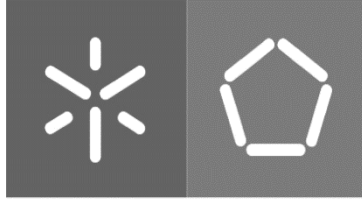




Universidade do Minho
Escola de Engenharia

José Nuno Leite da Silva

**V-Visor: lightweight
virtualization for
RISC-V**



Universidade do Minho
Escola de Engenharia

José Nuno Leite da Silva

**V-Visor: lightweight
virtualization for
RISC-V**

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Sandro Emanuel Salgado Pinto

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.



Atribuição-NãoComercial-Compartilha Igual
CC BY-NC-SA

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Acknowledgements

The development of this thesis would not be possible without a couple of people whom I could not be more grateful for.

Firstly, I would like to thank Dr. Sandro Pinto and Eng. José Martins for providing me with such an interesting thesis and for all the help and support given. They were the foundation for this house and made me push myself towards this end goal. I got motivated by their perseverance which was a great help for me throughout this magnificent journey.

Secondly, I would like to give a huge thanks to Dr. Adriano Tavares who was my teacher at the Embedded System's specialization. He instilled me his knowledge as well as pushing me to have more soft skills and a think-before-you-do approach to every problem that comes along the way.

To my friends and colleagues of the Embedded System Reseach Group (ESRG), my friends, that by any reason stayed with me throughout these five years, while knowing that it is not enough but it is all I can give you right now: a huge thanks for all the support. I would like to specify the ones that were there from the beginning of the first year and pushed me to thrive even when I did not believe in myself: Cristiano, Daniel, Ivo, Valter, and Zeca.

Last but not least, a huge thanks to my family who always believe in me and helped me in all they could while keeping me sane. I hope to make you proud.

'I would hate to die and never do the thing I was born to do.'

TD Jakes

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

V-Visor: virtualização leve para RISC-V

Virtualização para sistemas embebidos é uma tecnologia que permite a consolidação de funções e, ao mesmo tempo, reduz o tamanho do sistema, custo e consumo. Para além disso, melhora a segurança ao usar isolamento e redundância de software.

Aliada ao crescimento de uma gama de processadores gerada a partir de um Instruction Set Architecture (ISA) de Reduced Instruction Set Computer (RISC)-V que é atualmente usado por empresas como Nvidia, Google e Qualcomm, uma solução que proporcione um ambiente seguro e virtualizado para um Operating System (OS) correr é um dos objetivos desta dissertação. Sendo o RISC-V um ISA de código aberto e livre de direitos de autor, a transição entre uso educativo para comercial tem sido vertiginosa.

O objetivo principal desta tese é implementar, de raiz, um *hypervisor* do tipo-1, em modo máquina, para um processador de RISC-V, semelhante ao Lightweight TrustZone-assisted Hypervisor (LTZVisor) que foi implementado num processador Advanced RISC Machine (ARM), que permita a utilização do processador por diferentes OSs convidados enquanto proporciona isolamento temporal e espacial. Esta tese foi desenvolvida utilizando um gerador de processador RISC-V chamado Rocket-chip e implementado na plataforma Zybo.

Sendo V-Visor um hipervisor desenvolvido para a arquitetura RISC-V, contém características de virtualização completa e paravirtualização derivadas de restrições impostas pelo próprio processador. Utiliza o mecanismo de Physical Memory Protection (PMP) para fornecer isolamento espacial a cada OS enquanto, ao mesmo tempo, utiliza o algoritmo Round Robin (RR) para fornecer isolamento temporal o que faz com que o sistema apresente um comportamento determinístico.

Palavras-chave: *hypervisor*, isolamento, RISC-V, separação, virtualização.

Abstract

V-Visor: lightweight virtualization for RISC-V

Virtualization is a settled technology for embedded systems that allows to consolidate functions while reducing system size, cost, and power. It also enhances security and safety by using isolation and software redundancy.

Allied with an emerging core generated from the RISC-V ISA which is already deployed by companies like Nvidia, Google, and Qualcomm, a solution to provide a virtualized, safe and secure environment for an OS to run on top of is a goal of this thesis. Being RISC-V an open-source and royalty-free ISA, the transaction between academic-only to commercial use has been vertiginous.

The main goal of this thesis is to implement, from scratch, a machine-mode type-1 hypervisor for a RISC-V core similar to the LTZVisor which is deployed on top of an ARM core, to enable the usage of the processor by different OSs while having the feeling that all of them execute at the same time and provide temporal and spatial isolation. This project is developed using a RISC-V core generated using the Rocket-chip and deployed on a Zybo board.

Being V-Visor a lightweight hypervisor developed for the RISC-V architecture, it has features from both full and paravirtualization derived from the constraints introduced by the core itself. It takes advantage of the PMP mechanism to provide spatial isolation for each guest while using a RR scheduling algorithm to implement temporal isolation while having a deterministic behavior.

Keywords: hypervisor, isolation, RISC-V, separation, virtualization.

Contents

List of Figures	xi
List of Tables	xii
List of Listings	xiii
Acronyms	xiv
1 Introduction	1
1.1 Motivation and Contextualization	1
1.2 Goals	3
1.3 Document Structure	4
2 Context and State of the Art	5
2.1 Virtualization	5
2.1.1 Virtual Machine Monitor	7
2.1.2 VM encapsulation	9
2.1.3 Virtualization techniques	11
2.1.4 Architecture types	17
2.1.5 Real-time considerations	18
2.1.6 Memory Protection	19
2.2 Emdebbed Hypervisors	20
2.2.1 KVM	20
2.2.2 Minos	21
2.2.3 Mentor	22
2.2.4 Conclusion	22
3 Background	23
3.1 RISC-V	23
3.1.1 PLIC	26
3.2 LTZVisor	27

3.2.1	CPU virtualization	28
3.2.2	Scheduler	29
3.2.3	Memory Partition	29
3.2.4	MMU and Cache management	29
3.2.5	Device partition	30
3.2.6	Interrupt management	31
3.2.7	Time Management	31
4	V-Visor	32
4.1	System Design	32
4.1.1	Goals	32
4.1.2	Requirements	34
4.1.3	Zybo	35
4.1.4	RISC-V Core Generator	36
4.2	Overview	37
4.3	Implementation Considerations	38
4.3.1	Comparison to the LTZVisor	38
4.3.2	Timer	40
4.3.3	Memory Protection	41
4.3.4	Virtualization approach	41
4.4	Startup	41
4.5	Exception Handler	42
4.5.1	PLIC	46
4.6	Memory Protection	46
4.7	Guest Management	49
4.7.1	Guest context save and restore	51
4.7.2	Exceptions delegation to guest	51
4.7.3	Guest Exception Handler	52
4.7.4	Guest switch	54
5	Evaluation and Results	56
5.1	Memory Footprint	56
5.2	Context-Switch Overhead	57
5.3	Interrupt latency	58
6	Conclusion	61
6.1	Future Work	62
	References	63

List of Figures

2.1	VM overview.	6
2.2	Type-1 hypervisor general architecture.	8
2.3	Time isolation.	10
2.4	Spatial isolation.	11
2.5	Trap and emulate.	12
2.6	Privilege Levels.	13
2.7	Exception trigger example.	14
2.8	Full-virtualization overview.	15
2.9	Para virtualization overview.	15
2.10	Memory translation.	16
2.11	Differences between delegation approaches.	17
2.12	Hypervisors general architectures.	18
2.13	Memory map of an RTOS system.	20
2.14	KVM Hypervisor Architecture [DN14].	21
2.15	Mentor overviews.	22
3.1	PLIC overview.	26
3.2	LTZVisor General Architecture [PPG ⁺ 17].	28
3.3	LTZVisor Memory Partition.	30
3.4	Time management levels.	31
4.1	Zybo board.	35
4.2	Zybo SoC overview [Zyb17]	36
4.3	Communication between code and rocket.	37
4.4	System Overview.	38
4.5	PMP configuration register format.	47
4.6	PMP address register format.	47
4.7	PMP configuration CSR layout.	47
4.8	Memory translation and protection	49
4.9	Registers save at the beginning of an exception.	53

4.10 Guest switch process.	55
------------------------------------	----

List of Tables

3.1	RISC-V standard extensions [Qui19].	24
3.2	RISC-V GPRs.	25
3.3	RISC-V Floating-point registers.	26
3.4	RISC-V CSR example for different privilege levels.	26
4.1	Different core generators.	37
4.2	RISC-V exceptions table.	43
4.3	PMP regions according to configuration.	47
5.1	Memory footprint comparison.	56
5.2	Context-switch overhead times.	58
5.3	V-Visor interrupt latency.	59

List of Listings

4.1	Startup.	41
4.2	Guest information structure	42
4.3	Enabling interrupt.	44
4.4	Exception processing.	45
4.5	PMP trap processing algorithm.	48
4.6	Guest information structure	49
4.7	Context Restore Algorithm.	51
4.8	Timer interrupt scheduling algorithm.	53
4.9	Guest information structure	54
4.10	Guest information structure	55

Acronyms

ACTLR	Auxiliary Control Register
AMBA	Advanced Microcontroller Bus Architecture
APU	Application Processing Unity
ARM	Advanced RISC Machine
CPU	Central Processing Unity
CSR	Control Status Register
ESRG	Embedded System Reseach Group
FIQ	Fast Interrupt Request
FPGA	Field-programmable gate array
FVP	Fix Virtual Platform
GP	Global Pointer
GPOS	General Purpose Operating System
GPR	General Purpose Register
HAL	Hardware Abstraction Layer
IBM	International Business Machines
IC	Integrated Circuit
IDE	Integrated Development Environment
IP	Intellectual Property
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
KVM	Kernel-based Virtual Machine
LR	Linker Register
LTZVisor	Lightweight TrustZone-assisted Hypervisor
MAC	Mandatory Access Control
mie	Machine Interrupt Enable
MIO	Multiplexed I/O
mip	Machine Interrupt Pending
MMU	Memory Management Unity

OS	Operating System
PC	Program Counter
PL	Programmable Logic
PLIC	Platform Level Interrupt Controller
PMP	Physical Memory Protection
PS	Processing System
PUM	Protect User Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RR	Round Robin
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SBI	Software Binary Interface
SCR	Secure Configuration Register
SCTLR	System Control Register
SELinux	Security-Enhanced Linux
SoC	System-on-Chip
SP	Stack Pointer
SPSR	Saved Program Status Register
sVirt	Secure Virtualization
TP	Thread Pointer
TTC	Triple Time Counter
TZASC	TrustZone Address Space Controller
VM	Virtual Machine
VMCB	Virtual Machine Control Block
VMM	Virtual Machine Monitor

1. Introduction

The main goal of this thesis is to develop a lightweight hypervisor for the RISC-V architecture - V-Visor - and it is being developed at the ESRG, part of the ALGORITMI center of University of Minho and intends to be a system similar to the LTZVisor developed in this department as well. The following chapter describes the motivation, contextualization, goals, and the document structure for this thesis.

1.1 Motivation and Contextualization

Embedded systems remote to 1960 when Charles Stark Draper developed a Integrated Circuit (IC) to reduce the size and weight of the Apollo Command and Lunar modules [Rou18]. At the time, embedded systems were used to perform simple and unique tasks without the need to communicate with each other. Nowadays, however, the complexity of developing software for said systems has exponentially increased to enable the interaction of subsystems that are running at different levels of critically.

With the current pace of industry progress, economic factors become critical for a company to thrive. The reduction of costs due to labor or outsourcing makes for a favorable environment for the company to prosper. As far as embedded systems go, the commercialization of a processor architecture implies increased costs due to the author's rights so, RISC-V becomes a viable solution to this problem by being royalty-free as well as flexible on the development process which means that it can be used in systems that will be commercialized.

Being RISC-V an open-standard, royalty-free ISA, the adoption of both academia and industry has been vertiginous. It means anyone can design a processor to implement the ISA without paying any license fees since developers are free to safeguard Intellectual Property (IP) and commercialize it. One of the main hurdles to overcome for a new processor architecture to settle in the market is the fast-paced development practices since it brings a demand for processors to be stable and have a long market life. Furthermore, they must have support such as development tools, software libraries, OSs and more. This was all accomplished by the RISC-V ISA which led to its acceptance across the design industry. Another reason that led to the prosperity of RISC-V is that the ISA has been designed to be stable which implies viability while still being customizable to any particular application. To supplement the ISA's specifications, standard extensions are at disposal. Developers might implement them or not according to the needs

of their application. Companies like Nvidia and Western Digital have already embraced RISC-V in the development of their processors replacing proprietary designs.

A lot of modern embedded systems can be an example of how virtualization is needed and brings benefits. A phone that has a touch screen, camera, games, communications, supports multiple wireless connections... can be a great example of the evolution of virtualization on embedded systems since all the subsystems work as a cohesive unit on top of the same hardware.

Virtualization's purpose used to be time-sharing, testing and exploring new OS architectures as well being able to keep backward compatibility of legacy OSs [KK12]. Virtualization makes multiple Virtual Machine (VM)s able to run on top of the same hardware by the usage of a VM monitor also known as a hypervisor which is a layer of abstraction from the underlying processor core, memory, and peripherals that gives a guest the feeling of running directly on top of real hardware [All11]. The upsides to embedded virtualization are:

- The ability to run multiple OSs on the same hardware reducing the costs of buying several hardware platforms, while guaranteeing isolation, bringing security to the system [Hei08];
- It gives a security layer since a VM encapsulates a subsystem that has limited interference with other subsystems providing it with spatial isolation meaning that one application can not use the resources of another application.

However, there are also some downsides to virtualization since the larger the scale of the embedded system, the more problems it can bring to the table. When virtualizing a system, some concerns emerge:

- The fact that a lot of VMs are running on top of the same hardware platform induces a equation: more guests equal more code. With that in mind, it is assumable that it is impossible to get a bulletproof system since there is a great complexity level which means that it is expected to have a lot of bugs. If the subject is critical embedded systems, the existence of bugs can become a massive hazard since failure is not allowed. One possible way to minimize those effects is to have software that can detect faults and make a recovery from them. However, if the fault is relative to a large number of components, it might be impossible to recover [Hei08];
- It also becomes a very heavyweight system since each VM needs its own OS which leads to higher memory usage and consequently more power consumption [Hei08];
- Having a lot of VMs also implies that the integration of all of them becomes extremely complex. In cases where the scheduler has a priority policy, there is the possibility of a VM never having processor time [Hei08].

In order to mitigate these factors, when developing such systems, a few requirements must be fulfilled in order to ensure the system can contain faults at the VM level meaning the remainder VMs will never be affected by a fault in another guest.

- **Data Isolation:** a VM can never access another VM's data in order to keep a layer of separation between them. That means that at any point the guest's memory is safe from any other undue access.
- **Resource Share:** in some cases, there are peripherals that can be accessed by two or more guests. In this case, when the guest running changes, all the necessary data relevant to the peripheral must be saved and restored when that guest resumes ensuring that it continues from where he left off.
- **Temporal Isolation:** the system must be able to run a guest without being dependent on the temporal behavior of any other guest. An application running in a guest can never delay another application from any other guest to run.

So, by analyzing the up and downsides of virtualization, a choice according to metrics and constraints must be made. While in non-critical systems a virtualization approach is, with no doubt, better, when the system must be bulletproof, it is imperial to assure that the fault control mechanism guarantees the VMs encapsulating properties.

1.2 Goals

The main goal of this thesis is to implement, from scratch, a machine-mode type-1 hypervisor for a RISC-V core similar to the LTZVisor[PPG⁺17]. The hypervisor must enable the concurrent execution of several nearly unmodified OSs, while providing temporal and spatial isolation. This project will be developed using a RISC-V core generated with Rocket-chip and deployed on a Zybo board using the programming language assembly for the startup and critical sections of code and C for the development of all the hypervisor's mechanisms.

Due to the complexity of the project, and the existing time constraints, some requirements for the system were set aside and are deferred to future work. As an example, there is no priority for each guest and thus there are no critical nor less important guests in this system. This is mainly to ensure temporal isolation that otherwise would consume too much time to implement. On the other hand, by doing so, there is a guarantee that each guest's memory will be safe since it implements spatial isolation while avoiding hazards to the system. Having these limitations in mind, the steps for the V-Visor's implementation are:

- **Manage system's resources:** Since virtualization allows for multiple OSs to run on top of the same hardware, the hypervisor must act as a resource manager to avoid the loss of isolation and make decisions of what and when a resource must be assigned to each guest.
- **Provide temporal and spatial isolation:** In order to keep the coherence of memory and not disrupt the normal system behavior, temporal and spatial isolation is a must:

- Spatial isolation: A guest can never access memory outside its permitted zone. By this, it is meant that this goal is achieved by assigning a different memory partition for each VM and so, if access to an incorrect memory position is made, the PMP check, explained in chapter 2, will fail and a trap will rise [CP13];
- Temporal isolation: With temporal isolation, it is meant that a system has to be able to run different tasks while ensuring the task's time constraints are fulfilled as well as ensure that the execution of a task is not dependent on another task's time behavior.
- **Support for Multiple VMs:** To enable support for multiple VMs, a scheduling process and a mechanism to save and restore contexts must be a part of the system. The guest OSs must be close to unmodified.

The big picture of this thesis was subdivided into smaller goals that when together form the final system.

1.3 Document Structure

The remaining of this document's structure is: chapter 2 gives an insight into the theoretical concepts used in this thesis from virtualization to Real-Time Operating System (RTOS) considerations. Then it proceeds to give an insight into existing similar systems ending with a brief conclusion to the gathered information. Chapter 3 provides an in-depth analysis to RISC-V and the LTZVisor. Chapter 4 provides a design and implementation analysis as well as illustrations ranging from the hardware choices to the software approach to each goal. Chapter 5 presents the tests and results to which the V-Visor went through with a comparison to the LTZVisor. Finally, chapter 6 grants a conclusion and draws ideas for future work about this project in order to improve it.

2. Context and State of the Art

Throughout this chapter, the theoretical concepts for virtualization, memory protection, exceptions, and embedded hypervisors are exposed. After that, a state of the art analysis is made focusing on type-1 embedded hypervisors.

2.1 Virtualization

Over the last decades, some fundamental changes in how computer services are provided have been made. One of those examples is virtualization which dates to the beginning of the 1960s decade as a solution to divide system resources between different applications on the same computer [M.16, GA16]. In the last decades, some model-changing trends were made. Ranging from the mainframe processing in the 1960s to the personal computers and client/server technology in the 1980s and, now, virtualization [Por12]. The pioneer of mainstream virtualization was developed by International Business Machines (IBM) in the 1960s but Gerald J. Popek and Robert P. Goldberg defined the framework that illustrates the requirements for a computer system to support virtualization. [Por12].

In general terms, creating or using a virtual version of a physical resource is what defines virtualization [M.16]. It can be considered a disruptive technology since it redefines how computers are handled and services are delivered since, with the spread of internet and smartphones usage, online services have been created to allow anyone to buy services, objects, book traveling -which also include virtual tours to get a sense of the destination - and have virtual libraries to save books, music, and movies. All of this through virtual stores [Por12]. Being an increasingly efficient harnessing of existing physical resources - computer hardware resources, storage devices, network resources...- on a software environment, virtualization allows a software environment called VM to run as if on real hardware as shown in figure 2.1. This brings an isolated and efficient copy of the real machine providing huge budget savings in a small amount of time while offering companies new models of application deployment and allow services to be provided in minutes instead of weeks. [Hei08, M.16, Por12]. At its core, virtualization is all about how an application experiences its created environment. In computing, virtualization refers to the abstraction of a physical component into a logical object. A great example is the local area networks or LANs that provide great network, performance and increased manageability due to being apart from the physical hardware [M.16, Por12].

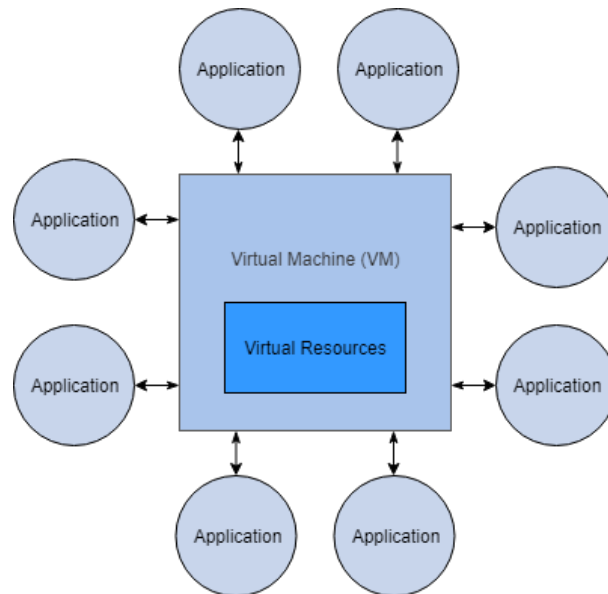


Figure 2.1: VM overview.

Being a VM usually defined as a software program that mirrors the behavior of a separate computer, it is also able to perform a range of tasks such as running applications. They are also defined as the emulation of a resource. By definition, an emulator is a system that allows one computer to mirror another computer system being the emulated system called a guest system and the system being emulated called a host system [Por12]. The OS and applications within the VM are called guests. There are two types of VMs: process VMs and system VMs. A process VM is a virtual platform that executes a single process and its only function is to support the process. This means that it is created and terminated alongside the process. An example of this is a Java VM. On the other hand, a system VM provides a complete and persistent system environment that supports an OS with permission to access virtual and real hardware resources [M.16, Hei08].

Virtualization has, however, limits that are highly relevant for embedded systems. The complexity of software development is growing by the day which makes it harder to achieve reliability. Currently, it is impossible to get a bulletproof system and so, bugs are expected. Although the security can encapsulate threats of a subsystem, this can downgrade the system's experience to the user so, a need for a mechanism that can detect and recover from faults emerges since a fault must be contained in a small component of the system [Hei08]. A guest that needs its own OS makes it heavyweight, leading to performance issues and consequently, increased memory size and power consumption. Hardware resources sharing must be done with strict control. A good example of this is a video display. Since more than one guest can have access to it, there is a need to divide the display into sub-displays in order to avoid overlap between them.

According to Popek and Goldberg's 1974 paper [PG74] classical virtualization can be defined through three essential principles:

- **Equivalence or Fidelity:** A program running on a virtual environment must have the same behavior as if it is running on a physical machine;

- **Efficiency or Performance:** The operations must be performed on real hardware resources without the need for the intervention of a virtualization layer - Virtual Machine Monitor (VMM)- and the difference in performance between a VM and a physical equivalent must be little to none;
- **Resource control(Isolation) or Safety:** The virtualization layer must control all the system's hardware resources [PG74, Por12].

All of these characteristics are important in order to make virtualization highly useful. The first one ensures that software running on the physical hardware will run on the VM and vice-versa. The second one ensures that virtualization is practicable since the majority of instructions are executed directly on hardware [POP⁺ 17]. Finally, the third ensures that software can not break out of the VM [Hei08].

2.1.1 Virtual Machine Monitor

In order for guests to share the physical resources of a host system, two points must be taken into account. Firstly, from the guest's perspective, it has to have access to the various hardware resources it needs to run properly and secondly, the OS in the guest should be able, or believe, it accesses physical memory directly [Por12]. That is where the VMM, also known as a hypervisor, steps in. As a technology, hypervisors were introduced a couple of decades ago being its main goal to maximize resource utilization while seamlessly allowing the execution of guest OSs by emulating either a virtual or a subset of the underlying platform [M.16]. A VMM is a layer of software that resides below the VMs and above real hardware and provides access to physical resources to be shared among multiple guest OSs [HAF11, Por12]. If there was no hypervisor, more than one guest could try to simultaneously control the hardware which would lead to chaos and so, the hypervisor acts as a manager from all the VMs [Por12]. Another main point is that the hypervisor not only abstracts the hardware from each VM but also balances the workload since every guest makes constant demands of the various system's resources [Por12]. Having a resource scheduling process ensures that all of the requested resources are serviced reasonably. Some hypervisors have the option to prioritize guests that run important applications. Moving to a virtual infrastructure makes allocating system resources critical when determining the physical hardware configuration [Por12]. Thus, it must be said that the sum of resources that all of the guests, running on the host consume, need to be available on that host and even have extra resources available for the hypervisor itself.

Porting a hypervisor on embedded environments has its fair share of restrictions. While efficiency is the main objective, hypervisors deal with added constraints outside traditional virtualization environments. Memory sharing is extremely critical as it is the most important constraint on embedded devices. Small memory sizes lead to a need for extremely small and efficient hypervisors. This also results in an easier validation of code to reduce bugs and thus becoming a more secure hypervisor [MTS⁺]. The smaller the hypervisor, the more reliable and secure the platform can be. This is due to the fact that the hypervisor is the system's portion that runs in the most privileged mode which acts as a trusted computing base (TCB) and leads to a more secure platform [M.16]. A way to ensure efficient and small hypervisors is by using

microkernel and separation kernel-based designs. This leads to a small memory footprint depending on the efficiency of the implementation and the functionality level provided. An embedded hypervisor has to fulfill constraints of embedded systems and RTOSs [M.16] for example:

- **Minimum interrupt latency:** The time between an interrupt occurs and the Central Processing Unity (CPU) executing the interrupt handler must be minimal. So, the handler can not stall the normal execution flow for long periods which happens when the system does not support nested interrupts.
- **Preemptive task scheduling:** A higher priority task can stop a lower priority task in order to run at any given time. So, an RTOS kernel must support preemptive task scheduling while keeping the time spent on this action as small as possible [Wan17].

The general architecture of a type-1 hypervisor is shown in figure 2.2 where the hypervisor runs in the most privileged mode on top of the hardware, the VMs run as the second most privileged mode and applications of each VM run in the least privileged mode. Later in this chapter, an analysis of the other architecture types is made [HRL⁺08a]. Generally, microprocessors implement several levels or rings of privilege: one for hypervisor, one for OSs and another for user-space applications. This means that to provide virtualization and execution of simultaneous OSs in different VMs, the hypervisor must support those execution levels [HAF11].

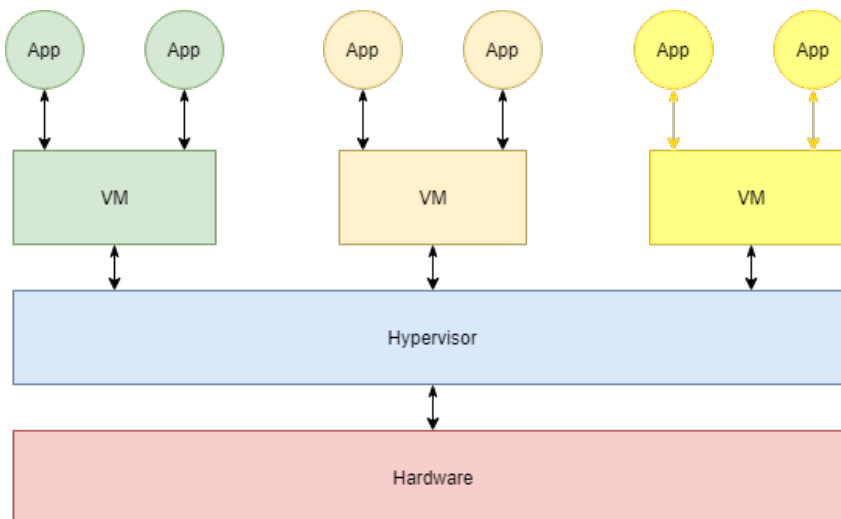


Figure 2.2: Type-1 hypervisor general architecture.

Since the hypervisor is responsible for all the system's resources, the CPU is required to provide a Memory Management Unity (MMU). The ability to run multiple RTOS and applications on the same hardware platform was possible due to embedded hardware performance increase but, sharing resources among them can lead to security and performance issues. By providing isolation between each RTOS or applications, a solution for these problems emerges[RP18].

2.1.2 VM encapsulation

As stated before, a VM behavior and malfunctions should not interfere with other VM's behavior. This concept can be defined by encapsulation. By encapsulating a VM, there is a guarantee that other VMs can continue to run without any problem. There are two types of isolation that are key to this principle: temporal and spatial isolation. Up until this point, a brief analysis of those two principles has been made. In the following subsections, a more in-depth analysis is detailed [PM19].

2.1.2.1 Temporal Isolation

The fact that VMs share hardware resources to enable a resource-efficient architecture leads to security and safety issues since it could lead to a corrupted VM to influence the temporal behavior of another VM which would break the temporal isolation [HRRH14]. Each available resource to a VM is provided by the hypervisor and although the VMs are not aware, only one guest can access a resource at a time. With that in mind, the scheduler must follow the criteria to enable a temporal separation between VMs. According to [HRRH14]:

1. **Predictable:** the scheduling must be deterministic. This means that it should be trivial to predict its behavior which would lead to a real-time analysis;
2. **Interface utilization:** virtualized resources should be idle for as little time as possible and context switches should be minimized;
3. **Hardware overhead:** A practical solution should be achieved in a resource-efficient way;
4. **Added latency:** The scheduling algorithm should induce minimal latency to the response time.

Round Robin

RR scheduling satisfies many of the aforementioned requirements. Being simplistic, its implementation allows for a deterministic system [HRRH14]. While a weighted RR would lead to increased complexity and need of configurability of weights, a simple RR states that requests are issued in turns from different VMs and this scheme is repeated in cycles over time as illustrated in figure 2.3. RR is a preemptive scheduling algorithm where the CPU executes a VM for a specific time and leaves the process even if their execution is not over. It is one of the simplest scheduling technique that utilizes the principle of time slices [Cha17] and it has the advantage of being highly predictable which means that, at any point, the time of switching guests can be determined being truly independent of its own behavior and the behavior of other VMs [HRRH14, PAR14]. Therefore, it leads to strong temporal isolation, through which timing properties hold true even under attack (eg: Denial of Service). When an approach of a non-time-based scheme is used, the scheduling is harder to predict since the usage of virtual interfaces influences the time window in which requests are served [HRRH14].

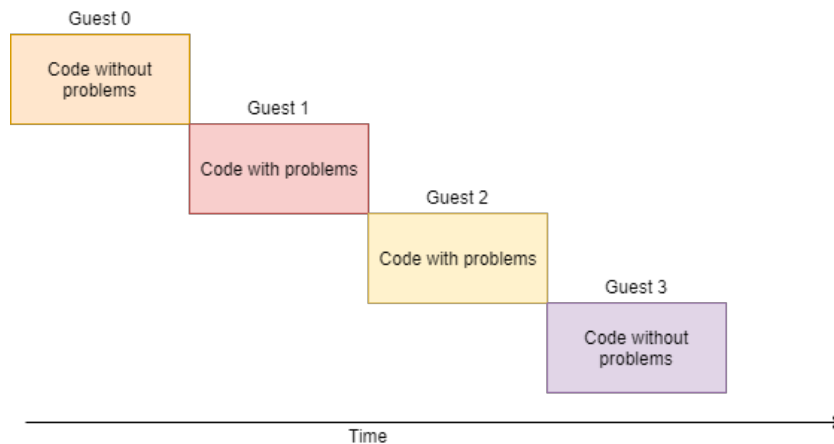


Figure 2.3: Time isolation.

2.1.2.2 Spatial Isolation

In virtualization, spatial isolation is key to ensure that VMs only have access to hardware resources that were assigned to them. A lack of this mechanism might lead to the ability of VMs to manipulate or read data from another VM [HRRH14]. Each VM has a unique address space at system-level ensured by a memory protection mechanism. The distribution of memory resources is crucial for the real-time capability of a VM. While desirable, from a memory utilization perspective, to share memory dynamically, a set amount of memory is necessary to ensure real-time capability [HRRH14]. Each VM has its own local storage which is physically inaccessible from other VMs which leads to absolute hardware-based isolation where even applications inside the VM might not be able to access it [YYL⁺12] and an overview of this approach is illustrated in figure 2.4. The ability to provide spatial isolation brings some advantages to the system. [Fir17]:

1. **Reliability and robustness:** The modularity and isolation provided to VMs improve reliability, robustness and operational availability since the impact of a malfunction can be localized to a single VM and software helps to recover from it;
2. **Safety:** Safety is improved by locating the impact of faults and failures on a single VM. Being isolated, it also makes for attacks to be more complex to have an effect. A faulted VM can be terminated and replaced with a new one that is booted from a known image which enables rapid system recovery following a malfunction.

Although those advantages are key prevention to a full-system meltdown, there are a few downsides as well:

1. **Shared resources:** The ability to share resources comes with a price. For this to properly work properly, the state of peripherals used by multiple guests must be stored and restored when a context switch happens which increases the time slice of a context switch;

2. **Temporal analysis:** The previous topic leads to a second conclusion. With time latency comes the lack of determinism. Since the latency interferes with the temporal behavior of the system, a temporal analysis is harder to make;

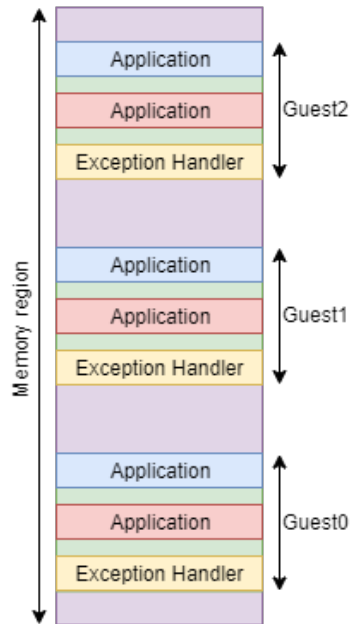


Figure 2.4: Spatial isolation.

2.1.3 Virtualization techniques

With the appearance of new virtualization technology such as isolation container, applications can run in isolated environments with minimal overhead. Due to open interrelation between systems, the has been designed to secure the integrity of the systems from software attacks [BAEA⁺16]. A few reasons are given to how virtualization can be used in practical scenarios:

- **Server consolidation:** consolidate workloads of various less used machines to fewer. If the consolidated servers all run the same OS, the heterogeneity enabled by full-virtualization is not required [HRL⁺08a];
- **Application consolidation:** Since virtualization to the hardware can be provided, an application that requires newer hardware or OS can have its needs fulfilled;
- **Sandboxing:** VMs are useful to provide secure and isolated environments to run different types of applications. Secure containment of malicious applications is a use case of full-virtualization. However, this isolation can and should be provided by the OS [HRL⁺08a];
- **Multiple OSs:** It can provide the feature to run multiple OSs which can run multiple types of applications [SU18].

Embedded hypervisors do not always work on full-virtualization and usually employ paravirtualization when the hardware does not support a particular feature. Hardware emulation at the register level is very rarely seen in embedded hypervisors since it is very complex and slow [M.16]. The two ways of ensuring that code running in a VM does not execute any of those instructions is:

- **Pure virtualization:** sensitive instructions are not executable within the VM (meaning a lower privilege level) and invoke the hypervisor instead;
- **Impure virtualization:** sensitive instructions are removed from the VM and replaced with the hypervisor calls. It is classified as a type of paravirtualization [Hei08].

Furthermore, there are two classes of instructions that must be interpreted At the VM level:

- **Control-sensitive instructions:** These modify the privileged state and thus, interfere with the hypervisor's control over resources;
- **Behavior-sensitive instructions:** These access privileged state and, while they can not change resource allocations, they reveal the real resource's state and, therefore, break the illusion provided by virtualization [Hei08].

Together, they are called virtualization-sensitive instructions. An architecture is classically virtualizable if all sensitive instructions in the ISA are also privileged. When a guest executes a sensitive instruction, the hypervisor must take control for it to take appropriate measures. This technique is used to virtualize the processor. The term virtualizable is used to describe an architecture that can be purely virtualized with a trap-and-emulate approach. This kind of approach states that when a guest OS attempts to execute a privileged instruction the CPU is able to catch it and allow the VMM to emulate the effect that is desired by that VM [HAF11]. Figure 2.5 illustrates this mechanism and figure 2.6 illustrates the privileged level pyramid.

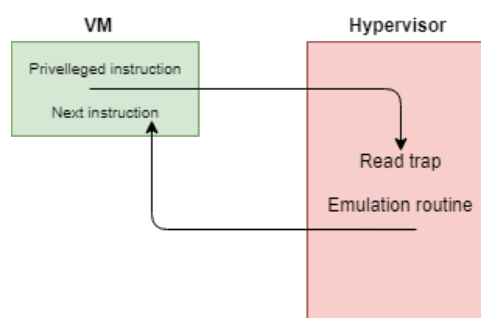


Figure 2.5: Trap and emulate.

Virtualization at the Hardware Abstraction Layer (HAL) takes advantage of the architectures of the guest and host platforms being identical to improve time latency [SU18]. When an emulated machine has the need to use physical resources, the emulator takes over and gives an according response. Although,

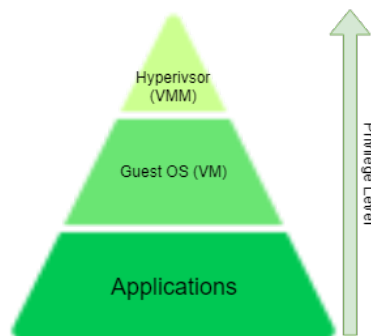


Figure 2.6: Privilege Levels.

for this technique to work, the CPU must be able to trap every privileged instruction and alert the VMM so it can take care of it. This is to avoid collision between guests when trying to use the same resource. Rather than crashing the system, when a trap occurs, it generates a trigger and the instruction is sent to the VMM which allows it to take complete control of the machine and keep the isolation between VMs. After that, the hypervisor can either execute the instruction or emulate the results and return them to the VM [SU18].

There are some virtualization approaches. The main approaches that need attention are full-virtualization and paravirtualization. As a big picture, full-virtualization states that the source code of the guest OS is not modified. In other words, it is not aware that it is being virtualized. Its major benefit is that the guest OSs can still run on a VMM unmodified. The major drawback is that the VMM must use tricks in order to virtualize the hardware for each VM which leads to a bigger overhead when accessing the hardware [BAEA⁺16].

On the other hand, paravirtualization refers to the improvement of performance due to collaboration between the guest OS and the hypervisor. This means that the source code of the guest OS must be modified in order to take advantage of the system calls so the hypervisor executes the privileged instructions. In this case, the guest OS is fully aware that it is being virtualized. The major drawback of this approach is the lack of compatibility and portability of the OSs [BAEA⁺16].

2.1.3.1 Full-Virtualization

Up until this point, an insight into what and how virtualization works has been given. In this subsection, a closer look at full-virtualization will be made stating what it is and what does it imply in its implementation.

Full-virtualization means that the hypervisor provides VMs with an interface similar to the real hardware platform and so, a VM is capable of hosting unmodified legacy software [CSA⁺16]. Being the classical approach, it requires that all sensitive instructions are privileged and execute successfully if the processor is running in the most privileged state but trigger a trap that runs the exception handler when executing

in a lower privileged level, as shown in figure 2.7, which then virtualizes the instruction [LUC⁺19, Hei08]. However, frequent switches between privilege levels imply a high runtime overhead which limits the use of this technology [LUC⁺19]. This approach allows the execution of OSs with their user environment completely untouched in a VM that replicates the interface of real hardware with the exception of temporal behavior [HRL⁺08b]. However, if the instruction set is not virtualizable, this approach can be implemented using partial binary translation [HRL⁺08a].

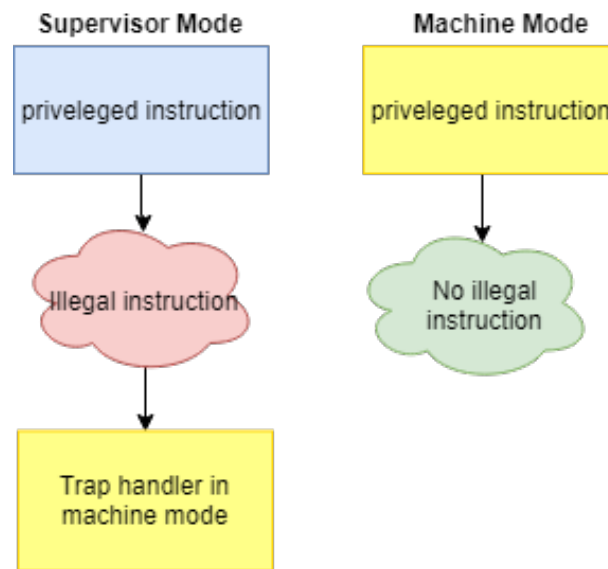


Figure 2.7: Exception trigger example.

A drawback of pure virtualization is that exceptions have a cost. On pipelined processors, an exception breaks the pipeline flow which results in a delay of processing instructions. This happens since exceptions are branches that can not, usually, be predicted by a processor's branch-prediction unit [Hei08]. Also, full-virtualization on low-end hardware must be achieved using mechanisms that were not originally designed for it. In order to be capable of developing full-virtualization, the hardware must fulfill the necessary requirements such as each low privileged level instruction set be a subset of the original, most privileged, instruction set [CSA⁺16]. Although this limits the efficiency, user-mode code of the guest OS can still run unmodified [HRL⁺08a]. To summarize it's main features:

- Guest OSs are unaware of each other;
- Allows guest OSs to be unmodified;
- Hypervisor directly interacts with the hardware;
- The hypervisor allows multiple guests to run on the same hardware platform.

An overview of full-virtualization is illustrated in figure 2.8

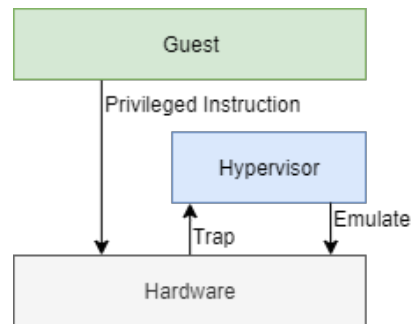


Figure 2.8: Full-virtualization overview.

2.1.3.2 Paravirtualization

To contrast with full-virtualization, a second type emerges: paravirtualization. Paravirtualization is used in order to allow user-mode applications to run inside VMs while OS compatibility is traded off for performance for certain applications running on systems without virtualization hardware assist [HRL⁺08a]. The fact that the virtual platform is not equal to the physical one violates Goldberg’s [PG74] equivalence requirement even if only for the architecture-dependent part of the OS kernel [HRL⁺08a]. To allow resource control, privileged instructions are replaced with system calls to the hypervisor which then acts accordingly (see figure 2.9).

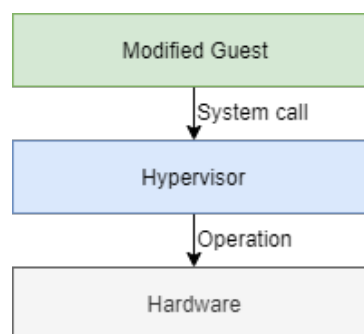


Figure 2.9: Para virtualization overview.

This approach also increases efficiency since all the non-privileged instructions run directly on the physical CPU [HRL⁺08a, VMW06]. Performance benefits are also a key attraction of paravirtualization since a guest OS is aware that they are being virtualized and so, they can be modified to take advantage of that. One downside to the approach is that modified guests can never be migrated back to real physical hardware [VMW06]. To summarize it’s main features:

- Contrasting to full-virtualization, guests are aware that they are being virtualized;
- The hypervisor does not need a high processing power;
- The system works as a cohesive unit.

2.1.3.3 Memory virtualization

Memory virtualization main goal is to map physical memory instructions to machine memory [LSS14]. As so, while a VM might have the feeling of accessing a memory region, it actually translates to a different physical address. An illustration of how this mechanism works is done in figure 2.10.

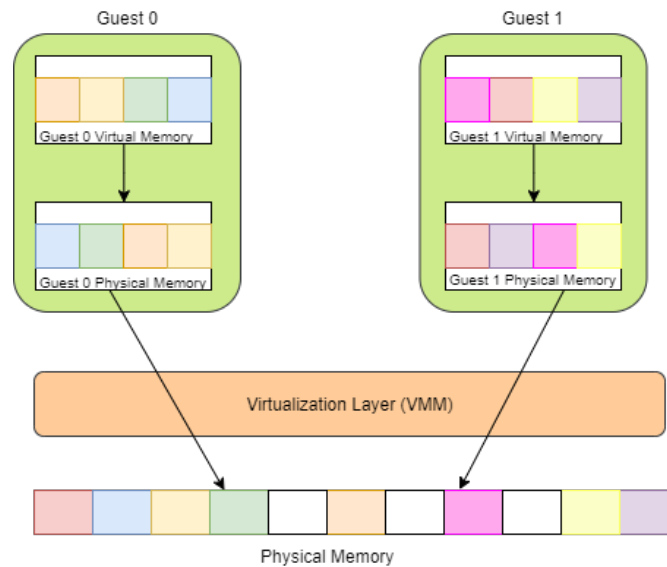


Figure 2.10: Memory translation.

As such, memory virtualization is a two-level address translation mechanism. The first level is the translation from the guest's virtual memory to the guest physical memory and it is done by the guest OS. The second level is the translation from the guest physical memory to real hardware memory which is done by the VMM. This allows for the guest to be compiled to any memory section and so, there is no need to modify the main core of the guest OS.

2.1.3.4 Interrupt virtualization

In order to virtualize interrupts, a couple of concepts have to be explained: interrupt delegation and the usage of Platform Level Interrupt Controller (PLIC). The interrupt delegation mechanism that RISC-V cores offer can be extremely useful to take a workload of the hypervisor. A great example of a use case for the delegation process would be when an application inside a VM tries to access memory it is not allowed to access. In this case, there could be two approaches: allowing the trap to be dealt with by the hypervisor or delegate exceptions caused by user-mode software to the VM to handle. This second approach is more efficient since the application only needs to communicate with the guest OS which then handles the exception without the need to take longer to handle it by having to wait for the hypervisor. The difference between those two approaches is made in figure 2.11

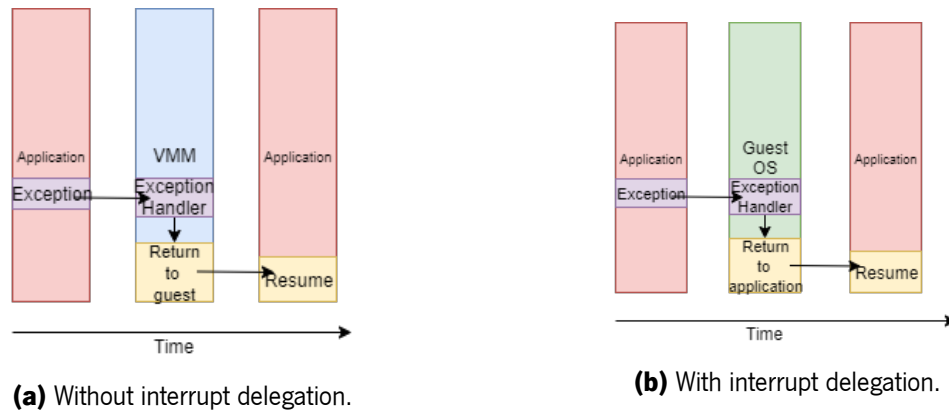


Figure 2.11: Differences between delegation approaches.

2.1.4 Architecture types

Previously in this chapter, it was stated that a more in-depth analysis of the different architecture types of hypervisors would be made. There are three types of hypervisor architectures: type 1, type 2 and type 1.5. While the name does not provide any clues about their differences, the main point of interest is on how they are deployed [Por12, M.16, PG74].

- Type-1:** These types of hypervisors run directly on top of the hardware without an OS beneath it. Due to that fact, they are also referred to as bare-metal hypervisors and so, since it communicates directly with hardware resources, it is much more efficient than a type-2 hypervisor [Por12]. They also manage the guest systems being a thin layer of software that creates an environment in which a guest OS can execute. Since they have minimum performance implications, they are the preferred chosen type for embedded systems [M.16];
- Type-2:** On the other hand, the type-2 hypervisors run on top of an OS which acts as a separation layer between the hardware and the hypervisor and also they are also responsible for abstracting the guest OS from the host OS. So, they are called hosted hypervisors. A great benefit of this type is that it can support a larger range of hardware being inherited from the OS it uses. Most of the time type 2 hypervisors are easier to install and deploy since a lot of the hardware configuration is already done by the underlying OS [M.16, Por12]. However, having that extra layer between the hypervisor and the hardware means they are not as efficient as type 1 hypervisors since they have performance implications as they are routed through the host OS to have access to the hardware. This can be explained with a memory read example. When a guest tries to perform a disk read, it creates a request to the hypervisor that will hand the request to the OS which then handles it, gives the information back to the hypervisor and then back to the guest which implies more time and processing overhead [M.16, Por12]. They are also less reliable since anything that affects the availability of the underlying OS will impact the hypervisor and consequently the guests it supports. A reboot of the OS would require a force reboot of all the VMs on that host [Por12];

- **Type-1.5:** These types of hypervisors are a mix of both worlds from type-1 and type-2. It has a part of the hypervisor running in bare-metal while a privileged VM runs hosted extensions.

In figure 2.12 there is an illustration of the three different architecture types where the differences can be enlightened.

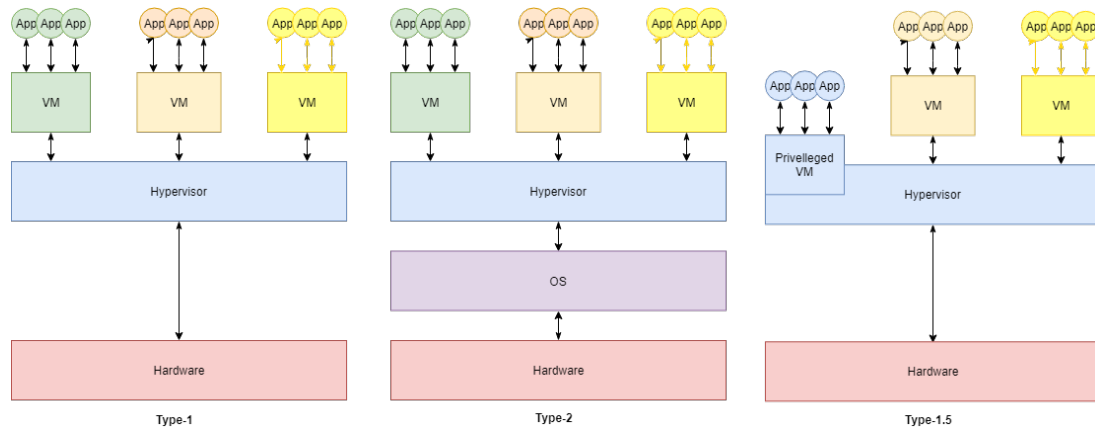


Figure 2.12: Hypervisors general architectures.

In most situations, embedded hypervisors are considered type-1. Having described them previously, a few conclusions about the basic requirements of embedded hypervisors can be made:

- Small size and sleek implementation;
- Support for independent execution (isolation) but interactive coexistence between applications;
- Low latency between system components;
- Minimum effect on system resources and fast response time [M.16].

2.1.5 Real-time considerations

With the exponential growth of embedded systems, safety and security aspects become highly relevant [KL18]. Some of the real-time considerations relative to this thesis were already done in previous sections. In this section, however, some that were left out will be explained.

Embedded systems often have strict timing and performance constraints so, it must be addressed that real-time issues and violations of real-time constraints in embedded systems are the hardest errors to detect since they are extremely sensitive to the external events stimulating the system and the temporal behavior of the system itself [GZ12, ACEP02].

The appropriate behavior of real-time applications depends on the correctness of the computations results as well as the time these results take to be produced [ACEP02] and so, the development of real-time systems requires a reduction in development costs while guaranteeing that temporal constraints are met [ACEP02]. An application running in a VM must provide consistent timing results and the CPU must

be configured to perform consistently [CW05]. An application must be adapted to suit the requirements and constraints of the VM so it must be designed having in mind the creation of clear, logical and easily implemented designs [AZ87]. Task management and inter-component task interactions have to be implemented in a way that ensures temporal predictability and determinism as well as support fault-tolerance by active replication [Kop11]. The scheduling design must be concerned with the allocation of all the system's resources that a task needs so it fulfills the timing requirements [Kop11].

The main properties of an RTOS are modularity, flexibility, composability, and dependability [GCL16]. Modularity is the ability to divide the system into well-defined communication pieces. Composability is a property that allows a composite system to have a set of functionalities from the functionalities of subsystems [GCL16]. First, for time-sensitive software components design, virtualization adds a set of issues that need to be addressed. The methodologies need to correctly account for the virtualization overhead on the execution of real-time tasks especially when it uses virtualized peripherals [CAA08]. Secondly, when OSs are running in parallel, the time measured in a VM might be discontinuous. If a timer is virtualized its resolution might be affected by the scheduler. To conclude, although out-of-scope of this thesis, multi-core platforms trigger even more problems to the real-time virtualization since written software components for high-performance concurrent machines might not run as expected [CAA08]. Since the VMs are managed by a hypervisor, while being a black box to it, modularity is enhanced but can become a weakness for real-time embedded systems.

2.1.6 Memory Protection

Memory protection is achieved through the usage of the PMP mechanism of the RISC-V architecture. This mechanism allows for the most privileged code to indicate in which memory range of addresses lower privileged code can access. It prevents inadequate access to peripherals and memory zones of other guest OSs. As stated before, the physical memory space must be equal or larger than the sum of memory used by each guest since there can not be a virtualization of a nonexistent resource. Still, memory zones can be shared among guests. A UART, for example, is mapped in the memory and each guest can be allowed to use it. An illustration of a memory map for an RTOS is done in figure 2.13. This approach grants some benefits:

- The memory of a guest is not accessible by another guest but there might be common memory zones;
- The lowest privileged mode can not disable interrupts and stack overflows can be detected by the PMP.

The PMP has a process table - in ROM - in which it stores the addresses that encode the memory zone the lower privilege level code can use.

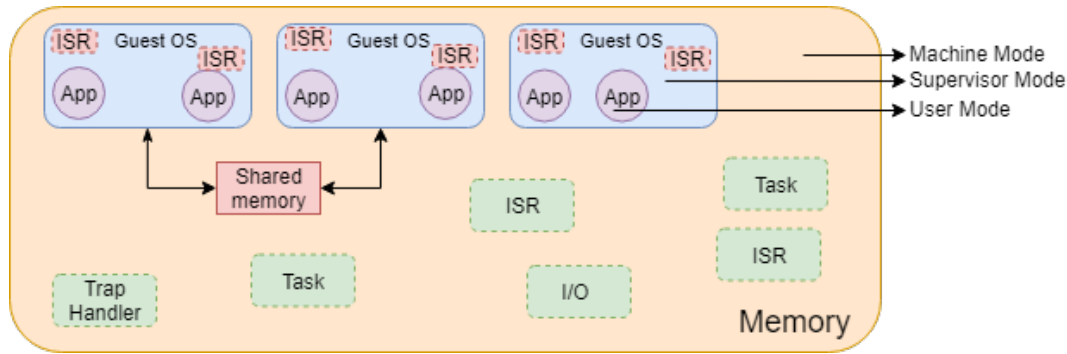


Figure 2.13: Memory map of an RTOS system.

Each time a new guest starts to run, that table must be refilled with according values. The configurations of the PMP mechanism can be enumerated:

- **Off:** In this state, the configuration for the PMP register `-pmpcfg-` is set so that the addresses `-pmpaddr-` are ignored by the PMP;
- **NA4:** This configuration states that the memory the lower privilege can access is the one on the `pmpaddr` register plus 4 bytes;
- **NAPOT:** Similar to NA4, NAPOT is a configuration that allows lower privilege to access the memory zone ranging from the value in `pmpaddr` to the granularity of the system which must be 2^{2+n} ;
- **TOR:** This configuration states that if a `pmpcfgi` is set to TOR, then the range of memory that lower privilege can access has the boundaries of `pmpaddri-1` to `pmpaddri`.

Each of the configurations has, as well, three bits to give permissions to read, write or execute in that region.

2.2 Emdebbed Hypervisors

A few examples of type-1 embedded hypervisors are enumerated and described in order to give the reader some insight into the current state of the art. Each one has its advantages and disadvantages and by the end of the section, a discussion is made in order to give an analysis.

2.2.1 KVM

The Kernel-based Virtual Machine (KVM) is an open-source virtualization technology built into Linux® that allows a host machine to run multiple, isolated VMs being an open-source hypervisor technology for virtualizing compute infrastructure running on x86 compatible hardware. It was announced in 2006 and became a part of the Linux Kernel's mainline a year after. Since it is Linux based, it benefits from every new feature that Linux has without any additional effort. Its kernel integration can provide its guests more fluid access to all the hardware and networking profiles that may exist [Cli19, Wha].

2.2.1.1 KVM/ARM

KVM/ARM implements split-mode virtualization to the hypervisor design in order to run across all the privileged CPU modes. It has ARM's virtualization support enabled by the hypervisor mode while running Linux kernel services in kernel mode. That allows it to be integrated with the kernel without too many modifications to the code. This can be done by splitting the hypervisor into two parts: a lowvisor running in hypervisor mode which provides the context switch while the highvisor runs in kernel mode as a part of the host Linux kernel being responsible for using all the Linux's functionalities like the scheduler. The KVM/ARM hypervisor general architecture is shown in figure 2.14.

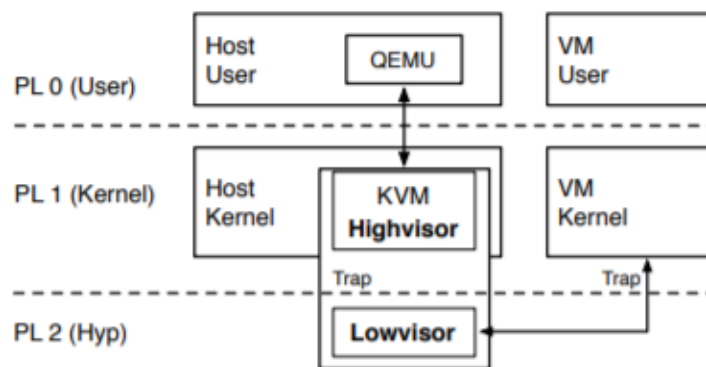


Figure 2.14: KVM Hypervisor Architecture [DN14].

While the lowvisor provides a low-level exception-handler as well as a mechanism to switch worlds, the highvisor handles the more complex faults and provides instruction emulation. An exception to the highvisor while executing a VM will first trigger an exception to the lowvisor running in hypervisor mode which then causes another exception to run the highvisor. That implies a double exception cost [DN14].

2.2.2 Minos

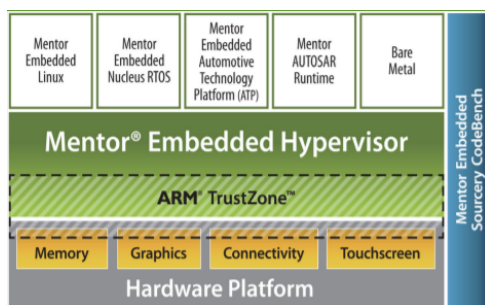
Minos is an open-source type-1 lightweight hypervisor that runs in bare metal. It implements a complete virtualization framework that can run multiple VMs on top of the same hardware providing CPU, interrupts, memory, timer, and some peripherals virtualization [Min]. Being developed for mobile and embedded platforms currently can only support the ARMv8-A architecture. For debugging it supports ARM's official Fix Virtual Platform (FVP) and ARM DS5 tools can be used for simulation and debugging [Min]. To create a VM it uses the a file that creates a device tree node (.dts) and allows VMs to have real hardware permissions. Minos has an application - nvm - running on VM0 to allow the management of the guest VM while nvm provides virtio-based paravirtualization [Min] since there is no need to emulate all the hardware on the host. This approach, however, makes it impossible to run an unmodified guest since it must be aware that is being virtualized to take advantage of hypervisor calls. Paravirtualization is also usually faster than full virtualization [LL12].

2.2.3 Mentor

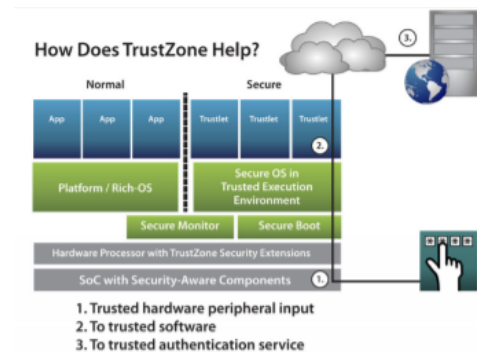
Mentor embedded hypervisor is a type-1 hypervisor with a small footprint that was designed and built specifically for embedded applications. It enables systems to boot quickly while at the same time minimizing the impact on guest OS execution. It is capable of running single or multi-core processor architectures and supports asymmetric multiprocessing, symmetric multiprocessing or a mix of both worlds. Its dynamic scheduling of VMs leads to the balancing of workload and priority-based execution to fulfill real-time and performance constraints. To complement it, Mentor features a device model that supports virtualized and direct device access for critical applications [Men13]. It's main attraction points are:

- Secure by design;
- Reliable and high-performance type-1 hypervisor;
- Gives a choice of the guest OS;

The Mentor system overview is illustrated in figure 2.15a and the secure by design feature is illustrated in figure 2.15b.



(a) Mentor system overview [Men13].



(b) Secure by design overview [Men13].

Figure 2.15: Mentor overviews.

2.2.4 Conclusion

As described during the previous section, there are plenty of solutions with different benefits to implementing a type-1 bare-metal hypervisor. Being RISC-V a core that is emerging, a solution to virtualize it brings up a few questions. Which metrics were chosen before implementing? What type of hardware and software constraints are imposed? How can trade-offs influence the system? These questions are answered in chapter 4 where the design of the V-Visor is presented.

3. Background

In this chapter, an in-depth description of RISC-V architecture and the LTZVisor is done. Since the V-Visor was made from scratch for a different architecture, core and platform, not all the functionalities described about the LTZVisor will be present in the V-Visor. In chapter 4 a comparison between the two will be made.

3.1 RISC-V

The first RISC architecture started to being developed in 1975 at IBM T.J.Watson research center and lasted until the early years of 80s. At the same time, University of California and Stanford University started to developed similar projects [Ok199].

The base of RISC-V is alike the original RISC but is updated with the awareness of new trends and needs of the embedded systems market [DMMF⁺19]. RISC-V is an open-standard ISA which has been experiencing exponentially growing support, being already supported by GCC and the Linux Kernel. It is not a processor but a specification that allows the developer to create its own 32-, 64- or 128-bit processor implementation. RISC-V was developed by the University of California in Berkeley and, at first, was developed to support computer architecture research and education oriented hardware implementations [DMMF⁺19]. An example of previous free ISAs is OpenRISC [DMMF⁺19].

RISC-V is little-endian, which allows straightforward integration with the most popular state-of-the-art embedded infrastructures. Great companies like Google, Qualcomm, Nvidia, and Samsung are members of the RISC-V foundation which is a great example of how RISC-V has come a long way. The major reason for this is that the concern about a monopolistic policy is growing since ISA owners protect their IP and so, they do not allow free available implementations as well as free-market competition for other core designers which, ultimately, leads to the inability to reuse[DMMF⁺19]. The usage of an open-standard and free ISA improves the time to market while leading to lower costs from reuse.

The RISC-V manual is divided into two volumes: user-level ISA - unprivileged specs - and another one describing the privileged architecture with three privilege levels - machine, supervisor and user mode - . Any implementation may employ user mode only, user and machine mode or all the three at the same time. The load and store architecture consists of register-to-register operations being, the reason, the fact that operands reside in General Purpose Register (GPR)s and the result of the operation is also

saved in a GPR [OkI99]. This allows determinism since it separates the memory access layer to the operations. Once a value is available in a register it can execute the instruction avoiding the need for possible multi-cycle accesses to memory while running [OkI99]. The RISC-V ISA has a mandatory core set (RV32I/RV64I/RV128I) - contains the loads, stores, shifts, arithmetic (add and subtract) and logical operations, compares, branches, jumps, system calls, and counters - and a few optional extensions designated by a single letter in the name's acronym (see table 3.1) which are atomic operations (A), compressed instructions (C), single-, double-, or quad-precision floating-point operations (F, D, Q), decimal floating-point operations (L), multiplication and division (M), bit-level manipulation (B), and vector operations (V). There is a designation for the ISA that includes the modules IMAFD that is designated by the letter G.

Extension	Description
M	Instructions that allow for multiplication and division of values held in two integer registers.
A	Instructions that allow for multiplication and division of values held in two integer registers
F, D, Q	Instructions that allow for (F) single-, (D) double-, and (Q) quad-precision floating-point computations. The precision's extension is dependable on the lower-precision extension being present.
G	The combination of the IMAFD extensions is denominated G
V	Instructions to allow vector instructions to the floating-point extensions.
L	Instructions that allow for decimal floating-point operations.
B	Instructions that enable bit-level manipulations.
C	This extensions introduces the concept of compress instructions. While the base integer specification states that instruction words must be 32-bit long and aligned on 32-bit boundaries in memory the C extension provides 16-bit encoding for common operations which allows the CPU to work with alignments on mixed 32- and 16-bit boundaries.

Table 3.1: RISC-V standard extensions [Qui19].

The base ISA of RISC-V has thirty-two GPRs (x0-x31) which have the width of the chosen implementation (32-/64-/128-bit) and are listed in table 3.2. The user-level ISA is defined as a base integer ISA which as stated before must be featured in any implementation but it is restricted to a minimal set of instructions which are sufficient to arrange a reasonable target for compilers, assemblers, linkers and OSs which is a great foundation for more customized processor ISAs.

A key feature of RISC-V is the software support since the community provides a complete software stack that can be handled in embedded applications. For smaller targets like 32-bit microcontrollers, a subset of the base integer can be implemented (E) which has sixteen general purpose registers instead of the usual thirty-two and being the base specification frozen, the ISA stays stable and software written

for RISC-V will have backward compatibility [DMMF⁺19, OCMP19]. RISC-V includes standard and non-standard extensions. While other ISAs are considered a single set that changes for each newer version as instructions are added over time, RISC-V focuses on keeping the base and standard extension the same over time and keep the newer instructions in an optional extension which leads to the increase of software reuse [DMMF⁺19].

Register	Name	Description
x0	zero	Hardwired to 0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporary registers
x8	s0/fp	Saved register/Frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function arguments/Return values
x12-x17	a2-a7	Function arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporary registers

Table 3.2: RISC-V GPRs.

The x0 register is always hardwired to the value zero as it is particularly useful for clearing a register value in register-to-register operations. X1 is the return address register which is used to store the Program Counter (PC) before performing a jump instruction to a different section of code. When that section ends and the return instruction is found, the PC is loaded with the x1 register value. As for x2, it refers to the Stack Pointer (SP). This value is used as a reference for the compiler to save and restore registers during the code translation from C to assembly. The register x3 is the global pointer. This register has the value of the address of the start of the data segment plus 0x800. This is because the RISC-V immediate values are 12-bit signed which is +/- 0x800 and so, this way the global pointer relative access can reach their full extent. X4 is the thread pointer used to hold the address to the beginning of a thread data.

X5 through x7 are temporary registers of the processor used to hold data during an arithmetic and logic operations. The same goes for x28-x31. For x8, saved registers are used to hold values that should be preserved across calls having x9 and x18-27 the same function although x8 can also be a frame pointer which is a pointer to the start of the stack frame that stays static during a subroutine call.

As far as function parameters go, a range from x10-x17 registers can be used to send values from a routine to another function. X10 also works as a return value for the called function.

Following the same logic, if the floating-point extensions are used, another thirty-two floating-point registers are brought into the table (f0-f31) as illustrated in table 3.3.

Register	Name	Description
f0-f7	ft0-ft7	Floating-point temporary registers
f8-f9	fs0-fs1	Floating-point saved registers
f10-f11	fa0-fa1	Floating-point arguments/Floating-point return values
f12-f17	fa2-fa7	Floating-point arguments
f18-f27	fs2-fs11	Floating-point saved registers
f28-f31	ft8-ft11	Floating-point temporary registers

Table 3.3: RISC-V Floating-point registers.

Besides those GPRs, RISC-V has a PC, and Control Status Register (CSR)s for each privilege level as shown in table 3.4.

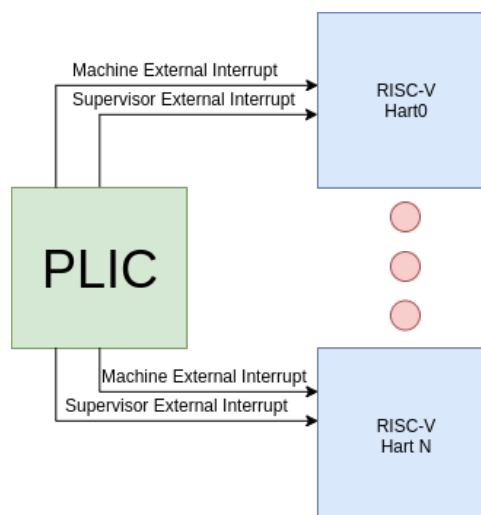
Machine CSR	Supervisor CSR	User CSR
Machine trap vector (MTVEC)	Supervisor trap vector (STVEC)	User trap vector (UTVEC)

Table 3.4: RISC-V CSR example for different privilege levels.

When it comes to exceptions, there are two types: synchronous and asynchronous. The synchronous are referred to as traps and are exceptions caused by an instruction while asynchronous are referred to as interrupts and, as the name suggests, are caused by an interrupt of the processor by an event.

3.1.1 PLIC

As for PLIC, this interrupt controller connects global interrupt sources to interrupt targets (see figure 3.1). It contains multiple interrupt gateways (one per interrupt source) as well as a PLIC core that ensures interrupt prioritization and routing [WLA⁺ 16].

**Figure 3.1:** PLIC overview.

Those global interrupts are addressed from their source to an interrupt gateway which processes the interrupt signal from each source and sends a single interrupt request to the PLIC core [WLA⁺ 16]. Each

interrupt source is assigned a different priority and the PLIC core contains an interrupt enable bits matrix to select which interrupts are enabled for each guest. If an interrupt amongst the enabled ones is pending, the core notifies the target, by writing the interrupt number in the claim register, but only if the priority of the interrupt is higher than the set threshold. When a handler goes to process the external interrupt, it sends a request to retrieve the highest priority interrupt to be taken from the PLIC core which then clears the corresponding interrupt source pending bit [WLA⁺16]. After it finishes to process the interrupt, it sends, to the gateway, an interrupt completion message so the interrupt gateway can forward another interrupt request from the same source to the PLIC.

Although the PLIC is a global interrupt controller, each hart has a set of local interrupt sources that are not processed by the PLIC. So, this interrupts can be attended quickly since there is minimal latency between the source of the interrupt and the interrupt handler of the hart. Also, there is no need to know which hart will service the request [WLA⁺16].

By using the PLIC the trap-and-emulate principle previously described can be used. When a guest tries to execute an instruction while using values it should not be able to, the hypervisor has to step in. In the case of using the PLIC, if a guest has permission to modify flags to only one or two interrupts, if it tries to write a value that tries to modify any of the other flags, the hypervisor will use a mask to protect them. As a practical example, if a guest can write on the first and second bits of a byte and still tries to write on the fifth, while the instruction is legal, the hypervisor must stop it from writing in any other bits other than the first and second.

3.2 LTZVisor

LTZVisor is a hypervisor that relies on hardware support while still implementing software privilege levels as well as a secure environment. The approach to this project consisted of three principles [PPG⁺17]s:

- **Minimal implementation:** The software component of a project is the main source of exploits for attacks. Being so, the LTZVisor relies on the hardware support of TrustZone as much as possible [PPG⁺17];
- **Least privilege:** A component must only have access to resources that it requires. This is helped by the fact that by design, TrustZone assures that despite the CPU execution mode, the non-secure world will always be less privileged [PPG⁺17];
- **Asymmetric scheduling:** This principle is based on fulfilling timing requirements. This is done by having the secure environment having a higher privilege level of execution than the non-secure [PPG⁺17].

LTZVisor uses ARM TrustZone to provide temporal and spatial isolation [PAO⁺19] between both worlds and so, virtualizes each CPU into two virtual CPUs being one for the secure world and the other to the

non-secure. Each of these worlds has a list of banked registers and the number of registers that need to be saved and restore in a partition-switch of a guest OS is less since the hardware supports the virtual CPU. Among the unbanked registers, there are some that can only be modified on the secure side in order to guarantee a high level of security [PPG⁺17]. The general architecture of the LTZVisor is represented in figure 3.2.

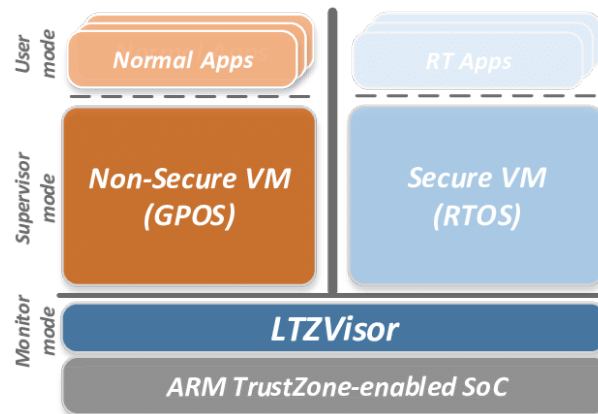


Figure 3.2: LTZVisor General Architecture [PPG⁺17].

3.2.1 CPU virtualization

As stated in the LTZVisor description, the TrustZone technology virtualizes each physical core into two virtual ones: one being used for in the secure world and the other in the non-secure world [MAC⁺17]. To help minimize the number of registers saved and restored during a context switch each of the worlds has its own physical register bank. The Virtual Machine Control Block (VMCB) of the non-secure side has twenty-seven registers: thirteen GPRs (R0-R12), the SP, the Linker Register (LR) and the Saved Program Status Register (SPSR) for each mode: supervisor, system, abort, and undef. GPRs from R8 to R12 as well as the SP, LR, and SPSR of the Fast Interrupt Request (FIQ) and Interrupt Request (IRQ) modes are not included since they are exclusive to each world. The only coprocessor registers that are not banked are the System Control Register (SCTLR) and the Auxiliary Control Register (ACTLR) that need to be preserved.

The VMCB of the secure side only has sixteen registers: thirteen GPRs (R0-R12), the SP, the LR and the SPSR for the system mode. By design, the monitor mode is dedicated to the secure world only. These optimizations lead to a decrease in interrupt latency from the secure guest perspective which increases the speed of the transition from the non-secure to the secure world when a secure interrupt is triggered while running in a non-secure environment.

By design, as far as unbanked registers go, some can only be written from the secure world while still be readable when in a non-secure state. Since those registers are used to change the MMU state, they can only be accessed while running in a secure world. When a non-secure guest boot happens, an attempt to modify those registers will occur to which the hypervisor must give a solution for. A solution for this is

filling some of the non-secure VMCB registers with a specific value so that MMU and level 1 cache of the non-secure world side are enabled before the boot of the General Purpose Operating System (GPOS).

3.2.2 Scheduler

The hierarchical scheduling is a well-known problem in virtualizing a real-time environment and a great example of this problem is when the hypervisor schedules virtual CPUs while a guest RTOS running over the virtual CPU schedules its own tasks. To enable real-time task execution over the executing on top of a virtual CPU requires a complex hierarchical scheduling analysis as well as for both schedulers to be accordingly modeled.

To avoid this problem, the LTZVisor implements an asymmetric scheduler which guarantees that the non-secure guest is only scheduled to run in the idle periods of the secure guest. The secure guest can also preempt the execution of the non-secure one. By having a higher priority scheduling in the secure VM the LTZVisor can have its workload of scheduling taken off and let a secure guest take care of that. This does not compromise the ring of protection however due to the fact that the LTZVisor still runs at a higher privilege mode than the secure guest.

3.2.3 Memory Partition

The traditional hardware-assisted memory virtualization uses the MMU to enable a two-level address translation by mapping the guest virtual addresses to guest physical addresses and finally to host physical addresses. To run unmodified guests and implement isolation between them is a two-stage MMU key feature. However, the TrustZone-enabled System-on-Chip (SoC)s only provide single-level address translation which makes the presence of TrustZone Address Space Controller (TZASC) a major requirement for the final solution [PS19]. This is due to the fact that this component enables partition of memory into segments which ultimately leads to spatial isolation between the non-secure and the secure VMs. While the non-secure VM should have its segment configured as non-secure the remainder partition of memory should be configured as secure. Any attempt for the non-secure guest to access a secure memory space will trigger an exception and the hypervisor steps in. The granularity of the hardware to which the LTZVisor was deployed on can be configured to be 64MB on the Zynq board. A zero or a one in a particular bit of the TZ_DDR_RAM register indicates if the memory partitions are secure or non-secure. Figure 3.3 illustrates the memory map of the LTZVisor. The non-secure VM uses a total of 960MB while the low memory footprint VM and hypervisor use the last available memory which corresponds to 64MB. The remainder of the memory is not used since the hardware platform entitled with 1GB of memory.

3.2.4 MMU and Cache management

The MMU which is TrustZone-aware provides two different MMU interfaces for each world which leads to the ability to have a local set of virtual-to-physical memory address translation tables which means that

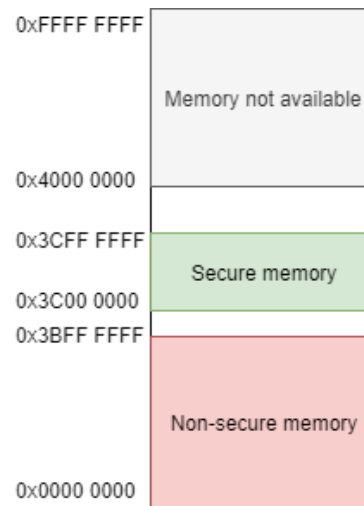


Figure 3.3: LTZVisor Memory Partition.

each world has its own copy of the TTBR register set as well as an independent MMU configuration.

To record the security state of the transaction that accesses memory, an extra tag (NS bit) is used by extending the processor caches. This bit can only be set by hardware and can not be accessed directly by software. With this design, when the system changes between the non-secure and secure worlds, the cache lines do not have the need to be flushed which leads to performance increase since no cache management operation needs to run each time there is a guest-switching operation.

In RISC-V the memory protection mechanism has checks even for cache. If lower privileged software tries to access a cache value that was retrieved from a memory zone it does not have permissions to access, a trap will be triggered.

3.2.5 Device partition

By design, TrustZone technology allows for devices to be configured as secure or non-secure which leads to isolation at the device level [CSFP20]. The LTZVisor adopts a pass-through policy for implementing device virtualization which by definition means that devices are directly managed by the guest partitions. To provide better isolation between both worlds, devices are not shared amongst guests but instead assigned to the respective partitions at design time and posterior configured at the boot. Secure devices are the ones configured for the secure VM while the ones configured for the non-secure VM are non-secure devices. This is done so that a non-secure guest partition trying to access a secure device will trigger an exception for the hypervisor to deal with which means that a non-secure VM can not compromise the state of any device belonging to the secure VM.

3.2.6 Interrupt management

By design, in TrustZone-enable SoCs, the interrupt controller supports the coexistence of both secure and non-secure interrupt sources. While providing configuration of the secure interrupt sources with higher priorities than non-secure, it has configuration models that allow for IRQs and FIQs assignment to secure and non-secure interrupt sources. The LTZVisor configures interrupts of the secure world as FIQs and interrupts from the non-secure world as IRQs. Secure interrupts that are triggered while running the secure VM are redirected to it without the need of the hypervisor to step in which leads to lower interrupt latency of the secure guest. This is done by disabling the FIQ bit in the Secure Configuration Register (SCR). If a non-secure interrupt is triggered while the secure guest is running, it does not affect the secure guest behavior and instead, when the non-secure guest starts to run, the interrupt will then be processed. When a non-secure guest is running and an FIQ is triggered, the execution flow is redirected to the hypervisor which then will handle it in monitor mode which allows for the secure world interrupts to be attended as soon as possible avoiding high interrupt latency overheads. However, if the non-secure VM triggers an IRQ, it will be processed by the non-secure VM itself.

3.2.7 Time Management

Being temporal isolation a key feature in virtualized systems, it can be achieved through two levels of timing: hypervisor level and partition level. For the partition level, it enables guests to have a notion of virtual or real-time. The virtual time refers to the time the guest has been running while the real-time keeps track of all the time that has past even while other guests were running. A comparison between both is illustrated in figure 3.4



Figure 3.4: Time management levels.

Due to its dual-OS configuration as well as the asymmetric scheduling, the hypervisor dedicates one timing unity for each guest. While the secure VM uses the Triple Time Counter (TTC)0, the non-secure VM uses the TTC1 which has to be configured as a non-secure device since otherwise an exception would be triggered on when the non-secure VM tried to access it. This time management allows for each VM to have its timing structures updated under all circumstances.

While the LTZVisor has a time unity for each world, RISC-V provides a single hardware timer. This means that every time management operation must be done using it.

4. V-Visor

In this chapter, the implemented RISC-V hypervisor - V-Visor - is presented. The system's overview, goals, design and implementation is made. This chapter will start by giving an in-depth analysis of the choices that were made before starting to implement this thesis - requirements and hardware choices - and after that, a more detailed description of goals, and finally an in-depth analysis of the implementation is done highlighting the trade-offs that were made in order to complete the main goal.

4.1 System Design

The following section provides an explanation of the system's design enumerating the goals, requirements, implementation considerations, and discussion about the hardware platforms alongside the core generator that are used giving an explanation as to why those choices were made.

4.1.1 Goals

The following subsection provides a more detailed and technical explanation to the implementation's points of interest.

1. **Boot and Startup:** The flow of a startup goes through a few stages:

- When the CPU has a reset, it has a set address it starts executing from;
- After that, the code that is loaded into that section runs. It is needed to set the Global Pointer (GP) as well as the SP and Thread Pointer (TP) which are three registers of the core;
- Following, there is a need to clear the .bss section which stores uninitialized or empty global memory regions. As so, in the startup, all the bytes of this memory region are set to zero;
- To finish, the main function is called in order for the system to run.

Like any system, a mechanism to initialize the GPRs and CSRs is needed. In order to have a working and coherent system, all the GPRs must have know values and those who have a special function must be written during the process. Some CSRs must be written with certain values in order for the system to be able to give a response to those calls.

2. **Exception Handler:** An exception handler must have mechanisms to ensure that any of the registers used in its execution do not overwrite the values they had prior to the exception trigger. A mechanism to store and load the registers before and after each exception occurs is needed. Subdividing this goal into sub-goals, when a trap or interrupt occurs, the normal procedure is:
 - Saving the CPU registers in a known memory location. This is done so when the exception handler ends, the registers can be restored to the value they had prior to entering the handler;
 - Read the CSRs that contains information about the reason behind the trap or interrupt to the registers that are used as function parameters and call the C handler;
 - Inside the C handler, according to the cause of the trap or interrupt, run a block of code to deal with the reason behind the trigger;
 - Update the CSRs accordingly in order to ensure the system gives an according response to the trap or interrupt;
 - Load the CPU registers from the know memory location. This ensures that the application that was running prior and that will be running after keeps its integrity by having the registers with the same values it had before entering the handler.
3. **Context Save and Restore:** In order to keep the integrity of the system, each time there is a trap or interrupt, the system has to give a response. If one of those triggers is not processed the system can start to have unpredictable behavior. One of the crucial steps of a trap or interrupt processing is the assurance of coherence across all the CPU registers. Since an exception handler can and will manipulate those registers, there is a need to store them at a know memory location before processing the reason behind the trigger. When the system has done so, all the registers must be restored.
4. **Changing Privilege Levels:** This system has support for all the three privilege modes - machine, supervisor, user - and so, each of these levels has a different level of access to the system's resources. While the machine mode has access to every resource, the supervisor and user mode are dependent on what the system provides them to use. As stated before, due to complexity, the time each guest has to run is static and so, it is a pure RR meaning that after the boot and configurations, the system starts to run in supervisor mode and only goes back to machine mode when there is an exception that needs to be handled by the hypervisor. In the same way, each guest can have multiple applications running in user mode and each application's resources are configured inside the guest.
5. **Use the PMP as memory protection:** The PMP mechanism allows machine mode to set memory locations with certain privileges - read, write and execute - for lower privileged modes. With that, a guest that runs in supervisor mode can never access a memory location outside its given range.

This also allows a guest to have access to read-only memory in order to avoid the need for machine mode to be running when executing supervisor code. This allows for spatial isolation which is the next goal.

6. **Supporting VMs:** A VM is an abstraction layer for the hardware which allows an application to run in an emulated environment. Being the hypervisor a VM monitor, it runs directly with higher privileges than the guest OS [FW14]. A VM needs, as the hypervisor, a boot, exception handler, and context switch. These are the three critical needs for a simple VM. After that, there is the trap and interrupt delegation so that some of those triggers can be processed in supervisor without the need for the hypervisor to intervene. This allows for the VM to be more independent as well as being quicker at processing traps and interrupts that might arise during the runtime. The main sub-goals for this goal are:

- **Boot and startup:** similar to the hypervisor boot and startup with the main difference being in the initialization of the supervisor CSRs instead of the machine mode ones;
- **Exception handler:** a smaller exception handler since not all the traps and interrupts can be processed in a lower privilege level;
- **Exception delegation:** the hypervisor will delegate exceptions that are possible to process in a lower privilege level in order to reduce the time of processing as well as to enable VMs to deal with each exception accordingly to their implementation;
- **Context save and restore:** equally, the supervisor needs to save its context when going into an exception handler in order to maintain the system's coherence.

4.1.2 Requirements

A project's design is a crucial part of the development cycle. One of the key elements to a good design is the requirements choice since it is the set of rules the system must fulfill in order to run smoothly and efficiently. The requirements for this project are:

- **Reliability:** this is the ability that states the hypervisor must be trustworthy as well as provide good performance;
- **Maintainability:** refers to the ease to maintain a product in order to ensure that faults or what causes them can be repaired, to prevent unexpected execution behaviors - bugs - and to maximize both the system's useful life as well as efficiency;
- **Isolation:** Isolation can be described in two main points: fault and software isolation. While fault isolation guarantees that a VM that went through a bug can not compromise physical components which is accomplished through smart scheduling as well as good resource allocation policies in

the VMM, software isolation is the natural isolation of each VM by running in a separate software environment;

- **Performance:** The principle of "less is more" fits virtualization as a glove. By having small portions of code for each mechanism, the system can have good performance while not compromising the main functional requirements;
- **Low Latency:** To have low latency to exception's response as well as to change context is essential to increase the performance requirement.

4.1.3 Zybo

The Zybo board (figure 4.1) is can be classified as an entry-level platform to embedded software development and it is built around the Xilinx Zynq-7000 family. It is based on the Xilinx All Programmable SoC architecture which integrates a ARM Cortex-A9 processor and Field-programmable gate array (FPGA) logic [Zyb17].

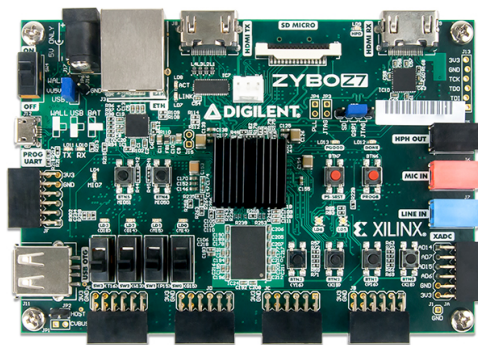


Figure 4.1: Zybo board.

The Zynq SoC architecture is divided into two subsystems: Processing System (PS) and Programmable Logic (PL) [Zyb17]. Figure 4.2 illustrates the architecture overview being the PS green and the PL yellow. The PL has a lot of several dedicated ports and buses that tightly couples it to the PS and must be configured by the processor or via the JTAG port [Zyb17]. It is in the PL that the RISC-V core will be running.

The PS, on the other hand, includes the Application Processing Unity (APU), Advanced Microcontroller Bus Architecture (AMBA), DDR3 memory controller and a number of peripheral controllers that have their inputs and outputs multiplexed to fifty-four dedicated pins (Multiplexed I/O (MIO)) [Zyb17]. Peripheral controllers that do not have their inputs and outputs connect to the MIO might route their I/O through the PL via the extended-MIO. The PL is also connected, as a slave, to the interconnect and, cores implemented in it, can also trigger interrupts to the processors [Zyb17].

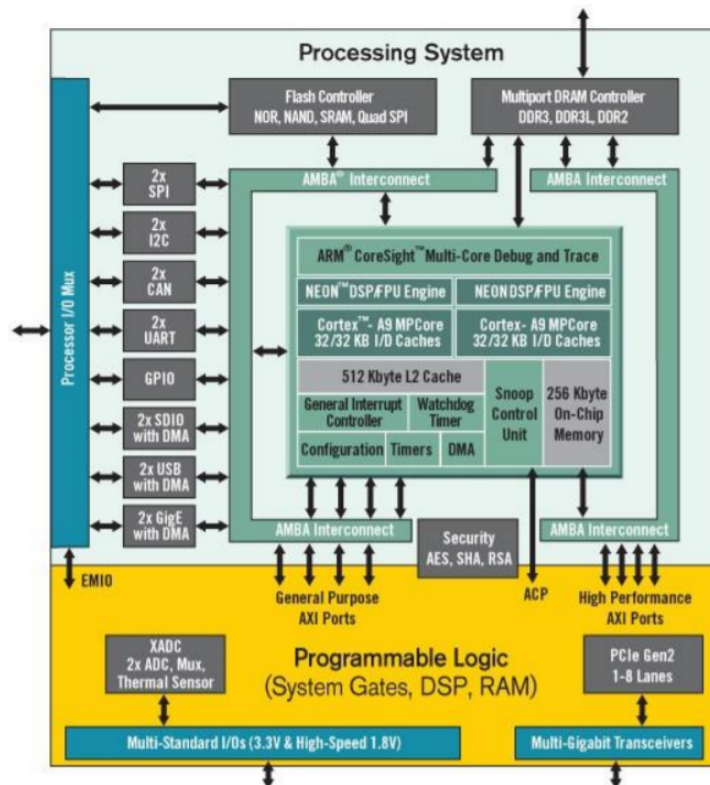


Figure 4.2: Zybo SoC overview [Zyb17]

4.1.4 RISC-V Core Generator

There is a great variety of RISC-V core generators to fulfill each project's intents. Among all of them, for this thesis, the Rocket-chip is the choice. Being a 32/64/128-bit RISC-V core generator, it has a 5-stage in-order pipeline as well as an MMU that supports page-based virtual memory, non-blocking cache and a front-end with branch prediction. As for privilege modes, it supports the three-machine, supervisor, and user- as well as a few ISA extra modules like multiplication and floating-point. Being the deployment board the Zybo a lot of constraints apply. Being one of them memory size, only the standard RV64I ISA plus the compressed instructions extension (C) will be used when generating the core.

4.1.4.1 Core

In order to deploy the hypervisor developed for this thesis, a choice of what core to use had to be made. Using the Zybo as a deployment board, a core that has support for it is a constraint. Amongst the core options listed in table 4.1, the Rocket-chip core generator is the chosen one to get the RISC-V core due to it being the only one that has support for the Zybo. However, due to the architecture of the FPGA the RISC-V will be tethered. This means that there is no direct communication between the RISC-V core and the developer but instead the front-end server, running within the ARM cores on the PL, will act as an intermediate between them (figure 4.3) as it will be illustrated in section 4.2.

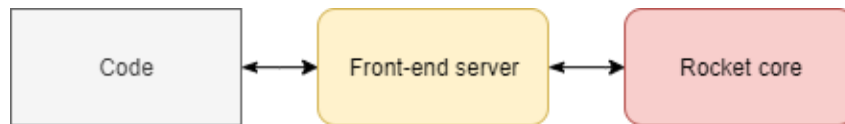


Figure 4.3: Communication between code and rocket.

Although the Rocket-chip can generate a RV64GC core, the Zybo board can only support the RV64IC one due to size constraints. This means that all of the instructions provided by the optional extensions will not be available to use during the implementation phase of the system besides the compressed instructions module.

Core Generator	Privilege Levels	Tethered	Support for Zybo	Possible ISAs
LowRISC	M/S/U	No	No	RV64GC
PicoRV32	M	No	No	RV32IMC
Rocket	M/S/U	Yes	Yes	RV64GC
RI5CY	M/U	No	No	RV32IMCF
BOOM	M/S/U	No	No	RV64GC
Ariane	M/S/U	Yes	No	RV64GC
VexRISCV	M/S/U	No	No	RV32IMCA

Table 4.1: Different core generators.

4.2 Overview

With the previous comparisons and requirements, a system overview can be structured in order to illustrate how everything interconnects. While the hardware core that runs on top of the FPGA is a RISC-V core that communicates with the front-end ARM core, the hypervisor itself runs on machine mode with the highest level of privileges. Inside the hypervisor, there are different mechanisms to support the execution of guest OSs. Each guest OS runs in supervisor mode to guarantee it can never access physical system resources, like the hardware timer, while being able to have full control of all the applications that run on top of them. Finally, applications are the lowest privileged layer of this system running in user mode. Figure 4.4 illustrates the system overview.

The hypervisor takes advantage of the exception handler to be able to process any undue instruction performed by a VM. It is also responsible for memory virtualization and protection as well as for interrupt virtualization. Every time there is a guest switch it is also the hypervisor's duty to guarantee the correct context switch.

Being this project a thesis, a simpler approach was made becoming the metrics of the project: costs, time to conclude and maintainability. For the thesis itself, the metrics of the working system are meeting deadlines, minimum interrupt latency and being reliable. Trade-offs may influence the system since all the

decisions made have an impact ranging from choosing the core generator and ISA extensions to software approaches of lower-level privileged interrupts.

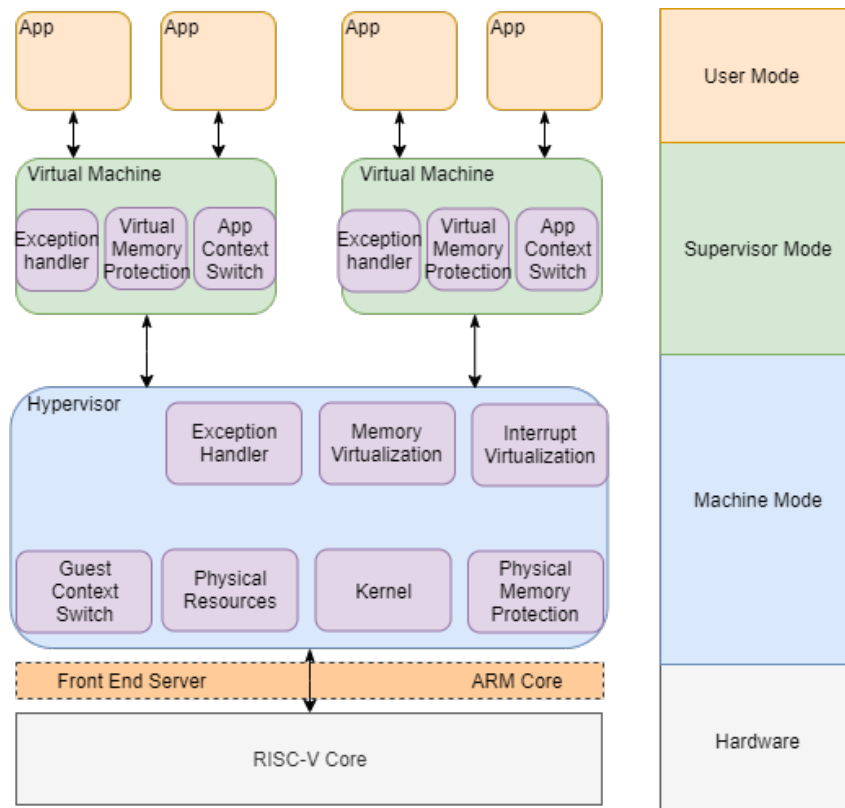


Figure 4.4: System Overview.

4.3 Implementation Considerations

In this section, some pre-implementation considerations are made. A comparison to the LTZVisor, memory protection constraints, timer share, and the virtualization approach.

4.3.1 Comparison to the LTZVisor

In the following subsection, the approach to some important topics will be made stating the differences between implementations the LTZVisor and the V-Visor

4.3.1.1 Scheduler

While the LTZVisor implements an asymmetric scheduling policy, since in this thesis scope there is no secure or non-secure VMs, the scheduling algorithm used, as stated before, is the RR which allows for a simple and deterministic implementation while providing temporal isolation between the VMs.

4.3.1.2 Memory Partition

In this matter, the V-Visor and LTZVisor are similar. While the LTZVisor takes advantage of the TZASC, in the V-Visor each guest has its own region of virtual and physical addresses. The memory protection mechanism is the provided PMP. The usage of PMP consists of a configuration of registers to enable lower level privileged modes to access certain memory regions for either reading, writing or executing. Since the guests are compiled prior to the hypervisor, they can be compiled to any memory region since, after that, the binary is placed, by the hypervisor linker scrip, in a pre-set memory region. This is done so that the VMM has a complete awareness of where all the system's resources and memory regions are mapped and can implement the PMP mechanisms to provide access to memory for each guest.

4.3.1.3 MMU and Cache Management

While the LTZVisor takes advantage of an extended bit in the processor cache to take away the need for cache flushes, in this thesis scope the cache is checked by the PMP and, so, there is no need to flush it after changing guests and, consequently, the PMP configuration. So, even if a memory position loaded into the cache by another guest tries to be accessed by another guest which does not have permissions for it, the PMP will trigger a trap.

4.3.1.4 Device Partition

LTZVisor implements a device assignment based on partitions which provides secure and non-secure devices. Device usage by guests, in this thesis scope, is done through the usage of PMP associated with external interrupts. The PLIC peripheral is a good example of how this is done. When a guest is running, it can access the PLIC memory region but, if the guest tries to modify variables it does not have permissions to (eg: enabling an external interrupt), it will not be able to do so.

4.3.1.5 Interrupt Management

When comparing the interrupt management, the LTZVisor implements a direct interrupt delegation to the secure world which lowers the interrupt latency of the guest. However, the interrupt management approach is a little different in the V-Visor since there are no secure or non-secure interrupts as well as no priorities. This thesis approach intends for the VM to take care of possible delegated interrupts by the hypervisor. Although that to set and clear an interrupt pending bit a call to the hypervisor must be made, the fact that when the interrupt is triggered the routine to process it stays within the VM which takes a workload of the hypervisor. However, this approach induces overhead since, in order to reach the VM exception handler, the exception handler of the hypervisor must be called first. When a guest is running and an interrupt for another guest was meant to be taken, the interrupt will only be processed by the corresponding guest after the time slice of the current guest is over. For example, if a guest configured a

timer to give an interrupt in 3s and, if another guest was running when that time expired, the interrupt will only be triggered after the current guest ends its time-slice.

4.3.1.6 Time Management

The LTZVisor takes advantage of using a time unity for each of the VMs but, in this thesis scope, the time assets are managed by the hypervisor by using the provided hardware timer. Guests are not aware of the time that has passed and, instead, the hypervisor keeps track of that variable. The hypervisor, then, schedules each interrupt based on the time left for each one. If three timer interrupt are scheduled: guest change, guest0 timer interrupt, and guest1 timer interrupt and the guest change interrupt is the one with less time left to rise, it is scheduled to be the first to be triggered. On the other hand, if guest0 is running and guest1 timer interrupt is the next scheduled, when guest0 pauses the execution flow, the guest1 interrupt is triggered and guest1 resumes execution.

4.3.2 Timer

There were four possible approaches to provide the guest with timer interrupts. The first one was to only allow for the guest to configure the timer to trigger an interrupt inside the time slice left of its runtime. This was discarded since the guest would be very limited to timer interrupts and would have the need to, at any point, know how much time it would have left which would take away the guest's original system goal. The second one was to let the guest have an interrupt at any time it would want even if it meant that another guest had to be stopped to attend an interrupt. This could not be the solution since there are no priorities implemented, which would mean that more values had to be stored during a guest switch which would increase the context-switch overhead and ultimately, a guest could monopolize the timer by having constant timer interrupts which would lead to constant context-switches and downgrade the performance to zero. The third option was to enable virtualization of the timer which would keep track of the time each guest had been running for and only count the time individually for each guest. For example, if a guest had a 3.5s interrupt configured, and each time slice was 2s, it would run the first time slice plus 1.5s of the second time slice before triggering the interrupt. This could bring problems since if the number of guests grew, it could reach a point where a more than 4s interrupt would only be triggered after a long time past that. The fourth and final approach which was implemented was to always keep track of real-time and trigger interrupts, based on it, during each guest runtime. For example, if there were two guests configured, and the first one had a 3.5s interrupt scheduled, that guest would run its 2s, the next guest would run its 2s and when it returned to the first guest, since the total of time (4s) that had past was higher than the 3.5s asked by the first guest, the interrupt is triggered first thing after the context-switch.

4.3.3 Memory Protection

While an instruction address and the memory position it is trying to access can be inside the same PMP configuration, four *pmpaddr* registers are exactly enough to allow the guest to have permission to access unlimited memory regions. This is due to the fact that in a worst-case scenario the instruction and the memory position it is trying to access are in different configurations and both of those regions configure the PMP as TOR. Since each TOR configuration needs two *pmpaddr* registers, two TOR configurations need four *pmpaddrs*.

4.3.4 Virtualization approach

The V-Visor has features of both full and paravirtualization. From full-virtualization, the hypervisor's interaction with hardware and the possibility of multiple guests to run on top of the same hardware platform are key features. This approach grants that the hypervisor provides isolation between guests and requires software-assisted virtualization to process privileged instructions.

As for paravirtualization, the V-Visor has features such as the entire system working as a cohesive unity, being the guests compiled prior to the hypervisor, placed in predetermined memory space, and the usage of system calls to communicate with the hypervisor which means that minimal changes to the guest OS must be made.

4.4 Startup

The main goal of the startup is to bring the system to a first working state. From the GPRs and CSRs initialization to the main function execution. As stated before, since this is a critical section of the system, it was written in assembly language.

The startup process begins in a known memory position and so, the boot is loaded into that memory space. Listing 4.1 enumerates the steps from start to finish of a boot for this system. The *mstatus* register holds information about the privilege mode, interrupt enables as well as the length of registers for lower-level privileged modes.

```
Reset trigger;
Clear the 32 general-purpose registers;
Write mstatus with the according values of interrupt enables and privilege mode;
Write mtvec with the address of the exception handler function;
Intialized stack pointer, global pointer and thread pointer;
Clear the bss section;
Call main function;
```

Listing 4.1: Startup.

The main function (listing 4.2) is always the last step of a boot process so it can take advantage of all the system's resources when entering development zone. It also intends to prevent inadequate modifications to the boot code that could lead to system failure.

```

1 int main()
2 {
3     GUEST_NUMBER=0;
4     Fill_PMPStruct();
5     Fill_Plic_Info();
6     Config_PMP();
7     scratch_update();
8     Delegate_traps();
9     HW_TIMER_CONFIG();
10
11     Start_First_Guest();
12     return 0;           //It should never reach this point
13 }
```

Listing 4.2: Guest information structure

The first step of the main function is to reset the active guest variable to zero. After that, a function that will translate the configurations to relevant information and store it in a data structure is executed. Shortly after, with the resulting information, the PMP registers will be written which is detailed in section 4.6. In order to update the registers and CSRs for each guest, another function is called followed by a delegation of exceptions to lower privilege levels. The PLIC configuration follows right after and finally the hardware timer is configured before starting the first guest.

4.5 Exception Handler

The exception handler acts as a recovery mechanism from an intended or non-intended exception. The CSR *mtvec* holds the address to where the exception handler is located in memory. When an exception occurs the context has to be saved to ensure that registers are not polluted by the handler. CSRs such as *mcause*, *mepc* and *mbadaddr* are also loaded with important information about what triggered the exception in order to be used in the handler to give an according response. *Mcause* is loaded with the corresponding value of the tabled exception (see table 4.2), *mepc* is loaded with the PC value prior to entering the handler, and *mbadaddr* is loaded with the memory position the instruction that triggered the exception was trying to access. This last CSR is only loaded in traps.

After the system's response is done, when returning from the exception handler, the CSRs as well as the context must be restored. In order to be able to do so, the registers must be stored in a know memory space which will be explained more in-depth in the section 4.7. To end the handler, a returning instruction is performed depending on the privilege level it is currently running as. To change the privilege levels, from machine to supervisor, when entering a guest the bits of Machine Previous Privilege (MPP) in the *mstatus*

register must be written with the value 1. 0 stand for user mode and 3 for machine mode. 2 is reserved for future use.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved for future standard use
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved for future standard use
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved for future standard use
1	11	Machine external interrupt
1	12-15	Reserved for future standard use
1	≥ 16	Reserved for platform use
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved for future standard use
0	15	Store/AMO page fault
0	16-23	Reserved for future standard use
0	24-31	Reserved for custom use
0	32-47	Reserved for future standard use
0	48-63	Reserved for custom use
0	≥ 64	Reserved for future standard use

Table 4.2: RISC-V exceptions table.

Since the scheduling process is done with a RR approach, the hardware timer interrupt is one of the more important exceptions in the whole system. It has to be well managed since it is shared between the hypervisor and the guest OSs in order to provide time resources to lower privilege levels. The timer interrupt can be triggered through two scenarios: execution time of a guest has ended or lower privilege timer interrupts. As for changing guests, since the temporal isolation is a must in this thesis, a set amount of time is given to each guest to run. As such, the timer comparison by default is to the guest's available time.

After a handler routine, the PC register is also loaded with an address to which the program must jump. If the value is not altered, it will return to the instruction where the exception was triggered.

To enable or disable exceptions two registers are used: *Machine Interrupt Enable (mie)* and *Machine Interrupt Pending (mip)*. The *mie* CSR alongside *mstatus* allow for interrupts enabling when *mstatus*'s enable all bit is set. As an example refer to listing 4.3 where the algorithm for a check to the timer interrupt enable is illustrated.

```

if Mstatus.mie is not enable then
|   all interrupts are off;
end
if Mstatus.mie is enabled then
|   if MIE.mtie is enabled then
|   |   Machine timer interrupts are enabled;
|   end
else
|   Machine timer interrupts are disabled;
end

```

Listing 4.3: Enabling interrupt.

As for the *mip* register, it has the same bit layout as the *mie* which eases the bitmasks used to read and write on these CSRs. Its bits are set when an enabled interrupt is triggered and so, in the handler, that bit must be cleared either by software or hardware by writing to the registers.

The exception processing routine is static which allows for every exception to be processed in the same way with the use of a switch case to redirect the necessary approach to each exception being it an interrupt or a trap. The process of a generic exception processing from start to finish is described in listing 4.4. It is important to refer that software interrupts are used to communicate between cores in a multi-core system. Since this thesis is developed for a single-core platform, software interrupts will not be given a handler processing. The same goes for the machine external interrupts since the hypervisor does not have any way of triggering an external interrupt in the current system. The biggest asset of the exception handler is the *mcause* CSR which allows for an algorithm to process a trigger to an exception almost by itself. The VMM has to support various traps and interrupts for the system to work:

- **Machine timer interrupt:** this is triggered when *mstatus.mie* and *mie.mtie* are set and the time that has past is greater than the timer comparison value. The *mip.mtip* is set and the interrupt is taken by the handler;
- **Instruction address misaligned:** when the types of a variable and a function do not match a trap is taken;
- **Instruction access fault:** when an instruction runs in or tries to access a memory position it does not have permission to a trap is taken;

- **Illegal instruction:** when either a lower privileged mode tries to access CSRs protected or the decode of the instruction determines that the opcode is not valid, an illegal instruction trap is taken ;

```

exception trigger;
store the 32 registers in a know memory position;
read mcause, mepc and mbadaddr to registers that are function parameters;
call the C handler;
read mcause register;
if mcause is a valid exception code then
    if mcause contains a interrupt code then
        if interrupt is enabled then
            | run code to process interrupt;
        else
            | trigger a message to indicate an error;
        end
    else
        if mcause contains an exception code then
            | run code to process exception;
        else
            | trigger an error message;
        end
    end
else
    | trigger an error message
end
write the mepc register with the value intended for the PC;
load the 32 registers from the known memory position;
execute a return instruction;

```

Listing 4.4: Exception processing.

- **Load address misaligned:** when a pointer of a type is converted to a pointer of a different type, a load address misaligned trap might occur;
- **Load access fault:** when the lower privileged mode tries to read from a memory space it can not access, a trap is triggered;
- **Store/AMO address misaligned:** similar to the load address misaligned this trap occurs when a pointer's address is not a multiple of its type's alignment;
- **Store/AMO access fault:** similar to the load access fault, when a lower privileged mode tries to write on a memory space it does not have permissions to, a trap is triggered;

- **Store/AMO page fault:** when a memory page is not mapped by the PMP for the VM, a page fault trap is triggered;
- **Environment call from S-mode:** this trap is triggered when an ECALL instruction is performed while running in supervisor mode and it is a way for the supervisor to call kernel mode for a variety of reasons like turning off an interrupt;
- **Environment call from M-mode:** like the ECALL from S-mode, the ECALL while in machine mode is used to trigger a trap and perform certain blocks of code inside the exception handler.

Some of those exceptions can lead the hypervisor to halt a VM since if there is a compromised VM, it can affect neither the other VMs memory space nor temporal behavior which would lead to total system failure. On the other hand, there are some exceptions, discussed in section 4.6, that are triggered on purpose to enable the change of values of CSRs in order to contain a fault and recover from it.

4.5.1 PLIC

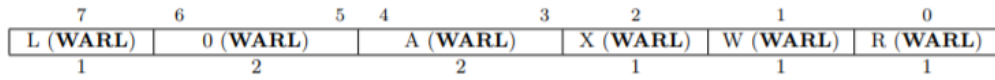
As far as the PLIC goes, it is a peripheral that controls the external interrupts (eg: I/O) for each hart by having pending bits as well as storing the enable, priority, priority threshold, and a completion bit.

This implementation has three possible interruptions. The process of setting up the PLIC information is linear and similar to the PMP mechanism. The values of enabled interrupts from the configuration structure for each guest are translated into a bit map and written to the peripheral which leads to it always having the values relevant to the guest running at that point in time. So, a guest that wants to enable, change the priority or the threshold of an interrupt can do so directly writing to the memory region of the PLIC peripheral.

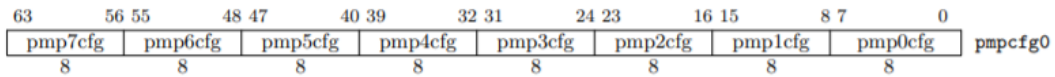
When an interrupt bit is pending the guest exception handler is called which then proceeds to check if the enable bit is on. Since there is a mask to enable bits in the enable register, the guest will never have an undue enabled interrupt. The guest then proceeds to read the claim register which puts the interrupt in a processing state. When the routine is over, the claim register is written with the interrupt ID to signal the end of the handler response.

4.6 Memory Protection

In this thesis scope, the memory protection mechanism is the previously described PMP. The PMP in this core has four usable *pmpaddr*₀₋₃ registers and one *pmpcfg0* register. Inside the *pmpcfg0* are the configuration registers *pmp0-7cfg* (see figure 4.5) which allow the configuration for the corresponding *pmpaddr*. The *pmpaddrs* alongside with a set configuration are used to state the range of memory space that a guest can access. The *pmpaddr* and *pmpcfg* register's layout is illustrated in figure 4.6 and 4.7.

**Figure 4.5:** PMP configuration register format.**Figure 4.6:** PMP address register format.

The fact that the address is [55:2] means that the value of the address intended to be read by the PMP must be shifted twice to the right before loaded into the register. The configuration in table 4.3 also mentions a g variable in the NAPOT configuration which stands for granularity. This g is set by the index of the first bit that has the value zero starting from right to left. If the configuration is NAPOT and the address register has the value 0x01111 then g will be equal to four which would lead to the size of the memory space to be 2^{2+4} .

**Figure 4.7:** PMP configuration CSR layout.

The configuration register has eight different bits that can be set to enable certain permissions or to lock that register until a reset happens. Bit zero through two are used to enable the lower privileged modes to read, write and execute instructions in the memory space defined by the respective PMP registers. Bit six is hardwired to zero and bit seven is the lock bit which allows the configuration register to be unwritable until a reset happens even by machine mode. The bits three and four are used to set the configuration. As stated in chapter 2, there are four configuration modes that can be used and table 4.3 illustrates practically what happens for each one of them.

$pmpaddr_i$	Memory Start	Size
Off	Not used	Not used
NA4	$pmpaddr_i$	4
NAPOT	$pmpaddr_i$	2^{2+g}
TOR	$pmpaddr_{i-1}$	$pmpaddr_i - pmpaddr_{i-1}$

Table 4.3: PMP regions according to configuration.

From a practical point of view, if a lower privileged mode tries to access a memory space it does not have permission to, a trap will happen for the VMM to act accordingly. Sometimes even if a guest has

permissions for that memory region, a trap can occur. This is due to the fact that since there are only four available *pmpaddr* registers, some of the configurations have to stay stored in a structure and when a trap happens, an algorithm (see listing 4.5) will determine if the instruction is inside a valid memory region for the guest, if the memory region the instruction is trying to access is also a valid position, and if the guest has permission to read, write or execute depending on the instruction that triggered the trap. If any of these checks fail, the V-Visor must halt the guest. If the memory region is outside the guest's scope, an instruction access fault will be triggered. If the memory region is inside the guest's scope and it is trying to read a value and the read permission is not set, a load access fault will be triggered. Else, if it is inside the memory region of the guest's scope and the writing permission is off and it tries to write a value, a store access fault will be triggered. Finally, if it is inside the memory region of the guest's scope and it is trying to execute an instruction and the execution permission is off, an instruction access fault will be triggered.

```

PMP trap triggered;
  if Memory address of instruction is inside the scope AND Memory address trying to be accessed is inside the scope
    AND Permissions are on then
    | Change the pmpaddr and pmpcfg registers to the according values
  else
    if Memory address of instruction is inside the scope AND Memory address trying to be accessed is inside the
      scope AND Permissions are off then
      | if Read permission is needed and off then
      | | Load access fault
      | else
      | | if Write permission is needed and off then
      | | | Store access fault
      | | else
      | | | if Execute permission is needed and off then
      | | | | Instruction access fault
      | | | end
      | | end
      | end
    else
      if Memory regions of address of instruction is outside the scope OR Memory address trying to be accessed
        is outside the scope then
        | Error Message and exit!
      end
    end
  end
end

```

Listing 4.5: PMP trap processing algorithm.

The PMP mechanism is configured according to the hypervisor memory placement of a guest. While the guest can be compiled to a certain memory space the hypervisor will place it in a different one which would mean that if the PMP was set to the guest's view of the memory region it would not work. An

example of this translation is illustrated in figure 4.8.

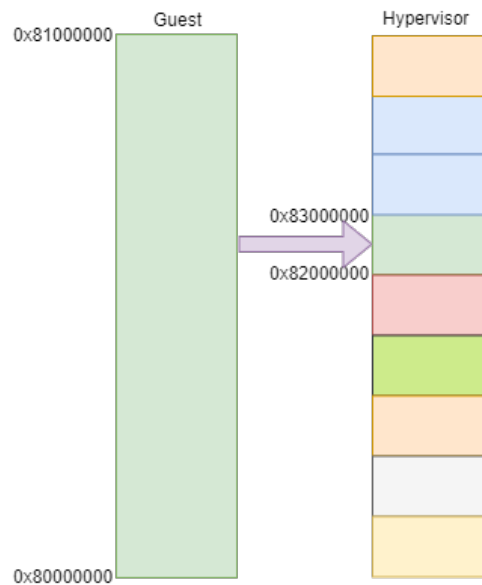


Figure 4.8: Memory translation and protection

4.7 Guest Management

In order to start a guest, it is needed to develop a mechanism to save all the information about it. This mechanism is centered on a guest management structure (see listing 4.6). Each guest will have a full structure with its registers, relevant CSRs as well as exceptions information namely the PLIC registers which will be described in the following section. When the hypervisor's boot is finished and the main function is called, the first step is to, using an algorithm that translates the configuration provided for each guest, store that information in the guest's info structure. That is the first step in starting a guest.

```

1 struct Guest_info {
2     uint64_t gepec;
3     uint64_t gstatus;
4     uint64_t gtvec;
5     uint64_t gscratch;
6     uint64_t gscause;
7     uint64_t gbaddr;
8     uint64_t gie;
9     uint64_t gip;
10    uint64_t gtime;
11    uint32_t prio[MAX_PLIC_INTERRUPTS];
12    uint32_t pend;
13    uint32_t enbl;
14    uint32_t threshold;
15    struct {
16        uint8_t napotPower;
17        uint8_t pmp_config;
18        uint8_t pmp_flags;

```

```
19     uint64_t pmp_addr;
20 } pmp_regs[MAX_PMP];
21 struct {
22     uint64_t zero;
23     uint64_t ra;
24     uint64_t sp;
25     uint64_t gp;
26     uint64_t tp;
27     uint64_t t0;
28     uint64_t t1;
29     uint64_t t2;
30     uint64_t s0;
31     uint64_t s1;
32     uint64_t a0;
33     uint64_t a1;
34     uint64_t a2;
35     uint64_t a3;
36     uint64_t a4;
37     uint64_t a5;
38     uint64_t a6;
39     uint64_t a7;
40     uint64_t s2;
41     uint64_t s3;
42     uint64_t s4;
43     uint64_t s5;
44     uint64_t s6;
45     uint64_t s7;
46     uint64_t s8;
47     uint64_t s9;
48     uint64_t s10;
49     uint64_t s11;
50     uint64_t t3;
51     uint64_t t4;
52     uint64_t t5;
53     uint64_t t6;
54 } context;
55 };
```

Listing 4.6: Guest information structure

After this step, the PMP registers addressed in the previous section have to be initialized. The data from the structure, already parsed from the configuration, is used to fill those registers. Since there are limited PMP registers, only the first configuration's information is used in the first stage which can make available, for the guest, up to four memory zones depending on the configuration for each memory space. Thirdly, also explained more in-depth in the next subsection the mscratch register has to be loaded with the memory position that points to the first register (x0) of the structure that corresponds to that guest. This variable is used for the context save and restore at each context-switch. The last two steps are to delegate exceptions which will be described later in this section and finally starting the guest by loading its binary to a set memory position and start the scheduler since it is, at this point, that the hardware timer is set to start triggering an interrupt from set time to time.

4.7.1 Guest context save and restore

The context save and restore for each individual guest is a must. Otherwise, if all the registers were read and written by all the guests, no coherence would be present in the system. When the hypervisor starts to run a guest for the first time, all its GPRs are initialized with the value zero as expected. However when the processor changes from the guest to the hypervisor either by its time slice ending or an exception, all the registers and necessary CSRs are stored in the guest management structure. This is done by having the *mscratch* CSR loaded with the address of the guest management structure. This allows for all the GPRs to be saved in that memory region. This allows the hypervisor to use the registers and modify them if needed because later, they can all be restored using the same structure. This alongside the PMP also prevents a guest from triggering or rewriting PLIC flags of another guest by having it always saved in memory. Right before that guest resumes execution, all the same registers and CSRs are restored to have a seamless cut in time for that guest execution time (see listing 4.7). It is ,also, at this point that if a guest had a timer interrupt request that should have already be triggered, if another guest was not executing, that interrupt will be triggered right after the context is restored for that guest.

```

Guest is scheduled to start;
  if Guest is not compromised then
    | Context restore;
    | Start guest;
  else
    | Do not start guest;
  end

```

Listing 4.7: Context Restore Algorithm.

A point worth noticing is that being the SP, GP, and TP important registers, when the hypervisor is called, after all the registers have been saved those three must be reinitialized to have a valid value for the hypervisor before calling the C exception handler to ensure that all the data stays coherent since the SP, for example, can not be the same for the guest and the hypervisor.

4.7.2 Exceptions delegation to guest

In order to downgrade the hypervisor workload, a few exceptions can be delegated for each guest to take care of. The delegation of these exceptions does not mean that the guest can disable the pending bits of the interrupt pending register but it does mean that the routine for each of those exceptions can be contained within the guest and only use the hypervisor as a mean to turn the pending bits off. The delegated exceptions are:

- **User ECALL:** A call from the lowest privileged mode of this system. This is triggered when an ECALL instruction is executed while running in user mode. It allows for user mode code to call the guest's kernel to provide support for a process;
- **Instruction page fault:** An instruction page fault trap is triggered when the supervisor mode tries to access an application memory zone that it is not entitled to access. Inside the memory region of the supervisor there can be one or more applications. When those applications are configured, one of the variables is if the supervisor can read or write from that memory region. If it is allowed, a *sstatus* bit is turned on so the trap does not trigger anymore. Otherwise, an error message is displayed;
- **Load page fault:** If the memory page an application is trying to access is not found, a load page fault trap is triggered to inform the supervisor of the occurrence;
- **Supervisor Timer Interrupt:** This interrupt bit pending is triggered by the hypervisor when the time a guest has asked before an interrupt was triggered, has ended. Since each guest can have different use cases to a timer interrupt, the delegation of this interrupt is a must to avoid for the hypervisor to have extensive code that would allow to give an individual response to a timer interrupt from each guest;
- **Supervisor External Interrupt:** A supervisor external interrupt pending bit is triggered when a PLIC bit is pending for the supervisor context. This allows the supervisor to catch the interrupt and run a routine to handle it.

If software interrupts were implemented, the supervisor software interrupt would also be delegated to the guest to give an answer to. While the user ECALL, instruction page fault, and load page fault can be contained for the guest to handle, the timer and external interrupts can not. Still, it provides an hierarchy since the user mode can not communicate directly to the hypervisor creating another isolation layer.

4.7.3 Guest Exception Handler

To give an according response to the previously enumerated exceptions, the guest kernel has an exception handler similar to the hypervisor. The exception handler is responsible for gathering important information about what caused the exception. Before jumping to the exception handler, the guest has to assure a context save but, in this point, it differs from the hypervisor since the registers are not saved in a structure but instead are saved in a known memory region using the *sscratch* as well as the SP of the guest (see figure 4.9). When the exception handler finishes the execution, it restores the values of the registers and performs the *sret* instruction to return to the normal execution of the system.

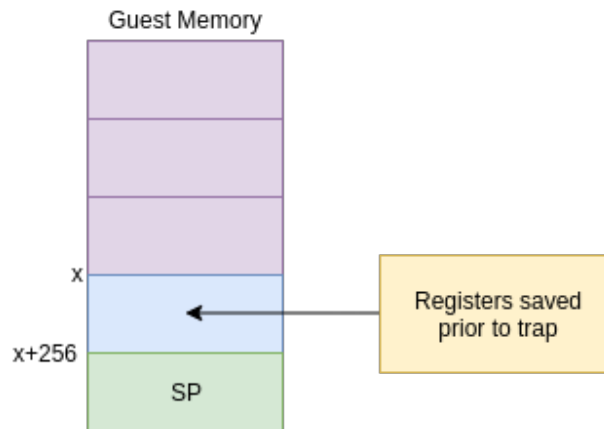


Figure 4.9: Registers save at the beginning of an exception.

The most used interrupt throughout the runtime is the interrupt timer. Since there is only one hardware timer that must be shared by the hypervisor and the guests, well design management must be provided. The algorithm for scheduling timer interrupts is demonstrated in listing 4.8.

```

Guest configures timer;
Hypervisor calculates which time interrupt has the closest time to occur;
if Guest timer is configured to end before the next guest switch then
    Load the mtimecmp with the corresponding value;
else
    if Guest timer is configured to end after the next guest switch then
        Load the mtimecmp with the guest switch time;
        Run the next guest until the end;
        if Time past since the first guest asked for interrupt is lower than the time that has past then
            Trigger timer interrupt for the guest
        else
            Resume executing the first guest and repeat process.
        end
    end
end

```

Listing 4.8: Timer interrupt scheduling algorithm.

The scheduling of a guest's timer interrupt goes through a function that determines which timer interrupt has the lower expiring time. If the scheduled time by the guest for an interrupt is higher than its remaining time to execute, the timer interrupt scheduled to be triggered next should be the guest switch interrupt. After the second guest ends its runtime, if the time since the first guest asked for an interrupt plus the time the second guest has been executing is higher than the time the first guest asked for an interrupt, the interrupt is triggered right before the first guest returns to execution. This assures temporal isolation since the interrupt from the first guest does not interfere with the temporal behavior of the second one.

To help detect bugs on the OS, supervisor mode (privileged mode the VMs run) can not access user memory pages. This changes however if there is a system call from the user mode. To enable the access to memory to the supervisor a bit in the *sstatus* register named Protect User Memory (PUM) must be set. So, when there is, that bit is set, the memory operation is done and the bit is again turned off.

4.7.4 Guest switch

Finally, to perform a guest switch, an aggregation of all the previous points has to be made. To ensure a guest switch is successful, a few steps have to be made:

- When the time of a guest ends, a machine timer interrupt is triggered;
- All the registers are saved in the *guest_info* memory structure;
- The CSRs important to a guest are saved in the guest management structure (see listing 4.9);
- The PLIC information is stored in the *guest_info* structure to later, if needed, to be restored;
- When the saving portion of the switching is done, the next guest's CSRs are written to the active ones (see listing 4.10);
- The PLIC information is written accordingly to the information stored in the *guest_info* structure for the new guest;
- The program counter is set with the value needed for the next guest;
- The registers are restored with the value of the new guest;
- The new guest starts to run;

Optimally, the latency induced by a guest switch would be minimal which would increase performance.

```

1  Guest_inf[GUEST_NUMBER].gepc    = read_csr(mepc);
2  Guest_inf[GUEST_NUMBER].gstatus = read_csr(sstatus);
3  Guest_inf[GUEST_NUMBER].gtvec   = read_csr(stvec);
4  Guest_inf[GUEST_NUMBER].gscratch= read_csr(sscratch);
5  Guest_inf[GUEST_NUMBER].gscause = read_csr(scause);
6  Guest_inf[GUEST_NUMBER].gbaddr  = read_csr(sbadaddr);
7  Guest_inf[GUEST_NUMBER].gie     = read_csr(sie);
8  Guest_inf[GUEST_NUMBER].gip     = read_csr(sip);

```

Listing 4.9: Guest information structure

The most relevant guest CSRs are: the PC value before its runtime ended which is stored in the *mepc*, the status of interrupt enables which is stored in the *sie*, the address to the exception handler which is stored in the *stvec*, the address needed to perform the previously described context save inside a guest

which is stored in the *sscratch*. For the exception handler, the individual interrupt enable and pending bits are stored in the *sie* and *sip* respectively and finally, the information that an exception might need for that guest is stored in the *scause* and *sbadaddr*. These saved registers might be changed under a particular occasion. If a timer interrupt from a guest was scheduled to happen at a time that another guest is running, when the guest that had a pending interrupt is scheduled to be the next to run, the *sip* CSR has its supervisor timer interrupt bit set in order for the interrupt to be attended as soon as the guest resumes execution.

```

1  write_csr(mscratch, &Guest_inf[GUEST_NUMBER].context.zero);
2  write_csr(sstatus, Guest_inf[GUEST_NUMBER].gstatus);
3  write_csr(stvec,   Guest_inf[GUEST_NUMBER].gtvec);
4  write_csr(scause,  Guest_inf[GUEST_NUMBER].gscause);
5  write_csr(sbadaddr, Guest_inf[GUEST_NUMBER].gbaddr);
6  write_csr(sip,     Guest_inf[GUEST_NUMBER].gip);
7  write_csr(sie,     Guest_inf[GUEST_NUMBER].gie);
8  write_csr(sscratch, Guest_inf[GUEST_NUMBER].gscratch);

```

Listing 4.10: Guest information structure

When a guest resumes execution, all the described CSRs must be restored in order for the guest to have the feeling of running throughout all the timeline of the system. PLIC information to the current guest is stored in the guest information structure and so, every time there is a guest switch, that structure must also be update with the values of the next guest so that an incorrect interrupt related bit is not set.

To summarize, the context switch can be defined in three major steps: guest context save, structure's information update, and context restore. The illustration of the guest switch overview is done in figure 4.10.

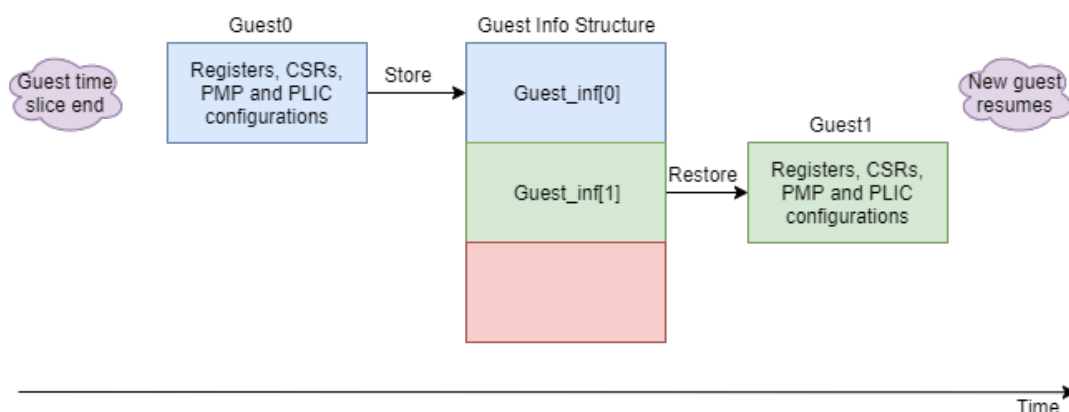


Figure 4.10: Guest switch process.

5. Evaluation and Results

This hypervisor was submitted to tests while running on a Zybo board which, as stated before, has two processors: a dual-core ARM-Cortex A9 which runs at 650MHz as a front-end server and the RISC-V core which runs at 500MHz. Having performance and low latency as requirements, tests to the V-Visor size as well as context-switch overhead, and interrupt latency for guests are made. Both the V-Visor and guests were compiled using the RISC-V GNU toolchain with -O3 optimizations. To have a fair comparison against the LTZVisor, the results displayed will be referent to only two guests running although this thesis's hypervisor has support for more.

5.1 Memory Footprint

To get the memory footprint values for V-Visor, the RISC-V GNU toolchain's size tool was used. By using the generated .elf file as input, as an output it gives the size of .text, .data, and .bss sections. A comparison between the LTZVisor, V-Visor without guests and V-Visor with two guests is made in table 5.1. The three given sections are:

- .text: which is the section which stores all the code from the binary;
- .data: which contains all the initialized data;
- .bss: is where all the non-initialized data is stored. It can be seen as an optimization since the entire section is not present in the binary but is filled by the loader.

As seen by the results, the introduction of guests increases the size of .text up to 261%, .data up to 9321% and .bss up to 498%. This is understandable since the V-Visor uses structures to store the

System	.text	.data	.bss	Total
V-Visor without guests	4634	264	4102	9000
V-Visor with two bare-metal guests without the possibility to add more	16284	6417	10269	32970
V-Visor with two bare-metal guests but possibility to extend up to five	16710	24873	24549	66132
LTZVisor with two bare-metal guests	2368	0	512	2820

Table 5.1: Memory footprint comparison.

information relative to each guest and so, the `.data` section is much bigger as the number of guests increases in the V-Visor. The increase in `.text` is due to the fact of code from the guests being taken into account.

Since the LTZVisor is dual-guest and running FreeRTOS and GPOS as guests, comparing the size of the LTZVisor to the V-Visor's with any bare-metal guests running would be an unfair comparison. So, to have a simple, but fair comparison, the values checked will be from both systems running without any guests. The V-Visor has a 96% bigger `.text` section which means more lines of code, 264 more `.data` section which is due to the fact of variable initializations, and a 700% bigger `.bss` segment which means that the V-Visor has a lot more of non-initialized variables. As final size, the V-Visor is 219% bigger than the LTZVisor.

As for lines of code, there are 1628 C lines from header files, 933 C lines from source files and 159 lines from assembly files. It is important to know that the `.data` segment is linear throughout the process of adding more guests. Since the memory structures stay static in size throughout the system execution time, the size increase in `.data` for each guest is relatively easy to predict.

5.2 Context-Switch Overhead

To evaluate the context-switch overhead of the V-Visor, firstly it is key to enumerate the steps that it takes. For comparison, the gathered results will be from two guests switching contexts while an interrupt is pending. The time to measure the context-switch overhead can be dividie in steps:

1. **Save the GPRs from the guest who ended its time slice:** Since there are thirty-one GPRs, it is key to store them in a predetermined memory region which will not be altered outside of the correspondent guest's context save;
2. **Save the CSRs:** After all the GPRs are saved, the CSRs are the second key part of a context-switch;
3. **Save PLIC information:** To avoid collision between read or write operations to the PLIC peripheral, all the relevant values -enables, priorities and threshold- need to be saved;
4. **Restore the CSRs guest that is scheduled to run next:** Since the GPRs can still be polluted by the hypervisor when the context-restore component of this process starts, CSRs are the first to be restored. This includes the *sscratch* which holds the address value for the new guest's information structure where all the PMP, GPRs and CSRs are stored;
5. **Restore the PLIC information:** Since each guest has a PLIC configuration, the values for the new guest need to be restored to the peripherals memory space;
6. **Update the PMP registers:** To guarantee spatial isolation, the PMP registers have to be updated so each guest can only access the predetermined memory zones. By doing so, there is no possibility for a guest to access another guest's memory space;

Task	Average Time (ms)	Sum of time (ms)
(1)	0.007	0.007
(2)	0.021	0.028
(3)	0.009	0.037
(4)	0.026	0.063
(5)	0.007	0.070
(6)	0.038	0.108
(7)	0.008	0.116
(8)	0.009	0.125
(9)	0.007	0.132

Table 5.2: Context-switch overhead times.

7. **Update the timer interrupt flag:** If a timer for a guest is configured and an interrupt was scheduled to be triggered while another guest was running, that interrupt must be triggered as soon as possible when the guest starts to run;
8. **Queue next timer interrupt:** To schedule the next timer interrupt, an algorithm to process the values for all the scheduled timer interrupts determines which one has the least time left and updates the *mtimecmp* register to that value so it triggers an interrupt when it reaches that value;
9. **Restore the GPRs for the new guest:** Since the hypervisor's exception handler has finished execution, the last mechanism that needs to be employed right before returning the execution to a guest is restore its GPRs to maintain the integrity of the guest.

To get the results for these nine points, the hardware timer was used. The value was read when it entered a point and again immediately after finishing. Then the value is converted to ms. The final results were taken from a thousand samples and the results are shown in table 5.2. All the tests gave very identical results with a maximum difference of 0.002ms for each average value. To put in perspective to the LTZVisor, the V-Visor has a much bigger context-switch overhead since according to [PPG⁺17], the context-switch takes, on average, 0.011ms which is twelve times less. However, the clock speed is higher on the LTZVisor as well as a smaller context-switch since there is a register bank for each world as well as less GPRs and CSRs that need to be stored. If in direct proportion, the clock of the LTZVisor was running at 500MHz, the estimated context-switch overhead would be 0.015ms which would still be much less than the V-Visor.

5.3 Interrupt latency

The interrupt latency is only a factor to have in mind for timer interrupts. This is due to the fact that only three interrupts can be delegated: timer, software and external. While the external interrupts are triggered via hardware and so, they reach the final handler without any latency, the software interrupts are

Trigger	Latency average time(ms)
Timer interrupt during guest's execution	0.040
Timer interrupt when changing guest	0.226

Table 5.3: V-Visor interrupt latency.

not implemented since the V-Visor is implemented on a single-core. As such, the timer interrupt is the only one that needs to have its flag triggered by the hypervisor.

Since the final handler is the supervisor's, the time since the interrupt is triggered to when it reaches the guest's handler is equal to the latency. The fact that the GPRs save and restore always happen when there is an interrupt, it is expected that the interrupt latency has a worst-case scenario where the latency is higher than the context-switch overhead but, as a best-case scenario it would be lower than it when there is no need to save CSRs or modify the configurations of PMP or PLIC. In table 5.3 the interrupt latency values for the V-Visor are shown. This tests were a result of a thousand samples with a two second time slice for each guest while the system's clock is 500MHz.

The two possible scenarios when a timer interrupt can be triggered are: when the guest is running and when a guest starts to run and the interrupt is pending. When the guest is scheduled to start running, the interrupt latency is higher since when the interrupt bit is pending, before handling it, the hypervisor has to perform the context-switch and store all the variables important to the guest that has finished its time slice and restore the values for the guest that is scheduled to start its execution. More so, only when all of those values are restored the hypervisor proceeds to trigger the timer interrupt and allow the guest to process it. The formula to calculate the latency for an interrupt is:

$$lat(interrupt) = t(context_save) + t(context_restore) + t(inside_hyp_handler)$$

where the context_save time can alternate between two types: GPR save only or GPR plus CSR save, and, following the same logic the context_restore can also change its time due to those two scenarios. The time it stays in the hypervisor handler is constant since it only triggers the interrupt for the guest in both scenarios. Since the average context-switch overhead was 0.132ms, all the interrupt latency values that are below it are relevant to interrupts which happened while a guest was still running which took off the need for a full context-switch.

In these tests, there was one guest that had a timer reload of 1.5s and another guest who had it with 4.5s. This would mean that the second guest would have fewer interrupts while the first one could have up to two interrupts in the same time slice. This ultimately leads to the results being a major part of the first guest.

To put in perspective against the LTZVisor, the RISC-V clock has a speed of 500MHz while the ARM has 667MHz which will reduce the interrupt latency by itself. To add, due to the number of registers, the V-Visor has a larger context save and restore which ultimately leads to even more added latency. The results from the LTZVisor's paper [PPG⁺17] show that with a native version of the FreeRTOS, the interrupt

latency is, on average, 0.089ms which is more than two times faster than the V-Visor. Due to the way interrupts are handled and the clock speed, the LTZVisor's latency is much lower. Even if introduced a world where both clocks were at the same speed - 500MHz -, in direct proportion the LTZVisor latency would only be 0.12ms which is still almost twice as fast.

6. Conclusion

Embedded systems have been through exponential growth in our everyday life. As the Peter Parker principle states "with great power comes great responsibility" which, in this case, states that embedded systems must provide safe and secure environments to avoid compromising a system. So, while designing the system, these metrics must be taken into account.

Since it is more than established that virtualization techniques reduce hardware costs in exchange for a slightly small performance decrease, this thesis goal was to port that technology to a RISC-V core which is an emerging architecture. As for now, the context-switch, memory protection, and exception delegation were implemented to provide a simple but functional environment. However, since the V-Visor was started a few months ago, the core's specification is outdated and would be profitable to update it to the latest one.

The RISC-V core allows the V-Visor to meet all the system's requirements. The PMP mechanism allows for reliability and isolation since the hypervisor can provide great spatial isolation where even cache accesses are checked which leads to the impossibility of a guest or application to access undue memory regions. Exception delegation provides performance, low latency, and maintainability since there is no need to always call the hypervisor to process an exception and instead the guest or application can provide its own routine to each delegated exception. So, if there is a fault, it can be contained to a VM instead of compromising the whole system. Finally, the RR scheduling algorithm allows for the final piece of isolation to be met since there are no priorities and no guest can interrupt another which leads to temporal isolation.

Having only one set of GPRs for all the privileged modes instead of one per privileged mode introduces the need to save all those registers and restore them after an exception. Furthermore, when there is a guest switch, all the supervisor CSRs also need to be saved which introduces a considerable context-switch overhead.

While the memory protection is a simple mechanism to implement, the usage of a configuration file allowed for an abstraction to values and, instead, a focus on names. Although the number of PMP configurations allowed for this core is the bare minimum, it would be nice to be able to have more configurations in order to reduce the need to constantly change the PMP registers since in a worst-case scenario only two memory zones could be given access to a guest and if the given regions start to scale, every time there is a need to access a memory zone outside the scope of the current configurations, a huge cost time overhead would be introduced.

As for the exception delegation, apart from the timer interrupt, it always increases performance and reduces interrupt latency while still taking a workload of the hypervisor. However, in the timer interrupt, it introduces a small interrupt latency since there is a need to set the interrupt pending bit of the supervisor timer interrupt.

To conclude, there are a few aspects that still need to be improved but, overall, it still illustrates the benefits of virtualization. The possibility of having more than one software system running on top of the same hardware for the cost of milliseconds in an interrupt trigger is a great trade-off when there are no critical systems which is the case of the V-Visor. However, a few optimizations to reduce the interrupt latency and context-switch overhead must still be made to increase performance.

6.1 Future Work

Although the V-Visor can provide a full environment for guests to be deployed in, some limitations are still present and future work must be enumerated. To increase performance and keep the system up to date:

- **Update the specification:** The first major improvement can be accomplished by updating the specification to the most recent one. While the RISC-V cores have backward compatibility, the CSR's names change and so, although the elf file can run on top of newer cores, the code itself can trigger errors when compiling;
- **Trap and emulate:** Secondly, the trap and emulate approach consists of trapping a guest's instruction that tries to write on bits it does not have permissions to. A great example is the PLIC where the thirty-two interrupt enable bits are grouped into four bytes which is the smallest granularity of the PMP. Since the guest might only have permissions for one or two interrupts, the PMP still allows it to write on all the thirty-two bits which would lead to turning on or off undue interrupts. So, the hypervisor has to catch those write instructions and apply a mask to emulate its correct behavior.
- **Guest priority:** To provide a full hierarchy to the system, a mechanism to assign priorities to each guest should be implemented. This would allow for a system to support critical guests which must have the least performance downgrade when virtualized;
- **Scheduling algorithm:** While the RR provides a deterministic system, with the introduction of priorities a more sophisticated scheduling algorithm, must be implemented. By changing from the current time-sharing approach to an event-driven one, it must be considered also, the deadlines for each guest that has the same priority;

References

- [ACEP02] Luis Alej, Ro Corts, Petru Eles, and Zebo Peng. Verification of real-time embedded systems. 04 2002.
- [All11] Intel Embedded Alliance. The benefits of virtualization for embedded systems. 2011.
- [AZ87] S. T. Allworth and R. N. Zobel. *The Real-time Virtual Machine — its Properties*, pages 17–32. Macmillan Education UK, London, 1987. https://doi.org/10.1007/978-1-349-18821-5_2.
- [BAEA⁺16] Lyes Bouali, Emad Abd-Elrahman, Hossam Afifi, Samia Bouzefrane, and Mehammed Daoui. Virtualization techniques: Challenges and opportunities. volume 10026, pages 49–62, 06 2016.
- [CAA08] Tommaso Cucinotta, Gaetano Anastasi, and Luca Abeni. Real-time virtual machines. 01 2008.
- [Cha17] Sougata Chakraborty. Parallelism in cpu virtualization and scheduling using cuckoo search algorithm. *International Journal of Grid and Distributed Computing*, 10:1–10, 06 2017.
- [Cli19] David Clinton. So what is kvm?, 07 2019. <https://www.freecodecamp.org/news/aws-just-announced-a-move-from-xen-towards-kvm-so-what-is-kvm/>.
- [CP13] Bekim Cilku and Peter P. Puschner. Towards temporal and spatial isolation in memory hierarchies for mixed-criticality systems with hypervisors. 2013.
- [CSA⁺16] Adriano Carvalho, Vitor Silva, Francisco Afonso, Paulo Cardoso, Jorge Cabral, Mongkol Ekpanyapong, Sergio Montenegro, and Adriano Tavares. Full virtualization on low-end hardware: A case study. pages 4784–4789, 10 2016.
- [CSFP20] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. 01 2020.
- [CW05] S. Cavanagh and Yingxu Wang. Design of a real-time virtual machine (rtvm). pages 2021 – 2024, 06 2005.
- [DMMF⁺19] Stefano Di Mascio, Alessandra Menicucci, Gianluca Furano, Claudio Monteleone, and Marco Ottavi. *The Case for RISC-V in Space*, pages 319–325. 05 2019.
- [DN14] Christoffer Dall and Jason Nieh. Kvm/arm: the design and implementation of the linux arm hypervisor. volume 49, pages 333–348, 02 2014.

- [Fir17] Donald Firesmith. Virtualization via containers, 09 2017. https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html.
- [FW14] Tord Fauskanger and Herman Wintermark. Virtual machines for embedded software development. 2014.
- [GA16] Waldemar Graniszewski and Adam Arciszewski. Performance analysis of selected hypervisors (virtual machine monitors - vmms). *International Journal of Electronics and Telecommunications*, 62, 08 2016.
- [GCL16] Amitava Gupta, Anil Kumar Chandra, and Peter Luksch. *Real-Time and Distributed Real-Time Systems: Theory and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 2016.
- [GZ12] Zonghua Gu and Qingling Zhao. A state-of-the-art survey on real-time issues in embedded systems virtualization. *Journal of Software Engineering and Applications*, 5:277–290, 01 2012.
- [HAF11] Xiaohui He and Jim Alves-Foss. A lightweight virtual machine monitor for security analysis on intel64 architecture. *Journal of Computing Sciences in Colleges*, 27:155–162, 10 2011.
- [Hei08] Gernot Heiser. The role of virtualization in embedded systems. pages 11–16, 04 2008.
- [HRL⁺08a] Hermann Härtig, Michael Roitzsch, Adam Lackorzynski, Björn Döbel, and Alexander Böttcher. L4 - virtualization and beyond. 01 2008.
- [HRL⁺08b] Hermann Härtig, Michael Roitzsch, Adam Lackorzynski, Björn Döbel, and Alexander Böttcher. L4 - virtualization and beyond. page 4, 01 2008.
- [HRRH14] Christian Herber, Andre Richter, Holm Rauchfuss, and Andreas Herkersdorf. Spatial and temporal isolation of virtual can controllers. *SIGBED Rev.*, 11(2):19–26, September 2014. <http://doi.acm.org/10.1145/2668138.2668141>.
- [KK12] David Kleidermacher and Mike Kleidermacher. *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Newnes, Newton, MA, USA, 1st edition, 2012.
- [KL18] Christos Koulamas and Mihai Lazarescu. Real-time embedded systems: Present and future. *Electronics*, 7:205, 09 2018.
- [Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2011.
- [LL12] Vincent Legout and Matthieu Lemerre. Paravirtualizing linux in a real-time hypervisor. *ACM SIGBED Review*, 9:3, 06 2012.
- [LSS14] Pankaj Lathar, Yudhvir Singh, and Girish Sharma. Memory virtualization technique for efficient access of data resources in cloud environment. *International Journal of Computer Applications*, 99:9–13, 08 2014.

- [LUC⁺19] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. 09 2019.
- [M.16] Mounika M. A comprehensive review on embedded hypervisors. *International Journal of Advanced Research in Computer Engineering Technology (IJARCET)*, 5:2, 05 2016.
- [MAC⁺17] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. Lrtzvisor: A secure and safe real-time hypervisor. *Electronics*, 6:93, 10 2017.
- [Men13] Mentor embedded hypervisor datasheet and features, 2013. https://www.presseagentur.com/media/3579/Hypervisor_Datenblatt.pdf.
- [Min] Minos - type 1 hypervisor for armv8-a. <https://github.com/minos-project/minos-hypervisor/blob/master/README.md>.
- [MTS⁺] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*.
- [OCMP19] A. Oleksiak, S. Ciełlak, K. Marcinek, and W. A. Pleskacz. Design and verification environment for risc-v processor cores. In *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, pages 206–209, June 2019.
- [Ok199] Vojin G. Oklobdzija. Reduced instruction set computers. pages 3–6, 1999.
- [PAO⁺19] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares. Virtualization on trustzone-enabled microcontrollers? voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304, April 2019.
- [PAR14] Nusrat Pasha, Amit Agarwal, and Ravi Rastogi. Round robin approach for vm load balancing algorithm in cloud computing environment. 2014.
- [PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. <http://doi.acm.org/10.1145/361011.361073>.
- [PM19] Sandro Pinto and José Martins. The industry-first secure iot stack for risc-v: a research project. 06 2019.
- [POP⁺17] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares. Lightweight multicore virtualization architecture exploiting arm trustzone. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 3562–3567, Oct 2017.
- [Por12] Matthew Portnoy. *Virtualization Essentials*. SYBEX Inc., Alameda, CA, USA, 1st edition, 2012.
- [PPG⁺17] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. Lrtzvisor: Trustzone is the key. 06 2017.

- [PS19] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6):130:1–130:36, January 2019.
- [Qui19] Rich Quinnell. Creating a custom processor with risc-v. 02 2019.
- [Rou18] Margaret Rouse. Embedded system, 11 2018. <https://internetofthingsagenda.techtarget.com/definition/embedded-system>.
- [RP18] Jithin R and Chandran Priya. Secure and dynamic memory management architecture for virtualization technologies in iot devices. *Future Internet*, 10:5, 11 2018.
- [SU18] N Saravanan and A Umamakeswari. Security vulnerabilities of virtualization technique. *International Journal of Engineering and Technology(UAE)*, 7:478–483, 04 2018.
- [VMW06] Virtualization overview. 2006.
- [Wan17] K.C. Wang. *Embedded and Real-Time Operating Systems*. 01 2017.
- [Wha] What is kvm? <https://www.redhat.com/en/topics/virtualization/what-is-KVM>.
- [WLA⁺16] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David Patterson, and Krste Asanovi. The risc-v instruction set manual volume ii: Privileged architecture privileged architecture version 1.9 document version 1.9, 07 2016. <https://people.eecs.berkeley.edu/~krste/papers/riscv-privileged-v1.9.pdf>.
- [YYL⁺12] D. Yamaguchi, T. Yajima, C.-Y Lee, H. Shimada, Yuki Kinebuchi, and Tatsuo Nakajima. Spatial isolation on realtime hypervisor using core-local memory. pages 415–421, 01 2012.
- [Zyb17] Zybo™ field-programmable gate array board reference manual, 02 17. https://reference.digilentinc.com/_media/reference/programmable-logic/zybo/zybo_rm.pdf.