0>;
      clock-frequency = <0x7735940>;
      compatible = "fixed-clock";
      phandle = <0x1>;
    };
    ethernet@ff0e0000 {
      compatible = "cdns,zynqmp-gem";
      status = "okay";
      interrupt-parent = <0xfde8>;
      interrupts = <0x0 0x3f 0x4 0x0 0x3f 0x4>;
      reg = <0x0 0xff0e0000 0x0 0x1000>;
      clock-names = "pclk", "hclk", "tx_clk", "rx_clk";
      #address-cells = <0x1>;
      #size-cells = <0x0>;
      clocks = <0x1 0x1 0x1 0x1>;
      phy-mode = "rgmii-id";
      xlnx,ptp-enet-clock = <0x0>;
      local-mac-address = [00 0a 35 00 22 01];
      phy-handle = <0x2>;
      xen,reg = <0x0 0xff0e0000 0x0 0x1000 0x0 0xff0e0000>;
      xen,path = "/axi/ethernet@ff0e0000";
      phy@c {
        reg = <0xc>;
        ti,rx-internal-delay = <0x8>;
        ti,tx-internal-delay = <0xa>;
        ti,fifo-depth = <0x1>;
        ti,rxctrl-strap-worka;
        phandle = <0x2>;
      };
    };
  };
};
```

Appendix C

Dts file with shared memory

File for the shared memory passthrough dtb. This file is composed by the reserved memory range that the new domain should be access:

```
/dts-v1/;

/ {
  #address-cells = <0x02>;
  #size-cells = <0x02>;
  passthrough {
    #address-cells = <0x02>;
    #size-cells = <0x02>;
    ranges;
    reserved-memory {
      #address-cells = <0x02>;
      #size-cells = <0x02>;
      ranges;

      xen-shmem@70000000 {
        compatible = "xen,shared-memory-v1";
        reg = <0x0 0x70000000 0x0 0x1000>;
      };
    };
    memory {
      device_type = "memory";
      reg = <0x0 0x70000000 0x0 0x1000>;
    };
  };
};
```

Appendix D

Kernel module files

File used for the Dom1 (receiver) kernel module.

This file is composed by:

- **The libraries used:**

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/of_address.h>
#include <linux/of_irq.h>
#include <xen/events.h>
#include <xen/xenbus.h>
#include <xen/xen.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/debugfs.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#include <xen/events.h>
#include <xen/xen.h>

```

- **Module information:**

```
    MODULE_LICENSE("GPL");
MODULE_AUTHOR
    ("Xilinx Inc.");
MODULE_DESCRIPTION
    ("testmodule - loadable module template generated by petalinux-create -t
    modules");

```

- **The callback functions and other structures:**

```
    static char *shared_mem;
```

```
static int irq1,irq2,irq3,irq4;
static int count=0;
static int random_numbers[4];
static int count1=0;
static int count2=0;
static int count3=0;
int count_total1=0;
int count_total2=0;
int count_total3=0;

struct timespec64 current_time_new, last_interrupt_time, delta_time_new;


void multiply_and_print_matrix(int *numbers) {

  int matrix[2][2];  // Define a 2x2 matrix
  int result[2][2] = {0};  // Define a 2x2 result matrix
  int i, j, k;
  // Arrange the numbers into the matrix
  matrix[0][0] = numbers[0];
    matrix[0][1] = numbers[1];
    matrix[1][0] = numbers[2];
    matrix[1][1] = numbers[3];
  // Multiply the matrix by itself
  for(i = 0; i < 2; ++i) {
    for(j = 0; j < 2; ++j) {
      for(k = 0; k < 2; ++k) {
        result[i][j] += matrix[i][k] * matrix[k][j];
      }
    }
  }
  // Print the resulting matrix
  /*for(i = 0; i < 2; ++i) {
    for(j = 0; j < 2; ++j) {
      printk(KERN_INFO "%d ", result[i][j]);
    }
    printk(KERN_INFO "\n");
  }*/
}

void get_evtchn_ts(void)
{
  /* Get the current time */
    ktime_get_real_ts64(&current_time_new);

    /* Calculate the time difference */
    delta_time_new = timespec64_sub(current_time_new, last_interrupt_time);

  if (last_interrupt_time.tv_sec==0 || delta_time_new.tv_sec>5)
  {
    printk(KERN_ALERT "First ts");
  }
  else
  {
    printk(KERN_ALERT "Time:%lld.%09ld", (long long)delta_time_new.tv_sec,
    delta_time_new.tv_nsec);

    }
  /* Store the current time for the next calculation */
  last_interrupt_time = current_time_new;
}
static irqreturn_t evtchn_handler1(int port,void *data)
{
    //get_evtchn_ts();
  count1++;
        /* Print the time difference in seconds and nanoseconds */
  xen_irq_lateeoi(irq1,0);
  return IRQ_HANDLED;
}

static irqreturn_t evtchn_handler2(int port,void *data)
{
  memcpy(&count, shared_mem+16, sizeof(count));
  //get_evtchn_ts();
  count2++;
         /* Print the time difference in seconds and nanoseconds */
  xen_irq_lateeoi(irq2,0);
  return IRQ_HANDLED;
}
static irqreturn_t evtchn_handler3(int port,void *data)
{
  memcpy(&random_numbers, shared_mem+20, sizeof(random_numbers));
  multiply_and_print_matrix(random_numbers);
  //get_evtchn_ts();
  count3++;
    /* Print the time difference in seconds and nanoseconds */
  xen_irq_lateeoi(irq3,0);
  return IRQ_HANDLED;
}

static irqreturn_t evtchn_handler4(int port,void *data)
{
    count_total1=count_total1+count1;
    count_total2=count_total2+count2;
    count_total3=count_total3+count3;
    printk(KERN_ALERT "\n\nFirst interrupt activated:%d\n", count1);
    printk(KERN_ALERT "Second interrupt activated:%d\n", count2);
    printk(KERN_ALERT "Third interrupt activated:%d\n\n", count3);
    printk(KERN_ALERT "\n\nFirst interrupt activated in total:%d\n",
    count_total1);
    printk(KERN_ALERT "Second interrupt activated in total:%d\n",
    count_total2);
    printk(KERN_ALERT "Third interrupt activated in total:%d\n\n",
    count_total3);
    printk(KERN_ALERT "______________________________");
    count1=0;
    count2=0;
    count3=0;
    /* Print the time difference in seconds and nanoseconds */
  xen_irq_lateeoi(irq4,0);
  return IRQ_HANDLED;
}

```

- **The function executed when the module is installed in the system:**

```
    static int __init testmodule_init(void)
{
  struct evtchn_alloc_unbound evtchn_alloc1,evtchn_alloc2,evtchn_alloc3,
    evtchn_alloc4;
  int rc,rc1,rc2,rc3,rc4;
  char *str = "go";
  struct device_node *np = of_find_compatible_node(NULL,NULL,"xen,shared-
    memory-v1");
  struct resource r;

  if (np==NULL)
  {
    return -ENODEV;
  }
  rc= of_address_to_resource(np,0,&r);

  if(rc<0)
  {
    return -EINVAL;
  }
  shared_mem=phys_to_virt(r.start);
  if(!shared_mem)
  {
    return -EINVAL;
  }

  evtchn_alloc1.dom=DOMID_SELF;
  evtchn_alloc1.remote_dom=2;
  rc1= HYPERVISOR_event_channel_op(EVTCHNOP_alloc_unbound, &evtchn_alloc1);
  if (rc1==0)
  {
    printk(KERN_ALERT "EVENT CHANNEL 1 CREATED\n");
  }
  else
  {
    printk(KERN_ALERT "EVENT CHANNEL NOTCREATED\n");
    return rc1;
  }
  rc1 = bind_evtchn_to_irqhandler_lateeoi(evtchn_alloc1.port, evtchn_handler1
    , 0, "Domain-0_1", NULL);
  evtchn_alloc2.dom=DOMID_SELF;
  evtchn_alloc2.remote_dom=2;
  rc2= HYPERVISOR_event_channel_op(EVTCHNOP_alloc_unbound, &evtchn_alloc2);
  if (rc2==0)
  {
    printk(KERN_ALERT "EVENT CHANNEL 2 CREATED\n");
  }
  else
  {
    printk(KERN_ALERT "EVENT CHANNEL NOTCREATED\n");
    return rc2;
  }
  rc2 = bind_evtchn_to_irqhandler_lateeoi(evtchn_alloc2.port, evtchn_handler2
    , 0, "Domain-0_2", NULL);
  evtchn_alloc3.dom=DOMID_SELF;
  evtchn_alloc3.remote_dom=2;
  rc3= HYPERVISOR_event_channel_op(EVTCHNOP_alloc_unbound, &evtchn_alloc3);
  if (rc3==0)
  {
    printk(KERN_ALERT "EVENT CHANNEL 3 CREATED\n");
  }
  else
  {
    printk(KERN_ALERT "EVENT CHANNEL NOTCREATED\n");
    return rc3;
  }
  rc3 = bind_evtchn_to_irqhandler_lateeoi(evtchn_alloc3.port, evtchn_handler3
    , 0, "Domain-0_3", NULL);
  evtchn_alloc4.dom=DOMID_SELF;
  evtchn_alloc4.remote_dom=2;
  rc4= HYPERVISOR_event_channel_op(EVTCHNOP_alloc_unbound, &evtchn_alloc4);
  if (rc4==0)
  {
    printk(KERN_ALERT "EVENT CHANNEL 4 CREATED\n");
  }
  else
  {
    printk(KERN_ALERT "EVENT CHANNEL NOTCREATED\n");
    return rc4;
  }
```

```
  rc4 = bind_evtchn_to_irqhandler_lateeoi(evtchn_alloc4.port, evtchn_handler4
    , 0, "Domain-0_4", NULL);
  irq1=rc1;
  irq2=rc2;
  irq3=rc3;
  irq4=rc4;
  memcpy(shared_mem+4, &evtchn_alloc1.port, sizeof(evtchn_alloc1.port));
  mb();
  memcpy(shared_mem+8, &evtchn_alloc2.port, sizeof(evtchn_alloc2.port));
  mb();
  memcpy(shared_mem+12, &evtchn_alloc3.port, sizeof(evtchn_alloc3.port));
  mb();
  memcpy(shared_mem+40, &evtchn_alloc4.port, sizeof(evtchn_alloc4.port));
  mb();
  memcpy(shared_mem, str, 3);
  return 0;
}


```

- **The function executed when the module is uninstalled:**

```
    static void __exit testmodule_exit(void)
{
  printk(KERN_ALERT "Goodbye module world.\n");
}
module_init(testmodule_init);
module_exit(testmodule_exit);

```

File used for the Dom2 (sender) kernel module: This file is composed by:

- **The libraries used:**

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/of_address.h>
#include <linux/of_irq.h>
#include <linux/kthread.h>
#include <linux/delay.h>
#include <xen/events.h>
#include <xen/xenbus.h>
#include <xen/xen.h>
#include <linux/jiffies.h>

```

- **Module information:**

```
MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR
    ("Xilinx Inc.");
MODULE_DESCRIPTION
    ("testmodule - loadable module template generated by petalinux-create -t
    modules");

```

- **The threads used for testing, functions and other structures:**

```
    static char *shared_mem;
static int irq;
static struct evtchn_send send1,send2,send3,send4;
static struct task_struct *task;
static int num_interrupts=0;
static int interrupts1=0;
static int interrupts2=0;
static int interrupts3=0;
/*Execute task1 with X sec between notification*/
void task1(int time)
{
  unsigned long start_time_new = jiffies;  // get the current time
  unsigned long end_time_new = start_time_new + 20*HZ;  // 60 seconds later
  while (time_before(jiffies, end_time_new)) {
      HYPERVISOR_event_channel_op(EVTCHNOP_send, &send1);
      interrupts1++;
    msleep(time);
  }
  HYPERVISOR_event_channel_op(EVTCHNOP_send, &send4);
  printk(KERN_ALERT "NOTIFIED %d times \n",interrupts1);
  interrupts1=0;
}

/*Execute task2 with X sec between notification*/
void task2(int time)
{
  int count=1;
  unsigned long start_time_new = jiffies;  // get the current time
  unsigned long end_time_new = start_time_new + 20*HZ;  // 60 seconds later
  while (time_before(jiffies, end_time_new)) {
      memcpy(shared_mem+16, &count, sizeof(count));
    HYPERVISOR_event_channel_op(EVTCHNOP_send, &send2);
    count++;
    interrupts2++;
    msleep(time);
  }
  HYPERVISOR_event_channel_op(EVTCHNOP_send, &send4);
  printk(KERN_ALERT "NOTIFIED %d times \n",interrupts2);
  interrupts2=0;
}

```

```
/*Execute task3 with X sec between notification*/
void task3(int time)
{
  int random_numbers[4];  // Array to hold four random numbers
  unsigned long start_time_new = jiffies;  // get the current time
  unsigned long end_time_new = start_time_new + 20*HZ;  // 60 seconds later
  while (time_before(jiffies, end_time_new)) {
       // Generate 4 random numbers
    for (int i=0; i<4; ++i) {
      get_random_bytes(&random_numbers[i], sizeof(int));
    }
    // Copy the 4 random numbers to the shared memory
    memcpy(shared_mem+20, &random_numbers, sizeof(random_numbers));
    HYPERVISOR_event_channel_op(EVTCHNOP_send, &send3);
    interrupts3++;
    msleep(time);
  }
  HYPERVISOR_event_channel_op(EVTCHNOP_send, &send4);
  printk(KERN_ALERT "NOTIFIED %d times \n",interrupts3);
  interrupts3=0;
}


static int thread_function1(void *data)
{
  printk(KERN_ALERT "....EXECUTING THE FIRST TEST....\n");
  printk(KERN_ALERT "NOTIFYING EVERY 1 SEC\n");
  task1(1000);
  printk(KERN_ALERT "NOTIFYING EVERY 0.1 SEC\n");
  task1(100);
  printk(KERN_ALERT "NOTIFYING EVERY 0.01 SEC\n");
  task1(10);
  printk(KERN_ALERT "NOTIFYING EVERY 0.001 SEC\n");
  task1(1);
  printk(KERN_ALERT "....ENDING THE FIRST TEST....\n");
    return 0;
}

static int thread_function2(void *data)
{
  printk(KERN_ALERT "....EXECUTING THE SECOND TEST....\n");
  printk(KERN_ALERT "NOTIFYING EVERY 1 SEC\n");
  task2(1000);
  printk(KERN_ALERT "NOTIFYING EVERY 0.1 SEC\n");
  task2(100);
  printk(KERN_ALERT "NOTIFYING EVERY 0.01 SEC\n");
  task2(10);
  printk(KERN_ALERT "NOTIFYING EVERY 0.001 SEC\n");
  task2(1);
  printk(KERN_ALERT "....ENDING THE SECOND TEST....\n");
    return 0;
}

static int thread_function3(void *data)
{
  printk(KERN_ALERT "....EXECUTING THE THIRD TEST....\n");
  printk(KERN_ALERT "NOTIFYING EVERY 1 SEC\n");
  task3(1000);
  printk(KERN_ALERT "NOTIFYING EVERY 0.1 SEC\n");
  task3(100);
  printk(KERN_ALERT "NOTIFYING EVERY 0.01 SEC\n");
  task3(10);
  printk(KERN_ALERT "NOTIFYING EVERY 0.001 SEC\n");
  task3(1);
  printk(KERN_ALERT "....ENDING THE THIRD TEST....\n");
    return 0;
}

```

- **The function executed when the module is installed in the system:**

```
   static int __init testmodule_init(void)
{
  struct evtchn_bind_interdomain evtchn_alloc1,evtchn_alloc2,evtchn_alloc3,
    evtchn_alloc4;
  int rc,rc1,rc2,rc3,rc4;
  struct device_node *np = of_find_compatible_node(NULL,NULL,"xen,shared-
    memory-v1");
  struct resource r;
  if (np==NULL)
  {
    return -ENODEV;
  }
  rc= of_address_to_resource(np,0,&r);
  if(rc<0)
  {
    return -EINVAL;
  }
  shared_mem=phys_to_virt(r.start);
  if(!shared_mem)
  {
    return -EINVAL;
  }
  while (1)
  {
    if (strcmp(shared_mem,"go")==0)
    {
      break;
    }
  }
  mb();
  memcpy(&evtchn_alloc1.remote_port, shared_mem+4, sizeof(evtchn_alloc1.
    remote_port));
  evtchn_alloc1.remote_dom=1;
  evtchn_alloc1.local_port=0;
  rc1= HYPERVISOR_event_channel_op(EVTCHNOP_bind_interdomain, &evtchn_alloc1)
    ;
  if (rc1==0)
  {
    printk(KERN_ALERT "EVENT CHANNEL CREATED\n");
  }
  else
  {
    printk(KERN_ALERT "EVENT CHANNEL NOTCREATED\n");
    return rc1;
  }
  send1.port = evtchn_alloc1.local_port;
  memcpy(&evtchn_alloc2.remote_port, shared_mem+8, sizeof(evtchn_alloc2.
    remote_port));
  evtchn_alloc2.remote_dom=1;
  evtchn_alloc2.local_port=0;
  rc2= HYPERVISOR_event_channel_op(EVTCHNOP_bind_interdomain, &evtchn_alloc2)
    ;
  if (rc2==0)
  {
    printk(KERN_ALERT "EVENT CHANNEL CREATED\n");
  }
  else
  {
    printk(KERN_ALERT "EVENT CHANNEL NOTCREATED\n");
    return rc2;
  }
  send2.port = evtchn_alloc2.local_port;
  memcpy(&evtchn_alloc3.remote_port, shared_mem+12, sizeof(evtchn_alloc3.
    remote_port));
  evtchn_alloc3.remote_dom=1;
  evtchn_alloc3.local_port=0;
  rc3= HYPERVISOR_event_channel_op(EVTCHNOP_bind_interdomain, &evtchn_alloc3)
    ;
  if (rc3==0)
  {
    printk(KERN_ALERT "EVENT CHANNEL CREATED\n");
  }
  else
  {
    printk(KERN_ALERT "EVENT CHANNEL NOTCREATED\n");
    return rc3;
  }
```

```
  send3.port = evtchn_alloc3.local_port;
  memcpy(&evtchn_alloc4.remote_port, shared_mem+40, sizeof(evtchn_alloc4.
    remote_port));
  evtchn_alloc4.remote_dom=1;
  evtchn_alloc4.local_port=0;
  rc4= HYPERVISOR_event_channel_op(EVTCHNOP_bind_interdomain, &evtchn_alloc4)
    ;
  if (rc4==0)
  {
    printk(KERN_ALERT "EVENT CHANNEL CREATED\n");
  }
  else
  {
    printk(KERN_ALERT "EVENT CHANNEL NOTCREATED\n");
    return rc4;
  }
  send4.port = evtchn_alloc4.local_port;
  task = kthread_run(thread_function1, NULL, "domain1_thread");
    if (IS_ERR(task)) {
        iounmap(shared_mem);
        return PTR_ERR(task);
    }
  msleep(87000);
    printk(KERN_ALERT "....STARTING TEST IN 3....\n");
  msleep(1000);
  printk(KERN_ALERT "....STARTING TEST IN 2....\n");
  msleep(1000);
  printk(KERN_ALERT "....STARTING TEST IN 1....\n");
  msleep(1000);
  task = kthread_run(thread_function2, NULL, "domain1_thread");
    if (IS_ERR(task)) {
        iounmap(shared_mem);
        return PTR_ERR(task);
    }
  msleep(87000);
    printk(KERN_ALERT "....STARTING TEST IN 3....\n");
  msleep(1000);
  printk(KERN_ALERT "....STARTING TEST IN 2....\n");
  msleep(1000);
  printk(KERN_ALERT "....STARTING TEST IN 1....\n");
  msleep(1000);
  printk(KERN_ALERT "....EXECUTING THE THIRD TEST....\n");
  task = kthread_run(thread_function3, NULL, "domain1_thread");
    if (IS_ERR(task)) {
        iounmap(shared_mem);
        return PTR_ERR(task);
    }
  return 0;
}
```

```


```

- **The function executed when the module is uninstalled:**

```
    static void __exit testmodule_exit(void)
{
  printk(KERN_ALERT "Goodbye module world.\n");
}

module_init(testmodule_init);
module_exit(testmodule_exit);

```

Appendix E

Zephyr RTOS build process

This procedure was executed in the following environment:

• Host PC with Linux OS • zephyr-sdk-0.16.1 already installed

The first step is initializing the virtual environment:

```
source ~/zephyrproject/.venv/bin/activate
```

Then, the board xenvm was used to compile the .bin files needed.

```
west build -p always -b xenvm <path to project> -d <path to output directory> -f
```

Appendix F

CPU load files

This was the c file used for the tests executed in the event channel demonstration. This file is composed by:

- **The libraries used:**

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>

```

- **The function used to stress the CPU:**

```c
#define NANOSECONDS_PER_SECOND 1E9
void do_work_for(long nanoseconds) {
    // Get the current time
    struct timespec start_time, current_time;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start_time);

    do {
        // Get the current time
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &current_time);

        // Break if we've worked long enough
        if ((current_time.tv_sec - start_time.tv_sec) *
    NANOSECONDS_PER_SECOND + (current_time.tv_nsec - start_time.tv_nsec) >=
    nanoseconds) {
            break;
        }
    } while (1);
}
void sleep_for(long nanoseconds) {
    struct timespec time;
    time.tv_sec = nanoseconds / NANOSECONDS_PER_SECOND;
    time.tv_nsec = nanoseconds % (long)NANOSECONDS_PER_SECOND;
    nanosleep(&time, NULL);
}

```

- **The main function used:**

```
int main() {
    double target_usage = <desired load>;  // Target CPU usage, e.g., 0.2 for
     20%

    while(1) {
        // Time spent working
        do_work_for(NANOSECONDS_PER_SECOND * target_usage);

        // Time spent sleeping
        sleep_for(NANOSECONDS_PER_SECOND * (1.0 - target_usage));
    }

    return 0;
}

```

Appendix G

Zephyr RTOS dom0less domain file

This file is composed by:

- **The libraries used:**

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>

```

- **The hypercall structure used to print directly into the Xen console:**

```c
#define HYPERVISOR_console_io           18
#define CONSOLEIO_write 0
/* hypercalls */
static inline int64_t xen_hypercall(unsigned long arg0, unsigned long arg1,
                                    unsigned long arg2, unsigned long arg3,
                                    unsigned long hypercall)
{
    register uintptr_t a0 asm("x0") = arg0;
    register uintptr_t a1 asm("x1") = arg1;
    register uintptr_t a2 asm("x2") = arg2;
    register uintptr_t a3 asm("x3") = arg3;
    register uintptr_t nr asm("x16") = hypercall;
    asm volatile("hvc 0xea1\n"
                     : "=r" (a0), "=r"(a1), "=r" (a2), "=r" (a3), "=r" (nr)
                     : "0" (a0),
                       "r" (a1),
                       "r" (a2),
                       "r" (a3),
                       "r" (nr));
    return a0;
}

static inline void xen_console_write(const char *str)
{
    ssize_t len = strlen(str);
    xen_hypercall(CONSOLEIO_write, len, (unsigned long)str, 0,
                  HYPERVISOR_console_io);
}

static inline void xen_printf(const char *fmt, ...)
{
    char buf[128];
    va_list ap;
    char *str = &buf[0];
    memset(buf, 0x0, 128);
    va_start(ap, fmt);
    vsprintf(str, fmt, ap);
    va_end(ap);
    xen_console_write(buf);
}

```

- **The main function used:**

```c
int main()
{
    xen_printf("Hello World\n\r");
    return 0;
}

```

Appendix H

Zephyr RTOS test file

This was the c file used for the Zephyr RTOS build to test the lowest time value between of the event channels.

This file is composed by:

- **The libraries used:**

```c
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>
#include <zephyr/xen/events.h>
#include <string.h>
#include <zephyr/random/rand32.h>

```

- **The threads used for testing, functions and other structures:**

```c
static char *shared_mem = (char *)0x70000000;
static struct evtchn_send send1,send2,send3,send4;
K_THREAD_STACK_DEFINE(my_stack_area, 1024);
struct k_thread my_thread1;
struct k_thread my_thread2;
struct k_thread my_thread3;
int counting1=0;
int counting2=0;
int counting3=0;
int random_numbers[4];
int count=1;
void timer_expiry_function1(struct k_timer *timer_id)
{
    notify_evtchn(send1.port);
    counting1++;
}
void timer_expiry_function2(struct k_timer *timer_id)
{
    memcpy(shared_mem+16, &count, sizeof(count));
        notify_evtchn(send2.port);
        count++;
  counting2++;
}
void timer_expiry_function3(struct k_timer *timer_id)
{
  for (int i=0; i<4; ++i) {
    random_numbers[i]=7;
  }
  memcpy(shared_mem+20, &random_numbers, sizeof(random_numbers));
  notify_evtchn(send3.port);
  counting3++;
}
K_TIMER_DEFINE(my_timer1, timer_expiry_function1, NULL);
K_TIMER_DEFINE(my_timer2, timer_expiry_function2, NULL);
K_TIMER_DEFINE(my_timer3, timer_expiry_function3, NULL);
void task1(int time)
{
    k_timer_start(&my_timer1, K_USEC(time), K_USEC(time));
    /* Wait for 20 seconds */
    k_sleep(K_SECONDS(10));
    /* After 20 seconds, stop the timer */
    k_timer_stop(&my_timer1);
    notify_evtchn(send4.port);
    printk("....Notified %d times....\n",counting1);
}
void task2(int time)
{
    k_timer_start(&my_timer2, K_USEC(time), K_USEC(time));
    /* Wait for 20 seconds */
    k_sleep(K_SECONDS(10));
    /* After 20 seconds, stop the timer */
    k_timer_stop(&my_timer2);
    notify_evtchn(send4.port);
    printk("....Notified %d times....\n",counting2);
}
void task3(int time)
{
    k_timer_start(&my_timer3, K_USEC(time), K_USEC(time));
    /* Wait for 20 seconds */
    k_sleep(K_SECONDS(10));
    /* After 20 seconds, stop the timer */
    k_timer_stop(&my_timer3);
    notify_evtchn(send4.port);
    printk("....Notified %d times....\n",counting3);
}
void thread_function1(void *dummy1, void *dummy2, void *dummy3)
{
  printk("....EXECUTING THE FIRST TEST....\n");
  int time=1000;
  while(1)
  {
          counting1=0;
    task1(time);
    if(counting1<(10/(time*0.000001)))
    {
      printk("Maximum in test1 is %d micro\n", time);
      break;
    }
    else
    {
      time=time*0.1;
      printk("Testing %d micro\n", time);
    }
  }
  printk("....ENDING THE FIRST TEST....\n");
}
void thread_function2(void *dummy1, void *dummy2, void *dummy3)
{
  printk("....EXECUTING THE SECOND TEST....\n");
        int time=1000;
  while(1)
  {
          counting2=0;
    task2(time);

    if(counting2<(10/(time*0.000001)))
    {
      printk("Maximum in test2 is %d micro\n", time);
      break;
    }
    else
    {
      time=time*0.1;
      printk("Testing %d micro\n", time);
    }
  }
  printk("....ENDING THE SECOND TEST....\n");
}
void thread_function3(void *dummy1, void *dummy2, void *dummy3)
{
  printk("....EXECUTING THE THIRD TEST....\n");
  int time=1000;
  while(1)
  {
          counting3=0;
    task3(time);

    if(counting3<(10/(time*0.000001)))
    {
      printk("Maximum in test3 is %d micro\n", time);
      break;
    }
    else
    {
      time=time*0.1;
      printk("Testing %d micro\n", time);
    }
  }
  printk("....ENDING THE THIRD TEST....\n");
}

```

- **The main function used:**

```
    void main(void) {

      uint16_t remote_port;
      uint16_t remote_domid = 1;
    while (1)
  {
    if (strcmp(shared_mem,"go")==0)
    {
      break;
    }
  }
  memcpy(&remote_port, shared_mem + 4, sizeof(remote_port));
  send1.port=bind_interdomain_event_channel(remote_domid, remote_port, NULL,
    NULL);
  if (send1.port!=0)
  {
    printk("EVENT CHANNEL CREATED\n");
  }
  else
  {
    printk("EVENT CHANNEL NOTCREATED\n");
  }
  memcpy(&remote_port, shared_mem + 8, sizeof(remote_port));
 send2.port=bind_interdomain_event_channel(remote_domid, remote_port, NULL,
    NULL);
  if (send2.port!=0)
  {
    printk("EVENT CHANNEL CREATED\n");
  }
  else
  {
    printk("EVENT CHANNEL NOTCREATED\n");
  }
  memcpy(&remote_port, shared_mem + 12, sizeof(remote_port));

  send3.port=bind_interdomain_event_channel(remote_domid, remote_port, NULL,
    NULL);

  if (send3.port!=0)
  {
    printk("EVENT CHANNEL CREATED\n");
  }
  else
  {
    printk("EVENT CHANNEL NOTCREATED\n");
  }
  memcpy(&remote_port, shared_mem + 40, sizeof(remote_port));
send4.port=bind_interdomain_event_channel(remote_domid, remote_port, NULL,
    NULL);
  if (send4.port!=0)
  {
    printk("EVENT CHANNEL CREATED\n");
  }
  else
  {
    printk("EVENT CHANNEL NOTCREATED\n");
  }
  notify_evtchn(send4.port);
  k_sleep(K_SECONDS(10));
  k_thread_create(&my_thread1, my_stack_area,
                  K_THREAD_STACK_SIZEOF(my_stack_area),
                  thread_function1,
                  NULL, NULL, NULL,
                  K_PRIO_PREEMPT(0), 0, K_NO_WAIT);
       k_sleep(K_SECONDS(100));
        k_thread_create(&my_thread2, my_stack_area,
          K_THREAD_STACK_SIZEOF(my_stack_area),
                  thread_function2,
                  NULL, NULL, NULL,
                  K_PRIO_PREEMPT(0), 0, K_NO_WAIT);
       k_sleep(K_SECONDS(100));
        k_thread_create(&my_thread3, my_stack_area,
                  K_THREAD_STACK_SIZEOF(my_stack_area),
                  thread_function3,
                  NULL, NULL, NULL,
                  K_PRIO_PREEMPT(0), 0, K_NO_WAIT);

}

```

Appendix I

Openamp APU and RPU files

This was the filed used for the RPU slave firmware.

This file is composed by:

- **The libraries used:**

```
    #include "xil_printf.h"
#include <stdlib.h>
#include <string.h>
#include <openamp/open_amp.h>
#include "matrix_multiply.h"
#include "platform_info.h"

```

- **The multiplication function used:**

```
#define MAX_SIZE    6
#define NUM_MATRIX    2
#define SHUTDOWN_MSG  0xEF56A55A
#define LPRINTF(fmt, ...) xil_printf("%s():%u " fmt, __func__, __LINE__, ##
    __VA_ARGS__)
#define LPERROR(fmt, ...) LPRINTF("ERROR: " fmt, ##__VA_ARGS__)
typedef struct _matrix {
  unsigned int size;
  unsigned int elements[MAX_SIZE][MAX_SIZE];
} matrix;

static struct rpmsg_endpoint lept;
static int shutdown_req = 0;
static void Matrix_Multiply(const matrix *m, const matrix *n, matrix *r)
{
  unsigned int i, j, k;

  memset(r, 0x0, sizeof(matrix));
  r->size = m->size;

  for (i = 0; i < m->size; ++i) {
    for (j = 0; j < n->size; ++j) {
      for (k = 0; k < r->size; ++k) {
        r->elements[i][j] +=
          m->elements[i][k] * n->elements[k][j];
      }
    }
  }
}

```

- **The callback function executed when the RPU gets notified and when the RPU gets notified with a shutdown:**

```
static int rpmsg_endpoint_cb(struct rpmsg_endpoint *ept, void *data, size_t
    len,
           uint32_t src, void *priv)
{
  matrix matrix_array[NUM_MATRIX];
  matrix matrix_result;

  (void)priv;
  (void)src;

  if ((*(unsigned int *)data) == SHUTDOWN_MSG) {
    ML_INFO("shutdown message is received.\r\n");
    return RPMSG_SUCCESS;
  }

  memcpy(matrix_array, data, len);
  /* Process received data and multiple matrices. */
  Matrix_Multiply(&matrix_array[0], &matrix_array[1], &matrix_result);

  /* Send the result of matrix multiplication back to host. */
  if (rpmsg_send(ept, &matrix_result, sizeof(matrix)) < 0) {
    ML_ERR("rpmsg_send failed\r\n");
  }
  return RPMSG_SUCCESS;
}

static void rpmsg_service_unbind(struct rpmsg_endpoint *ept)
{
  (void)ept;
  ML_ERR("Endpoint is destroyed\r\n");
}


```

- **Main function where the setup is initialized and the RPU waits for notifications:**

```c
int app(struct rpmsg_device *rdev, void *priv)
{
  int ret;
  ret = rpmsg_create_ept(&lept, rdev, RPMSG_SERVICE_NAME,
             RPMSG_ADDR_ANY, RPMSG_ADDR_ANY,
             rpmsg_endpoint_cb,
             rpmsg_service_unbind);
  if (ret) {
    ML_ERR("Failed to create endpoint.\r\n");
    return -1;
  }
  ML_INFO("Waiting for events...\r\n");
  while(1) {
    platform_poll(priv);
    /* we got a shutdown request, exit */
    if (shutdown_req) {
      break;
    }
  }
  rpmsg_destroy_ept(&lept);
  return 0;
}

int main(int argc, char *argv[])
{
  void *platform;
  struct rpmsg_device *rpdev;
  int ret;
  LPRINTF("Starting application...\r\n");
  /* Initialize platform */
  ret = platform_init(argc, argv, &platform);
  if (ret) {
    LPERROR("Failed to initialize platform.\r\n");
    ret = -1;
  } else {
    rpdev = platform_create_rpmsg_vdev(platform, 0,
             VIRTIO_DEV_DEVICE,
             NULL, NULL);
    if (!rpdev) {
      ML_ERR("Failed to create rpmsg virtio device.\r\n");
      ret = -1;
    } else {
      app(rpdev, platform);
      platform_release_rpmsg_vdev(rpdev, platform);
      ret = 0;
    }
  }
  ML_INFO("Stopping application...\r\n");
  platform_cleanup(platform);
  return ret;
}

```

This was the file used for the APU master application:
This file is composed by:

- **The libraries used:**

```
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <time.h>
#include <fcntl.h>
#include <string.h>
#include <linux/rpmsg.h>

```

- **The matrix multiplication functions used:**

```
 #define RPMSG_BUS_SYS "/sys/bus/rpmsg"
#define PR_DBG(fmt, args ...) printf("%s():%u "fmt, __func__, __LINE__, ##
    args)
#define SHUTDOWN_MSG    0xEF56A55A
#define MATRIX_SIZE 6
struct _matrix {
  unsigned int size;
  unsigned int elements[MATRIX_SIZE][MATRIX_SIZE];
};
static void matrix_print(struct _matrix *m)
{
  int i, j;

  /* Generate two random matrices */
  printf(" \r\n Master : Linux : Printing results \r\n");

  for (i = 0; i < m->size; ++i) {
    for (j = 0; j < m->size; ++j)
      printf(" %d ", (unsigned int)m->elements[i][j]);
    printf("\r\n");
  }
}
static void generate_matrices(int num_matrices,
        unsigned int matrix_size, void *p_data)
{
  int i, j, k;
  struct _matrix *p_matrix = p_data;
  time_t  t;
  unsigned long value;
  srand((unsigned) time(&t));
  for (i = 0; i < num_matrices; i++) {
    /* Initialize workload */
    p_matrix[i].size = matrix_size;

    printf(" \r\n Master : Linux : Input matrix %d \r\n", i);
    for (j = 0; j < matrix_size; j++) {
      printf("\r\n");
      for (k = 0; k < matrix_size; k++) {

        value = (rand() & 0x7F);
        value = value % 10;
        p_matrix[i].elements[j][k] = value;
        printf(" %d ",
        (unsigned int)p_matrix[i].elements[j][k]);
      }
    }
    printf("\r\n");
  }

}
static int charfd = -1, fd;
static struct _matrix i_matrix[2];
static struct _matrix r_matrix;
void matrix_mult(int ntimes)
{
  int i;

  for (i=0; i < ntimes; i++){
    generate_matrices(2, MATRIX_SIZE, i_matrix);

    printf("%d: write rpmsg: %lu bytes\n", i, sizeof(i_matrix));
    ssize_t rc = write(fd, i_matrix, sizeof(i_matrix));
    if (rc < 0)
      fprintf(stderr, "write,errno = %ld, %d\n", rc, errno);

    puts("read results");
    do {
      rc = read(fd, &r_matrix, sizeof(r_matrix));
    } while (rc < (int)sizeof(r_matrix));
    matrix_print(&r_matrix);
    printf("End of Matrix multiplication demo Round %d\n", i);
  }
}
```

- **The Remoteproc and RPMsg function used to communicate with the other processor:**

```c
void send_shutdown(int fd)
{
  int sdm = SHUTDOWN_MSG;

  if (write(fd, &sdm, sizeof(int)) < 0)
    perror("write SHUTDOWN_MSG\n");
}
int rpmsg_create_ept(int rpfd, struct rpmsg_endpoint_info *eptinfo)
{
  int ret;

  ret = ioctl(rpfd, RPMSG_CREATE_EPT_IOCTL, eptinfo);
  if (ret)
    perror("Failed to create endpoint.\n");
  return ret;
}
static char *get_rpmsg_ept_dev_name(const char *rpmsg_char_name,
            const char *ept_name,
            char *ept_dev_name)
{
  char sys_rpmsg_ept_name_path[64];
  char svc_name[64];
  char *sys_rpmsg_path = "/sys/class/rpmsg";
  FILE *fp;
  int i;
  int ept_name_len;

  for (i = 0; i < 128; i++) {
    sprintf(sys_rpmsg_ept_name_path, "%s/%s/rpmsg%d/name",
      sys_rpmsg_path, rpmsg_char_name, i);
    printf("checking %s\n", sys_rpmsg_ept_name_path);
    if (access(sys_rpmsg_ept_name_path, F_OK) < 0)
      continue;
    fp = fopen(sys_rpmsg_ept_name_path, "r");
    if (!fp) {
      printf("failed to open %s\n", sys_rpmsg_ept_name_path);
      break;
    }
    fgets(svc_name, sizeof(svc_name), fp);
    fclose(fp);
    printf("svc_name: %s.\n",svc_name);
    ept_name_len = strlen(ept_name);
    if (ept_name_len > sizeof(svc_name))
      ept_name_len = sizeof(svc_name);
    if (!strncmp(svc_name, ept_name, ept_name_len)) {
      sprintf(ept_dev_name, "rpmsg%d", i);
      return ept_dev_name;
    }
  }

  printf("Not able to RPMsg endpoint file for %s:%s.\n",
         rpmsg_char_name, ept_name);
  return NULL;
}
static int bind_rpmsg_chrdev(const char *rpmsg_dev_name)
{
  char fpath[256];
  char *rpmsg_chdrv = "rpmsg_chrdev";
  int fd;
  int ret;
  /* rpmsg dev overrides path */
  sprintf(fpath, "%s/devices/%s/driver_override",
    RPMSG_BUS_SYS, rpmsg_dev_name);
  PR_DBG("open %s\n", fpath);
  fd = open(fpath, O_WRONLY);
  if (fd < 0) {
    fprintf(stderr, "Failed to open %s, %s\n",
      fpath, strerror(errno));
    return -EINVAL;
  }
  ret = write(fd, rpmsg_chdrv, strlen(rpmsg_chdrv) + 1);
  if (ret < 0) {
    fprintf(stderr, "Failed to write %s to %s, %s\n",
      rpmsg_chdrv, fpath, strerror(errno));
    return -EINVAL;
  }
  close(fd);

  /* bind the rpmsg device to rpmsg char driver */
  sprintf(fpath, "%s/drivers/%s/bind", RPMSG_BUS_SYS, rpmsg_chdrv);
  fd = open(fpath, O_WRONLY);
  if (fd < 0) {
    fprintf(stderr, "Failed to open %s, %s\n",
      fpath, strerror(errno));
    return -EINVAL;
  }
  PR_DBG("write %s to %s\n", rpmsg_dev_name, fpath);
  ret = write(fd, rpmsg_dev_name, strlen(rpmsg_dev_name) + 1);
  if (ret < 0) {
    fprintf(stderr, "Failed to write %s to %s, %s\n",
      rpmsg_dev_name, fpath, strerror(errno));
    return -EINVAL;
  }
  close(fd);
  return 0;
}
static int get_rpmsg_chrdev_fd(const char *rpmsg_dev_name,
             char *rpmsg_ctrl_name)
{
  char dpath[2*NAME_MAX];
  DIR *dir;
  struct dirent *ent;
  int fd;

  sprintf(dpath, "%s/devices/%s/rpmsg", RPMSG_BUS_SYS, rpmsg_dev_name);
  PR_DBG("opendir %s\n", dpath);
  dir = opendir(dpath);
  if (dir == NULL) {
    fprintf(stderr, "opendir %s, %s\n", dpath, strerror(errno));
    return -EINVAL;
  }
  while ((ent = readdir(dir)) != NULL) {
    if (!strncmp(ent->d_name, "rpmsg_ctrl", 10)) {
      sprintf(dpath, "/dev/%s", ent->d_name);
      closedir(dir);
      PR_DBG("open %s\n", dpath);
      fd = open(dpath, O_RDWR | O_NONBLOCK);
      if (fd < 0) {
        fprintf(stderr, "open %s, %s\n",
          dpath, strerror(errno));
        return fd;
      }
      sprintf(rpmsg_ctrl_name, "%s", ent->d_name);
      return fd;
    }
  }

  fprintf(stderr, "No rpmsg_ctrl file found in %s\n", dpath);
  closedir(dir);
  return -EINVAL;
}

static void set_src_dst(char *out, struct rpmsg_endpoint_info *pep)
{
  long dst = 0;
  char *lastdot = strrchr(out, '.');

  if (lastdot == NULL)
    return;
  dst = strtol(lastdot + 1, NULL, 10);
  if ((errno == ERANGE && (dst == LONG_MAX || dst == LONG_MIN))
      || (errno != 0 && dst == 0)) {
    return;
  }
  pep->dst = (unsigned int)dst;
}
```

```

/*
 * return the first dirent matching rpmsg-openamp-demo-channel
 * in /sys/bus/rpmsg/devices/ E.g.:
 *  virtio0.rpmsg-openamp-demo-channel.-1.1024
 */
static void lookup_channel(char *out, struct rpmsg_endpoint_info *pep)
{
  char dpath[] = RPMSG_BUS_SYS "/devices";
  struct dirent *ent;
  DIR *dir = opendir(dpath);

  if (dir == NULL) {
    fprintf(stderr, "opendir %s, %s\n", dpath, strerror(errno));
    return;
  }
  while ((ent = readdir(dir)) != NULL) {
    if (strstr(ent->d_name, pep->name)) {
      strncpy(out, ent->d_name, NAME_MAX);
      set_src_dst(out, pep);
      PR_DBG("using dev file: %s\n", out);
      closedir(dir);
      return;
    }
  }
  closedir(dir);
  fprintf(stderr, "No dev file for %s in %s\n", pep->name, dpath);
}
```

- **The main function used:**

```
int main(int argc, char *argv[])
{
  int ntimes = 1;
  int opt, ret;
  char rpmsg_dev[NAME_MAX] = "virtio0.rpmsg-openamp-demo-channel.-1.0";
  char rpmsg_char_name[16];
  char fpath[2*NAME_MAX];
  struct rpmsg_endpoint_info eptinfo = {
    .name = "rpmsg-openamp-demo-channel", .src = 0, .dst = 0
  };
  char ept_dev_name[16];
  char ept_dev_path[32];

  printf("\r\n Matrix multiplication demo start \r\n");

  /* Load rpmsg_char driver */
  printf("\r\nMaster>probe rpmsg_char\r\n");
  ret = system("set -x; lsmod; modprobe rpmsg_char");
  if (ret < 0) {
    perror("Failed to load rpmsg_char driver.\n");
    return -EINVAL;
  }

  lookup_channel(rpmsg_dev, &eptinfo);

  while ((opt = getopt(argc, argv, "d:n:s:e:")) != -1) {
    switch (opt) {
    case 'd':
      strncpy(rpmsg_dev, optarg, sizeof(rpmsg_dev));
      break;
    case 'n':
      ntimes = atoi(optarg);
      break;
    case 's':
      eptinfo.src = atoi(optarg);
      break;
    case 'e':
      eptinfo.dst = atoi(optarg);
      break;
    default:
      printf("getopt return unsupported option: -%c\n",opt);
      break;
    }
  }

  sprintf(fpath, RPMSG_BUS_SYS "/devices/%s", rpmsg_dev);
  if (access(fpath, F_OK)) {
    fprintf(stderr, "access(%s): %s\n", fpath, strerror(errno));
    return -EINVAL;
  }
  ret = bind_rpmsg_chrdev(rpmsg_dev);
  if (ret < 0)
    return ret;
  charfd = get_rpmsg_chrdev_fd(rpmsg_dev, rpmsg_char_name);
  if (charfd < 0)
    return charfd;

  /* Create endpoint from rpmsg char driver */
  PR_DBG("rpmsg_create_ept: %s[src=%#x,dst=%#x]\n",
    eptinfo.name, eptinfo.src, eptinfo.dst);
  ret = rpmsg_create_ept(charfd, &eptinfo);
  if (ret) {
    fprintf(stderr, "rpmsg_create_ept %s\n", strerror(errno));
    return -EINVAL;
  }
  if (!get_rpmsg_ept_dev_name(rpmsg_char_name, eptinfo.name,
          ept_dev_name))
    return -EINVAL;
  sprintf(ept_dev_path, "/dev/%s", ept_dev_name);

  printf("open %s\n", ept_dev_path);
  fd = open(ept_dev_path, O_RDWR | O_NONBLOCK);
  if (fd < 0) {
    perror(ept_dev_path);
    close(charfd);
    return -1;
  }

  PR_DBG("matrix_mult(%d)\n", ntimes);
  matrix_mult(ntimes);

  send_shutdown(fd);
  close(fd);
  if (charfd >= 0)
    close(charfd);

  printf("\r\n Quitting application .. \r\n");
  printf(" Matrix multiply application end \r\n");

  return 0;
}

```

Appendix J

Iperf3 test results

These are the iperf3 TCP test results with the various connections.

Explanation of Metrics:

- **Transfer:** This indicates the amount of data transferred during the test. It is measured in Bytes or GBytes (Gigabytes) and shows how much data was sent or received within the specified time interval (10 seconds in this case).
- **Bitrate:** The bitrate measures the average data transfer rate over the network. It represents how many bits (or Mbits, Gbits, etc.) of data were transferred per second. Higher bitrates generally indicate better network performance.
- **Retransmissions:** This metric shows the number of retransmissions that occurred during the test. In the provided results, the value is consistently 0, which means no data packets needed to be retransmitted due to errors or congestion.

```

#Linux native to PC
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  1.10 GBytes   943 Mbits/sec    0             sender
[  5]   0.00-10.04  sec  1.10 GBytes   938 Mbits/sec                  receiver

#Dom0 to PC
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.02  sec   660 MBytes   552 Mbits/sec    0             sender
[  5]   0.00-10.03  sec   660 MBytes   552 Mbits/sec                  receiver

#Dom1 to Dom0
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  1.21 GBytes  1.04 Gbits/sec    0             sender
[  5]   0.00-10.02  sec  1.20 GBytes  1.03 Gbits/sec                  receiver

#Dom1 to Dom0 to PC
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   620 MBytes   510 Mbits/sec    0             sender
[  5]   0.00-10.03  sec   620 MBytes   510 Mbits/sec                  receiver

#Dom1 to Dom2
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  1.11 GBytes   955 Mbits/sec    0             sender
[  5]   0.00-10.00  sec  1.11 GBytes   952 Mbits/sec                  receiver
```

These are the iperf3 UDP test results with the various connections.
Explanation of Metrics:

- **Transfer:** This indicates the amount of data transferred during the test. It is measured in Bytes or GBytes (Gigabytes) and shows how much data was sent or received within the specified time interval (10 seconds in this case).
- **Bitrate:** The bitrate measures the average data transfer rate over the network. It represents how many bits (or Mbits, Gbits, etc.) of data were transferred per second. Higher bitrates generally indicate better network performance.
- **Jitter:** Jitter measures the variation in the delay of received data packets. It indicates the irregularity in the time it takes for packets to reach the receiver. Lower jitter values suggest a more stable network with consistent packet delivery.
- **Lost/Total Datagrams (Packets):** Lost/Total Datagrams (Packets): This metric shows the number of lost data packets compared to the total number of packets transmitted. It is usually expressed as a percentage. A lower loss percentage indicates a more reliable network.

```

#Linux native to PC
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total
    Datagrams
[  5]   0.00-10.00  sec  1.25 MBytes  1.05 Mbits/sec  0.000 ms  0/906 (0%)  sender
[  5]   0.00-10.04  sec  1.25 MBytes  1.05 Mbits/sec  0.010 ms  0/906 (0%)
    receiver

#Dom0 to PC
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total
    Datagrams
[  5]   0.00-10.01  sec  1.25 MBytes  1.05 Mbits/sec  0.000 ms  0/906 (0%)  sender
[  5]   0.00-10.01  sec  1.25 MBytes  1.05 Mbits/sec  0.138 ms  0/906 (0%)
    receiver

#Dom1 to Dom0
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total
    Datagrams
[  5]   0.00-10.00  sec  1.25 MBytes  1.05 Mbits/sec  0.000 ms  0/906 (0%)  sender
[  5]   0.00-10.01  sec  1.25 MBytes  1.05 Mbits/sec  0.003 ms  0/906 (0%)
    receiver

```

```
#Dom1 to Dom0 to PC
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total
    Datagrams
[  5]   0.00-10.00  sec  1.25 MBytes  1.05 Mbits/sec  0.000 ms  0/906 (0%)  sender
[  5]   0.00-10.04  sec  1.25 MBytes  1.05 Mbits/sec  0.192 ms  0/906 (0%)
    receiver

#Dom1 to Dom2
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total
    Datagrams
[  5]   0.00-10.00  sec  1.25 MBytes  1.05 Mbits/sec  0.000 ms  0/906 (0%)  sender
[  5]   0.00-10.00  sec  1.25 MBytes  1.05 Mbits/sec  0.005 ms  0/906 (0%)
    receiver
```