

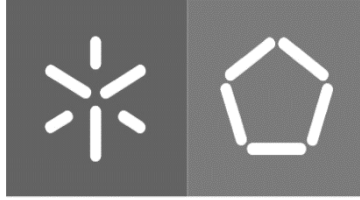


Universidade do Minho
Escola de Engenharia

Ailton Andersen Sanches Ferreira Lopes

Porting LTZVisor to i.MX 6Quad SD

Outubro de 2018



Universidade do Minho
Escola de Engenharia

Ailton Andersen Sanches Ferreira Lopes

Porting LTZVisor to i.MX 6Quad SD

Dissertação de Mestrado em Engenharia Electrónica Industrial
e Computadores

Trabalho efectuado sob a orientação do:

Doutor Sandro Pinto

Doutor Adriano Tavares

Outubro de 2018

Declaração do Autor

Nome: Ailton Andersen Sanches Ferreira Lopes

Correio Eletrónico: a72852@alunos.uminho.pt

Passaporte: J513940 - Cabo Verde

Título da dissertação: Porting LTZVisor to i.MX 6Quad SD

Ano de conclusão: 2018

Orientador: Professor Doutor Sandro Pinto

Designação do Mestrado: Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e Computadores

Área de Especialização: Sistemas Embebidos e Computadores

Escola de Engenharia

Departamento de Eletrónica Industrial

De acordo com a legislação em vigor, não é permitida a reprodução de qualquer parte desta dissertação.

Universidade do Minho, 29/10/2018

Assinatura: Ailton Andersen Sanches Ferreira Lopes

Acknowledgments

This thesis marks the completion of a generations of work and sacrifices that would not have been possible without the aid and support of countless people over the years. Firstly, I would like to express my sincere gratitude to my advisor and professor Sandro Pinto who today I can call a friend, for his continuous support, patience and availability to help with all sort of problems. I am deeply indebted to my professor Adriano Tavares for his fundamental role in my journey at ESG. Adriano's guidance, assistance and expertise has been a fundamental element in my self-growth and mindset. Also I thank the remaining professors and researchers working at ESG for the companionship and support during the thesis. I owe a great debt of gratitude to my fellow "rebarbados" colleagues: Ângelo Ribeiro, Francisco Petrucci, Hugo Araújo, José Silva, Nuno Silva, Pedro Machado, Pedro Ribeiro, Ricardo Roriz and Sérgio Pereira for the stimulating discussions, sleepless nights, and all the fun we have had in the past years. These guys have become family to me, especially my "Deutschgang" brothers Hugo and Sérgio whom I had the opportunity to develop/grow our relationship through the Erasmus program. I am very thankful to professor Nuno Cardoso, my professor for two semester and a friend ever since. Nuno really touched me for his transparency and unquestionable care for his students. I would like to thank all my previous professors specially Dionisio Costa, Nando Gomes, Saturnino Lopes, Sony Amarante, Katie Malkowski and Noemi Silva for being the cornerstone of my educational journey and life long friends. To the friends I made over the year, specially Ailine Lopes, thank you all for the love and companionship over the years.

My thanks also go to Celestino Martins, a life long adviser and supporter that has been my role model since I can remember. The devotion and commitment to follow his footsteps have been the main reason I am in this position today. To my dear Elizabeth, Waldmar, Rosa, Marta, Maisa, Natal, Naniny, Nany, Ismaïl, Dalia, Marie Jeane and Elisa for being my major supporters and the main source of motivation during this journey away from home. Last but not least, I am deeply thankful to Camilo Martins and Antonia Sanches the real architects behind the person im today, for their love, support, and sacrifices. Without them, this thesis would never have been written.

Abstract

Virtualization, a popular and successful concept started in the mainframes, became a fundamental part of cloud computing and it is heavily present in desktop systems has turn into a hot topic in the embedded system domain. Modern embedded devices have, and now it became complex machines by continuously adopting general purpose features to assist and reduce the design cost of their deployed applications. Additionally, in the past years we have witnessed a change in the aforementioned applications design approach, leaving the previous isolated design to become ubiquitous Internet-connected systems.

The new demands of modern embedded applications has lead to the adoption of virtualization as a mean to consolidate and isolate their heterogeneous services/functionalities. The real-time requirements often found in embedded application has turn hardware-assisted virtualization into the main approach in embedded applications due to its lower performance overhead compared to the software-based techniques. Therefore, embedded platforms manufacturer have included hardware to support virtualization in their platforms such as ARM's TrustZone technology. To instigate embedded-virtualization, lightweight TrustZone-assisted hypervisor (LTZVisor) arises as a solution/approach to explore and evaluate how ARM TrustZone can be efficiently exploited to assist virtualization. In this thesis LTZVisor is ported to i.MX 6Quad SD along with the extension of its multicore configuration to a quad-core architecture. Throughout this thesis it is demonstrated the benefits and discussed the identified limitations while showing how they can be overcome. It is conducted extensive experiments with popular benchmark suites, focusing on the penalties incurred on the real-time properties of the real-time operating system, demonstrating how TrustZone-assisted virtualization can effectively meet real-time needs demanded by embedded applications.

Resumo

Virtualização, um conceito popular e bem-sucedido iniciado nos mainframes, tornou-se uma parte fundamental da computação em nuvem e muito presente nos desktops e ultimamente tem sido num tópico importante no desenho de sistemas embebidos. Os sistemas embebidos modernos estão a tornar-se cada vez mais complexos, adotando continuamente funcionalidades de sistemas operativos genéricos para auxiliar e reduzir o custo de projeto. Além disso, nos últimos anos, assistimos a uma mudança na abordagem no projeto de sistemas embebidos, com a migração de projetos isolados para sistemas ubíquos conectados à Internet.

Os novos serviços/funcionalidades encontrados nos sistemas embebidos modernos levaram a adoção da virtualização como uma abordagem para consolidar e isolar o seu ambiente heterogêneo. Os requisitos de tempo-real geralmente encontrados nos sistemas embebidos tornaram a virtualização assistida-por-hardware no principal método de virtualização devido ao seu melhor desempenho em comparação com as técnicas baseadas completamente em software. Com esse reconhecimento, os fabricantes de plataformas de sistemas embebidos passaram a incluir hardware para dar suporte à virtualização nas suas plataformas, como por exemplo, a tecnologia TrustZone da ARM. Para instigar a virtualização em sistemas embebidos, o Lightweight TrustZone-assisted Hypervisor (LTZVisor) surge como uma ferramenta para explorar e avaliar como a TrustZone pode ser eficientemente explorada para auxiliar na virtualização. Nesta dissertação LTZVisor é portado para a plataforma i.MX 6Quad SD juntamente com a extensão de sua configuração multicore para uma arquitetura quad-core. Ao longo desta tese, é demonstrado os benefícios e discutido as limitações identificadas e apresentada formas como estas podem ser superadas. São realizadas várias experiências usando benchmark populares, com foco nas penalizações que afectam as propriedades de tempo-real do sistema operativo tempo-real demonstrando como a virtualização assistida pela TrustZone pode atender de forma eficaz as necessidades de tempo-real dos sistemas embebidos modernos.

Contents

List of Figures	xviii
List of Tables	xix
List of Listings	xxi
List of Algorithm	xxiii
Glossary	xxv
1 Introduction	1
1.1 Motivation and Contextualization	1
1.2 Objectives	4
1.3 Document Structure	5
2 Fundamental Concepts	7
2.1 Virtualization	7
2.1.1 Hypervisor	9
2.1.2 Classical virtualization	14
2.1.3 Dynamic Binary Translation	16
2.1.4 Paravirtualization	16
2.1.5 Hardware-Assisted Virtualization	17
2.1.6 Memory Virtualization	19
2.1.7 I/O Virtualization	22
3 Related Work and Background	27
3.1 Embedded Hypervisors	27
3.1.1 Xen	27
3.1.2 KVM	28
3.1.3 XtratuM	30
3.1.4 Xvisor	31

3.2	ARMv7-A architecture	31
3.3	ARM Security Extension - TrustZone	35
3.3.1	Context-Switching	36
3.3.2	Interruption	37
3.3.3	Processor Configuration	37
3.3.4	Level One Memory System	38
3.3.5	SoC Partitioning	38
3.3.6	Memory Partitioning	39
3.4	TrustZone-Based Virtualization	39
3.4.1	Related Work	40
4	System Specification	43
4.1	LTZVisor	43
4.1.1	General Architecture	44
4.1.2	LTZVisor-AMP (Multicore Extension)	45
4.1.3	Secure Guest	46
4.1.4	Non-Secure Guest	47
4.2	Development Platform	48
4.2.1	ARM Cortex A9 MPCore Quad-Core	49
4.2.2	Central Security Unit (CSU)	49
4.2.3	Pin Multiplexing - IOMUX	50
4.2.4	On-Chip Memory	50
4.2.5	Off-Chip Memory	51
4.2.6	Enhanced Periodic Interrupt Timer (EPIT)	51
4.2.7	Universal Asynchronous Receiver/Transmitter(UART)	52
5	Implementation	53
5.1	LTZVisor: Board Support Package (BSP)	53
5.1.1	Pin Assignment (IOMUX)	53
5.1.2	UART Driver	54
5.1.3	EPIT Driver	55
5.1.4	GIC Driver	57
5.1.5	Security Features	59
5.2	LTZVisor	61
5.2.1	Context-Switching	64
5.2.2	Secure Guest	69
5.2.3	Non-Secure Guest	70
5.2.4	Multicore Extension	73

6	Evaluation	75
6.1	Memory Footprint	75
6.2	Performance	76
6.2.1	Partition Context-Switching	76
6.2.2	Secure OS	78
6.2.3	Non-Secure OS	84
6.3	Discussion	87
7	Conclusion	93
7.1	Future Work	94
	References	108

List of Figures

2.1	Type-1 and Type-2 Hypervisors Virtualization Stack.	10
2.2	Monolithic kernel and microkernel Structure adapted from [40, 41].	11
2.3	Relationships between instruction sensitivity and privilege regarding architecture virtualizability.	15
2.4	CPU Privilege rings.	18
2.5	CPU Privilege rings of architecture with hardware virtualization technology. . .	18
2.6	Two Level Address Translation.	20
3.1	TrustZone privilege levels.	33
3.2	TrustZone General Architecture adapted from [117]	36
4.1	LTZVisor General Architecture.	44
4.2	LTZVisor-AMP Dedicated Core Configuration General Architecture.	45
4.3	LTZVisor-AMP Shared Core Configuration General Architecture.	46
4.4	i.MX 6Dual/6Quad Consumer Grade System Block Diagram from [131]. . . .	48
5.1	LTZVisor's worlds memory partitioning.	60
5.2	LTZVisor context switching cases.	69
6.1	LTZVisor Partition Context Switching Process flow.	77
6.2	Thread Metrics benchmark results on LTZVisor's single-Core and shared multi-core configuration.	80
6.3	N_IRQ Thread Metrics Benchmark Results Variation.	81
6.4	N_FIQ Thread Metrics Benchmark Results Variation.	81
6.5	TZ_FIQ Thread Metrics Benchmark Results Variation.	82
6.6	Thread Metrics benchmark results on LTZVisor's dedicated multicore configuration. .	82
6.7	TZ_FIQ Thread Metrics Benchmark Results Variation.	83
6.8	LMBench memory bandwidth (bw_mem) benchmark results.	85
6.9	LMBench memory bandwidth benchmark L2 Cache Effect (part 1).	86
6.10	LMBench memory bandwidth benchmark L2 Cache Effect (part 2).	87

6.11	Lmbench lat_ops benchmark results.	88
6.12	Lmbench lat_ops benchmark results.	89
6.13	Lmbench lat_ops benchmark results.	90
6.14	Guest-switching rate vs performance overhead.	90

List of Tables

2.1	Performance comparison of the memory virtualization techniques used to manage and translate two-levels page tables.	22
3.1	ARMv7-A Vector Tables.	34
4.1	Permission access table.	50
5.1	LTZVisor hypercalls Table.	70
6.1	LTZVisor Memory Footprint (bytes)	76
6.2	LTZVisor performance statistics (clock cycles).	78

List of Listings

5.1	Non-secure guest creationg.	63
5.2	Non-secure context-switching.	64
5.3	GET_NS_WORLD_CONTEXT macro implementation.	65
5.4	WORLD_CONTEXT_RESTORE macro implementation.	66
5.5	SCR_NS_SET macro implementation.	66
5.6	Monitor FIQ handler implementation.	67
5.7	WORLD_CONTEXT_SAVE macro implementation.	68
5.8	Hypercalls definitions.	71
5.9	Hypercalls definitions.	72
5.10	Itzvisor_wake_secondary_cpu hypercall.	73
5.11	Linux secondary cores enabler function.	74

List of Algorithms

1	UART Driver Initialization algorithm.	54
2	UART Driver GetChar algorithm.	55
3	UART Driver PutChar algorithm.	55
4	EPIT Driver configuration algorithm.	56
5	EPIT enable algorithm.	56
6	EPIT disable algorithm.	56
7	GIC CI initialization algorithm.	57
8	GIC Controller initialization algorithm.	58
9	CSU configuration algorithms.	59
10	DDR TZASC configuration algorithm.	60
11	LTZVisor Hardware initialization algorithm.	62
12	Non-secure context-switching algorithm.	65
13	Secure context-switching algorithm.	67

Glossary

ACP	Accelerator Coherency Port
AMBA	Advanced Microcontroller Bus Architecture
ANSI	American National Standards Institute
API	Application Programming Interface
ARM	Advanced Risc Machine
AXI	Advanced Extensible Interface
BSP	Board Support Package
CCM	Clock Control Module
CE	Consumer Electronics
CISC	Complex Instruction Set Computer
CMS	Cambridge Monitor System
CP	Control Program
CPU	Central Process Unit
CSU	Central Security Unit
CTSS	Compatible Time Sharing System
DMA	Direct Memory Access
EPIT	Input/Output Memory Management Unit
ESRG	Embedded System Research Group
FIQ	Fast Interrupt Request
FLAT	First Level Address Translation
GIC	Generic Interrupt Controller
GPIO	General Purpose Input/Output
GPOS	General-Purpose Operating System
GPT	General Purpose Timer
IBM	International Business Machines
IDE	Integrated Development Environment
IOMMU	Input/Output Memory Management Unit
IOMUXC	Input/Output Multiplexer Controller
IoT	Internet of Things

IPC	Inter-Partition Communication
IRQ	Interrupt Request
ISA	Instruction Set Architecture
LTZVisor	Lightweight TrustZone-assisted Hypervisor
MMIO	Memory-mapped Input/Output
MMU	Memory Managment Unit
MPMU	Message Passing Management Unit
MULTICS	Multiplexed Information and Computing Service
NPT	Nested Page Table
OS	Operating System
PIC	Position-Independent Code
PMIO	Port-mapped Input/Output
PMU	Performance Monitor Unit
POSIX	Portable Operating System Interface
PTE	Page Table Entry
RISC	Reduced Instruction Set Computer
RTOS	Real-Time Operating System
SCU	Snoop Control Unit
SLAT	Second Level Address Translation
SMC	Secure Monitor Call
SMP	Symmetric Multiprocessing
SoC	System-on-Chip
SRC	System Reset Controller
TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
TPM	Trusted Platform Modules
TSS	Time Sharing System
TZASC	TrustZone Address Space Controller
TZMA	TrustZone Memory Adapter
TZPC	TrustZone Protection Controller
UART	Universal Asynchronous Receiver-Transmitter
VFP	Vector Floating-Point
VM	Virtual Machine
VMCB	Virtual Machine Control Block
VMM	Virtual Machine Monitor
XSDK	Xilinx Software Development Kit

1. Introduction

This master's thesis was developed at the Embedded System Research Group (ESRG), part of the ALGORITMI center of University of Minho. It target the expansion of the bulk of hardware platforms that support LTZVisor, a lightweight open-source Hypervisor. LTZVisor is the subject of others thesis currently under development at ESRG.

1.1 Motivation and Contextualization

From the tiny cellphone we carry around in our pockets to complex military satellite system, embedded devices have revolutionized the way we live individually and as a society. These devices are now a fundamental part of our daily routine and lifestyle.

The state of embedded devices today it is largely an aftereffect of continuous advancement of the System-on-Chip (SoC) Transistor Count. In April of 1965, Dr. Gordon E. Moore published Cramming more components onto integrated circuits[1], an article that reshaped the direction of the semiconductor industry. The publication was well acclaimed for proposing the Moore's Law, an observation that predicted that the number of transistors in a dense integrated circuit doubles about every two years. The constantly gain of transistor count is accompanied by gain in the SoC performance and overall complexity. The continuous SoC enrichment pioneered the shift in the scope and aspiration of embedded application, as traditionally embedded systems are by design resource-constrained systems with dedicated function that largely craves for real-time responsiveness. Despite the restricted ISA compared to the CISC machines, modern RISC machines have evolved and grown into complex powerful systems supporting a wide range of functionalities, embodying integrated circuit capable of incorporating billions of transistor packed into a space with the size of a fingernail. The side-effects of this advancement is perceived when designing software for these architectures, i.e., designing software for systems with a higher level of complexity requires a higher engineering cost and it is less efficient than a software designed for a less complex systems. A technique widely used to handle this issues at a reasonable engineering cost is the integration of general purpose functionalities into the embedded application. The high level APIs provided by general purpose operating systems (GPOS) is appropriated to create systems with such characteristics. GPOS not only provide highly abstracted APIs but also

existing software resources such as middleware, network applications and development tools, which when reused saves engineering cost and development time. Engineering heterogeneous system (including both real-time and non-real-time tasks) requires re-evaluating the embedded software design approach.

Security requirements are often found in embedded system as embedded application covers areas such as military and spatial explorations equipments [2, 3, 4, 5], avionic industry [6, 7, 8], automotive industry [9, 10], medical appliances [11, 12, 13, 14], areas where the embedded systems must be rugged and highly reliable due to their high level of criticality. Additionally, in the past years we have witnessed a switch from the antecedent isolated design approach to ubiquitous Internet-connected systems[15], becoming part of the global network infrastructure which enables people, process, data, and things to be connected at anytime, anyplace, with anything and anyone known as Internet-of-Things (IoT). IoT is rapidly emerging as the next generation of communication infrastructure, where myriad of multi-scale end-devices such as sensors and devices are integrated for ubiquitous computing and communication. By associating end-devices with the IoT, this gain the ability to share information across platforms through a unified framework or access cloud computing services. However, this gain of functionalities comes at a price, e.g., the necessity of an end-to-end security mechanism since network connections expose applications to intrusions and malicious attacks. Contrary to IoT service ecosystem and communication infrastructure verticals where security problems and solutions have been consolidated for years, the security solution at the endpoint ecosystem (end-devices) is still very immature, mainly due to the novelty of endpoint devices and the related pressure for low cost, which most of the time prevents the inclusion of security mechanisms [16]. Therefore, developing a systematic mechanism to ensure the security of embedded systems is a demand when dealing with modern safety-critical system. Virtualization, rather a successful technique widely applied to general purpose computers, arises as a possible solution to integrate the heterogeneous system's tasks while improving the systems performance, software design cost and security. Virtualization is a concept, which enables transparent resource sharing by strictly encapsulating modules running concurrently on the same hardware platform while guaranteeing spatial and temporal isolation between them. The major appeals of embedded virtualization are due to its ability to consolidate diversified tasks (general purpose and tasks with real-time requirements) at a reduced size, weight, power and cost (SWaP-C) budget.

Among many existing virtualization techniques, Hardware-based virtualization has become an attractive approach due to the virtualization overhead reduction accomplished by transferring workload previously in software to hardware. This technique is captivating when tackling embedded applications, where it is often demanded time and resources requirements. This has led manufacturer to include hardware to support virtualization such as AMD-V [17], Intel VT[18] for x86 architectures, VE and SE (TrustZone) [19, 20, 21, 22, 23] for ARM Cortex-A and Cortex-M

architectures. Among existent COTS technologies, ARM TrustZone have attracted particular attention, due to the adoption of ARM-based processors in the embedded market, including low-end microprocessors.

It is unequivocal the assets virtualization brings into the embedded world. Therefore, it is important to understand its limitations. Virtualization issues in embedded devices arises across some root fields of attractiveness of server virtualization. Unlike the server domains where the isolated subsystems created by virtualization can thrive independently from one other (strongly-isolated virtual environments), embedded modules are meant to work cooperatively, with one another. Cooperation between the various subsystems of an embedded system requires highly-efficient communication mechanisms largely doing to the real-time dependency of embedded applications. Additionally designing virtualization platforms for embedded environments requires a new approach in the virtual modules scheduling, i.e., combining virtualization with an RTOS creates a two level hierarchy of scheduler, with the hypervisor scheduler at the first level and the RTOS at the second level. Usually the hypervisor schedule the virtual machines as black-boxes, using some kind of proportional sharing of the processor with the control over time requirements entirely entitled to the hypervisor. While this technique operates successfully with virtualization of GPOS, real-time applications demands the awareness of timing properties in order to successfully implement its scheduling policy. With this in mind, choosing the right scheduling algorithm for the embedded hypervisors requires an apriori understanding of the properties of the real-time tasks in the system. The proliferating complexity of modern embedded software poses formidable challenges to system reliability. The coarse-grained modeling of modules isolation features, adds a substantial payload to the hypervisor which reduces its performance. Thus, when addressing hypervisor's complexity for a virtualized embedded platform it is preferred a lightweight virtualization layer with a small-scale overhead and agile inter-modules communication. Thus, partitioning the system into smaller isolated interconnected subsystems disables the proliferating of fault units throughout the system. Additional security mechanisms are needed to secure the inter-modules communication as the embedded system's reliability weights substantially in the information-flow around its components. Therefore, securing the communication of the different modules in a virtualized embedded environment arises as one of the main design requirement. Security mechanisms must allow separating subsystems according to their defined security policy, ensuring that the access to protected data or devices are only allowed to authorized modules.

To instigate embedded virtualization, Lightweight TrustZone-assisted Hypervisor (LTZVisor) [Section 4.1] arises as a tool to expose and evaluate how ARM Virtualization Extension and TrustZone hardware can be efficiently exploited to assist embedded virtualization. TrustZone technology virtualizes a physical core as two virtual cores, providing two completely separate execution domains. The non-secure world acts as a virtual machine (VM) under the control of the secure OS and the hypervisor running in the secure world side. LTZVisor allows the consolidation of

two virtual machines (VMs), running each of them in an independent virtual world (secure and non-secure) supported by TrustZone-enabled processors. The secure world is typically used for running a small TCB size VM with real-time requirements, while the non-secure world is used for running a rich environment. The platform is limited by design to TrustZone-enabled ARM processors, which include all Cortex-A series (ARMv7-A and ARMv8-A) and the new generation Cortex-M series (ARMv8-M). LTZVisor relies heavily on the isolation provided by the TrustZone hardware to ensure the confidentiality, integrity and availability of the secure world components and data.

1.2 Objectives

This thesis aims to enlarge the scope of platforms implementing and supporting LTZVisor along with extending LTZVisor's multicore implementation to a quad-core configuration.

LTZVisor currently has been ported and evaluated on: Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit, ZedBoard Zynq-7000 ARM/FPGA SoC Development Board and Zybo Zynq-7000 ARM/FPGA SoC Trainer Board, supporting both single and dual-core configurations. In this work LTZVisor is ported to i.MX 6Quad Smart Devices Board with the following configurations:

1. LTZVisor Single-Core Implementation:

LTZVisor implementation comes in both single and multicore and this work implements and evaluate the single-core configuration of LTZVisor. In this configuration the secure world dictates the VM's scheduling and the non-secure world only executes in the idle points of the secure world VM. By dictating the scheduling its demonstrated that the Secure world application is capable of meeting its real-time requirements.

LTZVisor Multicore Extension Implementation

The second configuration also implemented in this work aims to extend the existing dual-core LTZVisor implementation to a Quad-core configuration, following two approaches:

(a) Dedicated Secure-core:

In this approach, one core is dedicated to LTZVisor deployed in the secure world and the remaining three cores is dedicated to the non-secure world.

(b) Shared Secure-core:

In this approach the secure world shares the secure core with the non-secure world VM as in the single-core configuration. So, the non-secure world runs with four cores when the secure world is inactive and three when active.

For every configurations mentioned before, it is detailed its implementation, highlighting its benefits and discussing identified limitations and how they can be overcome. It is conducted an extensive set of experiments over the physical hardware platform based on popular benchmark

suites, focusing on the penalties incurred on the real-time properties of the secure guest OS, as well as on performance of the non-secure guest OS demonstrating, how TrustZone-assisted virtualization can effectively meet secure guest real-time needs.

It is also highlighted the implementation of all the infrastructure developed to support the Porting and extension of LTZVisor implementation to the i.MX 6Quad Smart Devices.

1.3 Document Structure

The remaining of this thesis is structured as follows. In the 2nd chapter highlights the fundamental concepts and ideas of virtualization technology and its path to the embedded world. In the 3rd chapter it is surveyed the state-of-art of hypervisors and their design principles tackling embedded applications. The chapter also introduces some foundation that have allowed and assisted during the implementation of LTZVisor. It is also surveyed virtualization groundwork tackling ARM hardware supporting virtualization. The 4rd chapter layout LTZVisor's structure and its highlighted the different implementation supported. In the second part of this chapter is detailed the hardware specific devices and tools necessary to implement LTZVisor in i.MX 6Quad. In the 5th chapter it is highlighted the implementation of LTZVisor's modules, as well as shown the implementation of the different multicore extension configuration. Chapter 4 ends with the specification of the implementation of the MV's secure and non-secure guests. In the 6th chapter it is demonstrated the result of an extensive evaluations over the hardware platform presenting the penalties induced to the system when running LTZVisor. In the 7th chapter it is provided a summary of this thesis and draws conclusions regarding the LTZVisor's implementation in i.MX 6Quad. It is finalized by outlining the road-map for future improvements and research.

2. Fundamental Concepts

In this chapter it is first presented the fundamental theoretical concepts of virtualization starting with a small historical background of the virtualization technology, followed by a definition and classification of hypervisors. Generically, there are two types of hypervisor based on their position and control over the system's hardware. It will also presented the architectural design models used in the development of hypervisors alongside with the main virtualization techniques used both in the mainstream, server, desktop and embedded applications. Finally is presented the main techniques uses in memory and I/O devices virtualization.

2.1 Virtualization

The term virtualization describes the separation of a resource or request for a service from the underlying physical delivery of that service by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation or emulation [24]. The computer's resources are divided into multiple execution environments known as Virtual machines (VM). Each VM is a virtual representation of the native system providing an equivalent environment to run a program compared to a native system. The Virtual Machine Monitor (VMM) or hypervisor is the abstraction layer between the VM and the physical hardware. These two terms (hypervisor and VMM) are treated as synonyms in this document. The hypervisor manages the resources of the underlying hardware and provides an abstraction to one or more virtual machine, i.e., to it, is entitled the responsibility to establish the environment on which the VM's operates.

The history of virtualization technology goes back to the late 1950's and earlier 1960's when IBM introduced the the time-sharing [25, 26, 27, 28] which was the original driving force behind virtualization [26]. The philosophy behind time-sharing is based on a memo submitted by John McCarthy in 1955 and in 1959 proposing a time-sharing system for the IBM 709 computer at MIT [25, 27]. John viewed time-sharing as an operating system that permits each user of a computer to behave as though it was in full control of the computer system [29]. John's memo brought interest to the time-sharing concept and over the next decade or two many time-sharing systems were developed. One of the earliest demonstration of the time-sharing concept was the

Compatible Time Sharing System (CTSS) [27, 30] developed by researchers at MIT, in the first version of CTSS operating system (OS) the time-sharing occurs between four users. The CTSS system had a time-sharing supervisor assisted by a timer interrupt that would periodically swap the users giving them the illusion of having completely control of the hardware. The CTSS evolved throughout the years to support up to 30 virtual machines. The key features of CTSS were virtual machines and hardware isolation of users from the supervisor and from each other, features which are still present in today's virtualization techniques. The CTSS project had the most influence in the history of virtualization [31] serving as pioneer for the development of majority of the classical main virtualization system.

After the success of CTSS, MIT continued valuing the prospect of time-sharing by the MAC project as an effort to develop the next generation of advances in time-sharing technology. In so doing, hardware manufacturers to deliver improved platforms with time-sharing capabilities. IBM's response was the System/360 computer family being the model 67 the most acclaimed for having hardware support to virtual memory [31, 32, 33, 34]. Various OSes were developed for the System/360 family being Time-Sharing System TSS/360 IBM's the official OS. TSS pioneered a number features such as being one of the first symmetric multiprocessing (SMP), some of which later appeared in more popular systems such as MULTICS. TSS provided an early implementation of position-independent code (PIC), the ability to have different processes run a single copy of an executable possibly mapped to a different virtual addresses in each process [35].

IBM also created the Control Program (CP) [31, 34] project whose initial goals was studying the behaviors of a system in a virtual memory environment. It was started since the idea of a virtual machine system had been talked about before, but it had never successfully been implemented. The CP project had strong influences from CTSS but it had a key concept in the design, the bifurcation of computer resource management and user support, i.e the project was divided in two parts, CP and CMS. CP solved the problem of multiple usage by providing separate computing environments at the machine instruction level for each user, while CMS provided single user service free of the problems of sharing, allocation, and protection [27]. Instead of explicitly dividing up memory and other resources among users, which had been the traditional approach, CP was the first successful virtual machine operating system based on fully virtualized hardware, providing each user with a simulated stand-alone System/360 computer. Each system was able to run any System/360 software that ran on the bare machine and in effect giving each user a private computer system. CMS was basically, a lightweight single-user operating system, for interactive time-sharing use. With the development of virtualization technology and the increase of interest in software and hardware virtualization, in 1974 Gerald J. Popek and Robert P. Goldberg created a set of formal requirements for architectures titled "Formal Requirements for virtualizable Third Generation Architectures". The paper explore computer architectures of

third-generation-like machines and set the requirements necessary to determine whether an architecture can support a virtual machine [36]. This requirements has been used as guidelines in the design of virtualizable machines.

While the virtualization procedure discussed previously only summarizes a list of early virtualization efforts, many companies, such as Sun, Microsoft, and VMware, have released enterprise-class products that have wide acceptance. The projects initiated since those days are too numerous to be referenced entirely. Since IBM's first time-sharing system we have see virtualization technology moving from the mainframe systems to the data center industry and in the past decade, with the development and increase of complexity of embedded devices, into the embedded world.

Virtualization has been around for over 50 years and it has changed almost every aspect of how we manage systems, storage, networks, security, operating systems, and applications. The continuous growing of interested in virtualization in today's system is doing vast amount of applications capable of benefiting of virtualization. Many forms of virtualization exist today the most common and successful form is the server virtualization [31]. The technology permits combining and consolidating workloads on a smaller number of physical servers to maximize the investment in hardware. Studies have shown the enormous improvement in energy consumption and resources utilization by applying virtualization for server consolidation [37]. Traditionally, only of 5-15 percent of the server overall capability is used and the utilization rate can be increased when applying server consolidation by adding more virtual servers. The IT organizations can reduce hardware and operating costs by as much as 50 percent deploying the server consolidation solution. In addition to the significant reductions in server hardware expenses made possible by improving server utilization, they have realized savings in costs for administration, cooling and data center infrastructure [38]. Another used form of virtualization is the desktop virtualization. In the desktop domain the virtualization implementations can be classified based on whether the virtual desktop runs remotely (hosted) or locally. In the hosted desktop virtualization model, data and applications run on servers and streamed to clients over a network. That leads to greater security, because data and applications reside on physically secure servers at the same time transferring much of the desktop processing workload from client devices to the data center enabling IT departments to reduce wear and tear on client hardware and thereby extend their life span. In the local desktop virtualization model is normally used an hypervisor to abstract the local hardware allowing one or more virtual machines to run in the same hardware.

2.1.1 Hypervisor

The original virtual machine monitor was created to solve a specific problem, but VMMs have evolved into something quite different. The continuous evolution of the system's computation power has allowed today's hypervisor to have a bigger role and consequently spreading the scope

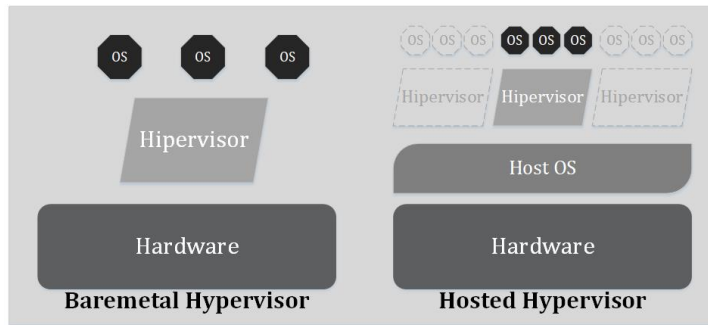


Figure 2.1: Type-1 and Type-2 Hypervisors Virtualization Stack.

of virtualization capabilities in the system's resources. As said before, Hypervisor is the software or firmware component that virtualize the system resources creating the foundation that allows the creation of virtual machines. In their 1974 article, Formal Requirements for Virtualizable Third Generation Architectures, Gerald J. Popek and Robert P. Goldberg classified hypervisors in two types: Type-1 - Native or bare-metal hypervisors, and Type-2 or hosted hypervisors.

Type-1 Hypervisor runs directly on top of the system's hardware and in this version the hypervisor runs in the most privileged mode having full control of the hardware. It creates and monitors the virtual machines that runs directly on top of it. The Type-2 hypervisor is hosted or installed on an already existing operating system, in this implementation the host OS runs in the highest privileged level which means that the hypervisor relies on host operating system for device support and physical resource management. As illustrated in Figure 2.1, in the type-2 hypervisor the hosted operating system has an additional layer above it where the hypervisor resides and the virtual OS resides on the third level. In the type-1 the virtual OS resides on the second layer above the hypervisor that has a direct access to the physical hardware.

The hypervisor's main purpose is managing the resources for the virtual machines running on top of it, and due to the wide range of possible application where virtualization mechanisms are advantageous, a number of different virtualization techniques are available in the market today each one with their strengths and weakness, leaving the adoption up to the intended application. Despite the common purpose, their designs may differ as an effort to meet specific performance expectations, embody desired characteristics, emphasize certain roles, or limit inherent weaknesses [39]. Their differences can be summarized in terms of two hypervisor design architectures: monolithic and microkernel. The monolithic and microkernel are the main architecture approach used when designing the kernel of an operating system. These techniques are applicable in the hypervisor's design due to the common nature existing between the two concepts. For instance, while the OS abstract the hardware, monitors and assist the user space application in a system, the hypervisor monitoring and assistance are performed not in the application level but to a set of Virtual machines. The structure of both technique is illustrated in the figure 2.2.

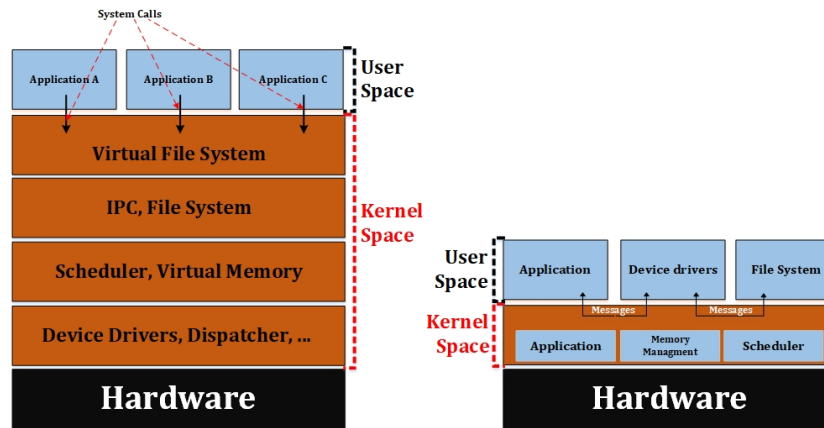


Figure 2.2: Monolithic kernel and microkernel Structure adapted from [40, 41].

The monolithic kernels (in the left) provide a rich and powerful abstractions of the underlying hardware, the microkernel provide a small set of simple hardware abstractions and use applications called servers to provide additional functionalities.

2.1.1.1 Monolithic Architectures

Traditionally the kernel architecture development follows the monolithic approach. Here, the kernel's subsystems such as memory management, filesystems, or device drivers are all implemented in the same domain as illustrated in the figure 2.2. The performance boost in this technique is due to the direct communication between the different kernel components. Being in the same domain allows the different subsystem to directly access both the system's hardware and the others subsystems. This implementation often entitle as the The Big Mess [42] for the lack of a structural separation of the different kernel components but even in this messy system it is possible to find some level of structure. The structural arrangement is found due to the implementation of all the kernel components in the highest processor privilege level (kernel space) separated from the application that are implemented in the user level with lower priority creating two distinctive execution domain. The kernel provide the abstraction of the system's hardware to the used domain through system calls and when the user application request this service the processor trap and switch to the kernel mode where is executed the procedure attached to the system call invoked in the user space. To each system call is attached a certain procedure in the kernel mode and after the procedure is performed the processor switches back to the user mode. The state of the system is saved before the switch and restored back after the procedure is finished. The main concern regarding the monolithic architecture is the lack of structure in the kernel level. This issue can cause huge problem to the system if any subsystem in the kernel space are compromised, i.e. since all the modules run in the same address space, a bug in one module can bring down the whole system. Studies on the subject [43, 44, 45, 46] have shown that the majority of bugs that occur in the kernel is caused by fault in the device drivers. This

matter has made the researches in the field to focus on finding solution capable of dealing with these issues by adding execution isolation between subsystems within the kernel. The kernel execution domain is split in multiple groups isolated from each other. [47] list multiple mechanism used to achieve such design goal. One solution used is the adoption of microkernel architecture that solves the problem caused by fault drivers in the monolithic architectures and at the same time it decreases the kernel overall complexity.

2.1.1.2 Microkernel Architectures

Usually the word Kernel is used to denote the part of the OS that is mandatory and common to all software [48] and the idea behind a microkernel architecture is to minimize the kernel as much as possible. This idea started with the research community efforts on building a post-Unix OS to take advantage of the new hardware resources available [49] and fulfill software requirements such as stated by Liedtke "... minimizing the kernel and implementing servers outside the kernel. Ideally, the kernel implements only address spaces, inter-process communication (IPC), and basic scheduling" when describing the microkernel concept in that same publication.

In a microkernel architecture only the core kernel modules are isolated in the highest privilege level (kernel mode) and the rest of its components are enclosed in modules known as servers in a less privileged level. The server components communicate directly with one another through direct messages (other cases using shared memory) with the microkernel serving as the monitor (receiving, validating and forwarding the message to the destination). The motivation of microkernel-based systems is the minimization of the trust computing base (TCB) size by using a small kernel that implements only a minimal set of abstractions [50, 51]. The TCB is the set of all components whose correct functionality is a precondition for security and generically hardware and kernel both belongs to the TCB. In the microkernel architecture the system's TCB is reduced by removing all the components from the kernel that being outside the kernel space would not violate the security requirements of the user applications. Examples of methods to achieve such condition is explained in [51]. One can see that the performance in a microkernel system depends heavily on the performance of the communication mechanism (IPC) between the components migrated out of the kernel space and for this reason the first generation of microkernels were not well accepted due to their poor performance compared to the well accepted monolithic kernel. Benchmarks test on the first generation of microkernels shown that at least 73% of the overall penalties were related to the communication (IPC) [49]. Since then the improvements in next generation of microkernels have solved most of the performance issues found in the previous generation [52, 53, 54, 55] to a point where adding the extensibility and portability nature of the microkernels are sometimes more appropriate for the adoption of a microkernel architecture over the monolithic.

The minimalist view and the kernel overall size including only the components necessary for the correct execution of the application suits well in the embedded world, where hardware resources such as memory are limited. The improvements made on the second generation microkernels have made it possible to bear the microkernel scheme into the real-time applications world what have lead more researchers to pursue new techniques in designing microkernel based systems. The microkernel architectures performance can get even better with hardware support presented in moderns computer systems. The success of the second generation microkernels has lead to researches in others design approaches for the OS kernel architecture implementation, from the many proposed architectures one notable implementation standout, the Exokernel approach.

2.1.1.3 Exokernel Architectures

The design approach behind this kernel architecture solves some problems related to a fixed high-level abstraction traditionally used in the OS [56, 57, 58], by moving the hardware abstraction previously done in the OS level to be implemented entirely at the application level. This technique uses library operating systems (a low level exokernel interface) to securely multiplex and export the physical resources to the applications level (user space), leaving the abstraction of the hardware entitled to the application using it. This way instead of abstracting the entire system's hardware only the part used by the applications are abstracted. The challenge here is managing the library operating systems access to the physical hardware, i.e. give then the maximum freedom in accessing the resources while isolating them from from each other. To securely export the hardware resources, an exokernel architecture export all the sensitive instructions to the library OS, giving it the capability of abstracting the hardware resources. The exokernel will only abstract hardware necessary for the monitoring of the resources allocation, revocation and ownership and so, it is capable of monitoring the application using the application when necessary. The applications interface the system resources through system calls defined in the library OS. The exokernel also leaves the resources policy decision to the library OS, this way the the application are able to monitor the resources usage and make better decisions when requesting resources allocation. Even by shipping the resources management to the library OS, the exokernel is still the tiebreaker when two applications compete for the same resource. An additional feature provided by the exokernel is the ability to securely multiplex system resources allowing it to be accessed by different applications. To perform this task efficiently, an exokernel allows library operating systems to bind to resources using secure bind, a method detailed in the paper [59]. Additionally, exekernel also implements a revocation feature that gives it the ability of reclaiming resources allocated to a certain application. The main idea behind an exokernel architecture is the overthrow of the extensive hardware abstraction performed traditionally by the OS and hypervisor. These supervisors software/firmware control the entire process of accessing

the systems resources while the exokernal approach moves all the resource access procedures to the application level and intervene only in critical situations.

Currently, there are several virtualization techniques that can be used to support the execution of multiple operating systems in a single platform and similarly to the architecture, the selection of each technique depends heavily on the intended application and platform used. Below is presented the most popular CPU virtualization techniques used nowadays to meet today's current virtualization demands.

2.1.2 Classical virtualization

The guidelines to achieve a classical virtualization environment are defined and discussed in detail in Popek and Goldberg's Formal requirements for Virtualizable Third-Generation Architectures [36]. The Requirements set by Popek and Goldberg demands that in order for a system to be considered VMM it should establish the followings characteristics:

1. **The efficiency property** - All innocuous instructions (non sensitive instructions) are executed by the hardware directly, with no intervention at all on the part of the hypervisor.
2. **The resource control property** - This property requires that the VMM retain full control over system resources, and it should not be possible for a virtualized guest to access or manipulate resources without the VMM's explicit authorization.
3. **The equivalence property** - This property requires that a program running under the VMM should exhibit behavior essentially identical to that demonstrated when running on an equivalent machine directly, with two exceptions: timing and resource availability. Timing will differ doing to processing overheads introduced by the VMM or other guests execution, and resources available due to the utilization of system's resources by the VMM or other VM.

An important concepts of classical VMM implementations is the de-privileging of the virtual system ISA i.e. adding the hypervisor layer in the system instigate a privilege deterioration on the virtual OS level. For example, without virtualization the OS run with the highest privilege level and full control over the system's hardware, in order to separate the OS from the application running on top of it which runs in a less privileged mode than the OS (usually the OS run in the kernel mode and the applications in the user mode). The machine where the OS must runs have two kinds of instructions in the privilege standpoint: privileged and non privileged instructions. Privileged instruction requires the process to be privileged (kernel mode) to be executed. If the process calling a privileged instruction is in a non privileged mode (user mode) the CPU traps and switch control to the supervisory software (running with the highest privilege). With this in mind

when using the classical virtualization technique, the hypervisor is required to have the highest priority level causing a privilege decline in the virtualized OSes that are kept in a less privileged zone. With the priority drop in the virtualized OS, this becomes unfit to execute the privilege instructions.

In a classically virtualizable architecture, it is necessary that all instructions that could affect the correct functioning of the VMM (privileged instructions) always trap and pass control to the VMM. One technique used to implement this concept is known as trap-and-emulate or classical virtualization. Emulation refers to the capability to mimic a particular type of hardware for an operating system regardless of the underlying host operating system. The guest operating system has no awareness of its presence. When the guest executes a privileged instruction it traps giving control to the hypervisor that decodes and interprets the original intended instruction by the guest OS, emulating the instruction, updating the guest state and subsequently resuming the guest execution. This way following the classical virtualization implementation, an architecture must be capable of trapping all sensitive instructions and handle the execution to the VMM that decodes and emulates the instruction. Popek and Goldberg also defined the requirements of a fully virtualizable architecture in formal requirements for virtualizable third-generation architectures stating that "...if the set of sensitive instructions for that computer is a subset of the set of privileged instructions". Sensitive instructions means all the instructions that have the capability of interfacing with any resources controlled by the VMM. If this is not the case, and some sensitive instructions are not capable of being trapped, then the architecture is not fully virtualizable as shown in the figure 2.3.

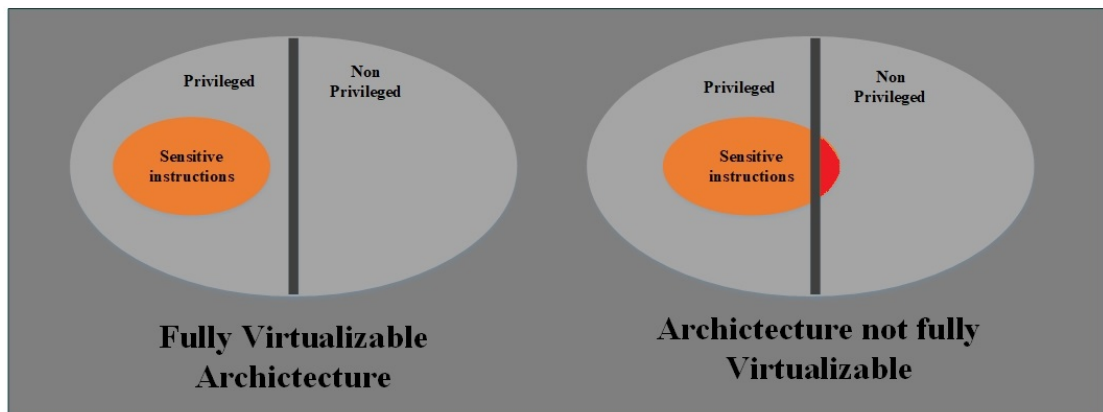


Figure 2.3: Relationships between instruction sensitivity and privilege regarding architecture virtualizability.

The red area in the the not fully virtualized architecture represents the non-privileged sensitive instructions, i.e, the instructions that uses privileged resources and it does not trap when executed in non privileged mode (user mode).

Because of the existence of architectures that are not fully virtualizable and the realization of performance increase when sacrificing some of the requirements established by Popek and

Goldberg, others measures were developed taking virtualization to a wider scope of architecture and applications.

2.1.3 Dynamic Binary Translation

Binary translation is a technique very similar to emulation. It involves running guest code (both OS and application code) on an interpreter that handles any sensitive instructions correctly, i.e. the idea is that the hypervisor can examine a piece of guest code before it runs, and in case it finds any sensitive instructions, these are translated into privileged instructions before running the translated code. Instead of depending on the processor itself to detect the sensitive instructions and pass control over to the hypervisor or host, virtualization software inspects the instruction stream in software. Whenever the virtualization software detects a sensitive instruction, it rewrites it on-the-fly, typically replacing it with a kind of manual trap, that will hand over control to the hypervisor at the appropriate moment. However, this method can have a heavy performance overhead, which can be reduced by using optimization. Examples of optimization include switching between virtualized and translated instructions, depending on the privilege level of the code (an adaptive binary translation). This technique in some cases can outperforming a classical VMM because intercepting instructions causes a context switch, a situation which binary translation can minimize[60].

2.1.4 Paravirtualization

Different from the techniques presented before where all the virtualization mechanism resides outside the VM scope, leaving the VM unchanged when virtualizing the system (full-virtualization), paravirtualization differentiates from them by instead adjusting the virtualization platform to support different guest operating systems and their behaviors (specifically the processor instructions they invoke). The guest operating systems themselves are altered to support the virtualization platform they are running on top of. When modifying the OS kernel the non virtualizable instructions are replaced with hypercalls that when executed redirect the execution flow to virtualization layer. The hypervisor also provides hypercall interfaces for other critical kernel operations such as memory management, interrupt handling and time keeping. The direct communication between the hypervisor and the VM (through hypercalls) allows the VM to coordinate with the hypervisor. It typically outperform fully virtualized virtual machines doing to the reduction of privileged instructions executed in the virtualized OS, the operation often responsible for the major performance penalties in the full virtualization techniques.

Paravirtualization, in cooperation with Kernel modifications on the guest OS, provides improved performance for CPU, memory and I/O virtualization compared to full virtualization [61]. Although paravirtualization performance improvements compared to the classical approach there

are several high level goals which must be balanced in designing an API for paravirtualization. [62] list these the following:

1. **Portability:** When developing the virtualization API it should be taken in consideration that the closer the API resembles the native platform that the OS supports, the smaller is the portability cost.
2. **Performance:** When designing the API's, the hypervisor performance concerns should be taken in consideration, from the fact that many operations require costly trapping procedures. While a low level API that closely resembles hardware is preferred for portability, we should make sure not to sacrifice the performance, since a low level API does not explicitly provide support for higher level compound operations. The interface must not exclude the possibility of optimizing low level operations in some way to achieve the same performance that would be available had it provided higher level abstractions.
3. **Extensibility:** The API must be designed in a way that it would be possible future expansion, The API can be split into sections, and each section can be incrementally expanded as needed.
4. **Transparency:** The API should takes in consideration allowing the same paravirtualized kernel to run on both native hardware and on other hypervisors.
5. **Maintainability:** When developing the API's it should been taken in consideration the continuous maintenance of the guest OS Kernel.

2.1.5 Hardware-Assisted Virtualization

Hardware vendors are rapidly embracing virtualization and developing new features to simplify virtualization techniques. Hardware assisted virtualization techniques can address the overhead imposed by software virtualization by moving many of the computational tasks associated with platform management to hardware. Previously in the introduction it was listed some of the main hardware technology used to enhance virtualization in a set of architectures. The listing includes technology such as Intel Virtualization Technology (VT), a technology used to assist the virtualization in the x86 architecture. Intel-VT introduces x86 architecture with two new forms of CPU operations: VMX root and VMX non-root operations, i.e., instead of the Virtual OS de-privileging often found in the software based virtualization, this technique adds an extra privilege mode known as the Ring -1 where the virtual machine monitor operates. x86 CPU ring architecture also known as hierarchical protection domains and protection rings, is a mechanism design to protect the access to the system's hardware resources. This mechanism is enforced in the CPU architecture that provides more than one mode of execution, with rings arranged in a hierarchy, from the

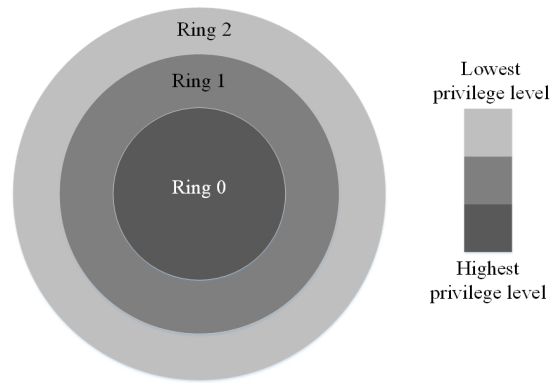


Figure 2.4: CPU Privilege rings.

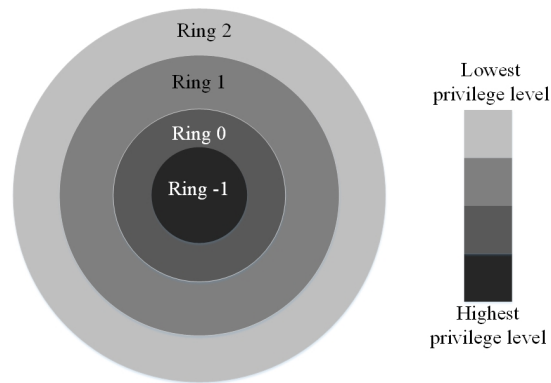


Figure 2.5: CPU Privilege rings of architecture with hardware virtualization technology.

most privileged (ring 0) to the the less privileged (last layer of the ring) as illustrated in the figure 2.4. The number of rings is architecture dependent. The security is maintained by not allowing processes from the outer ring to access resources with higher privilege level, i.e. a process running in the ring 2 cannot access resources in that requires ring 0 privilege level. Natively the OS would run in the ring 0 that is also known as kernel space, and the applications in the ring 2 (user space) guaranteeing the isolation of the OS from the applications and way of supervising system's resources access. Figure 2.5 depicts the the architecture protection rings with the inclusion of the hardware virtualization technology.

Having the virtual machine monitor at Ring -1 grants to the VMM full control over the system's hardware and at the same time hosting unmodified guest OS. The latter can still run at CPU Ring 0, just as it normally would when running natively. It is also implemented hardware based mechanisms to hand-off between the virtual OS and the VMM, relying on hardware to perform switching between both execution level to reduce the additional overhead added to the software based switching mechanisms. An in-memory data structure known as the virtual machine control block (VMCB) is often used to hold the virtual guest state when passing control to the VMM. It also holds others information to assist the VMM, such as conditions variables.

Hardware assisted virtualization also impacts the architecture instruction set architecture

(ISA), i.e., virtualization implementation usually comes with additional instructions necessary to interface the virtualization's hardware, such as instructions to enter and exit the VMM mode (VM Entry and VM Exit). Upon execution of the instruction VMM Entry, the hardware loads guest state from the VMCB and continues execution in guest mode. Guest execution proceeds until some condition, expressed by the VMM using control bits of the VMCB, is reached. At this point, the hardware performs an exit operation, which saves guest state to the VMCB, loads VMM state into the hardware, and resumes in host mode.

Although the hardware technique described above refers to the Intel-VT, this design structure is found also in ARM's VE and SE (TrustZone). ARM VE implements two of additional CPU privilege levels reserved for execution of monitoring software such as the VMM: Monitor and hypervisor modes. The processor's ISA is also extended with new instruction to interface the services provided by the virtualization hardware extension. A detailed description of ARM's VE and SE is provided in the sections 3.2 and 3.3.

2.1.6 Memory Virtualization

Memory virtualization is a well known technique used in the majority of today's general-purpose OS. Due to the advantages leveraged by such technique, the majority of modern processors includes hardware to support memory virtualization. The technique delivers benefits such as increase of security due to the creation of virtual address space for applications, isolating each process in their own address space managed at granularity of pages. Each virtual page is mapped to a physical page by the operating system. All the virtual-to-physical mappings are stored in a page table managed and assigned by the OS. Most commercial processors today use page-based virtual memory management. With virtual memory, the processor must constantly translate addresses from virtual-to-physical and from physical-to-virtual to access the data. To reduce the overhead of these operations, the OS relies as much as possible in hardware support presented in the processor. Hardware support such as the memory management unit (MMU) enable the automation of address translation from both virtual-to-physical and physical-to-virtual, leaving the software only to provide the remaining support. Most MMUs uses an in-memory table to store page tables (data structure used to store the mapping between virtual addresses and physical addresses). Each mapping in the page table is know as page table entry (PTE). Most MMUs also contain an associative cache of PTEs called a translation lookaside buffer (TLB) which is used to avoid unnecessary access to the main memory every time a virtual address is mapped, i.e., the TLB accelerates translation by caching the most recently used page table entries.

When running a virtual machine, the hypervisor creates a contiguous addressable memory space for the virtual machine. This memory space has the same properties as the virtual address space presented to the applications by the guest operating system. This allows the hypervisor to run multiple virtual machines simultaneously while isolating the virtual machine memory spaces.

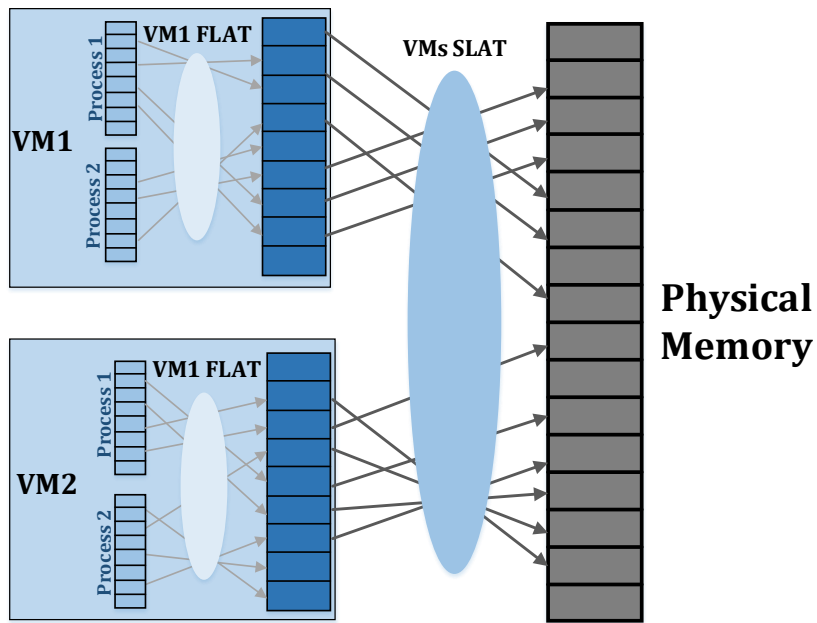


Figure 2.6: Two Level Address Translation.

From the view of the application running inside the virtual machine the Hypervisor had a new layer of address translation as illustrated in the figure 2.6.

As a result, a two-level address translation is required with the inclusion of the hypervisor guest virtual(gVA) to guest physical (PA) and guest physical(gPA) to host physical(hPA). To fully virtualize memory, it is necessary the following address translation:

1. **gVA $\Rightarrow$ PA:** guest virtual address to guest physical address translation via a per-process guest OS page table (gPT), which is known as first level address translation (FLAT).
2. **gPA $\Rightarrow$ hPA:** guest physical address to host physical address via a per-VM host page table (hPT), which is known as second level address translation (SLAT).

Next we will discuss two state-of-the-art techniques used to manage and translate two-levels page tables each providing different tradeoffs.

2.1.6.1 Shadow Paging

Shadow Paging is a software-based technique used to manage two-level address translation. It creates a shadow page table (sPT) that maps the guest virtual page table (gPT) directly to the physical memory. When the guest is active the hypervisor forces the processor to use the sPT to perform the address translation. When the address needed its not present in the sPT a page fault exception is raised causing the hypervisor to intervene and update the sPT and the guest page table (gPT). The hypervisor keeps sPT updated by monitoring the gPT. There are two techniques

used by the hypervisor to keep track of the gPT. The first one is by letting the processor page fault behaving as natively [63]. As the hypervisor lets the guest write new entry to the gPT, when the guest try to access the date using the new entry the processor page faults because the translation is not present in the sPT. The hypervisor then intervene and make the corresponding change to the sPT and the gPT. The other approach is by write-protecting the gPT, i.e. the hypervisor write-protect all the gPT and anytime the guest tries to write a new entry to the gPT it results in a page fault exception. The hypervisor take control and add the pretended address to the sPT and also update the gPT. The shadow page technique adds a substantially amount of overhead [63] [64] due to the hypervisor intervention on the page faults.

2.1.6.2 Nested Paging

Nested paging also known as nested page table (NPT) is a hardware-assisted virtualization technique which reduces the overhead associated with software-based technique (shadow page tables)[63]. The NPT translate the guest virtual address to physical, and it leaves the guest in full control over its page tables with no need of the hypervisor's intervention. In the implementation of this technique used two tables when translating an address. The gPT and the NPT, the gPT owns the maps of the guest virtual address to the guest physical addressed and the NPT the map of the guest physical addresses to the physical memory. When a guest attempts to reference the physical address, a two-dimensional table walk is performed using gPT and NPT to translate the guest linear address to its mapping on the physical memory ($gVA \Rightarrow hPA$).

2.1.6.3 Dynamic Switching Paging

This technique tries to take the best of both worlds by dynamically switching the technique used to monitor the TLB misses and guest page fault on the fly. As seen before the big disadvantage of using shadow paging is the overhead caused by the hypervisor intervention when synchronizing the sPT and the gPT, while the drawback of the nested paging is the two-dimensional page walk in the address translation. The challenge in this technique is the definition of a switching police. In the implementation documented in the article [65], the switching strategy used consist in keeping an historic on TLB misses and page faults frequency. To make a decision was also create a threshold values for both TLB misses and page faults and if one of them is above the threshold, the dynamic switching paging mechanism would switch to the other mode (nested or shadow paging). If both values are under the thresholds there would not be necessary the switching if both values are over the thresholds the mechanism would use others metrics to choose the best mode to use. Another implementation of this technique known as agile paging documented in the article [66] takes thing a little bit further, differently from [65] which uses a temporal approach (the switching policy is based only in the current state of the TLB misses and

	Base Native	Nested Paging	Shadow Paging	Agile Paging
TLB Hit	Fast (VA $\Rightarrow$ PA)	Fast (gVA $\Rightarrow$ hPA)	Fast (gVA $\Rightarrow$ hPA)	Fast (gVA $\Rightarrow$ hPA)
Max. memory access on TLB miss	4	24	4	(4 $\Rightarrow$ 5) avg.
Page Table update	Fast direct	Fast direct	Slow (by the Hypervisor)	Fast direct
Hardware support	1D page walk	2D+1D page walk	1D page walk	2D+1D page walk with switching

Table 2.1: Performance comparison of the memory virtualization techniques used to manage and translate two-levels page tables.

page faults) when implementing the switching policy. Its switching mechanism implementation in temporal and spatial, i.e., it takes in consideration that some regions of an address space see far more changes than others, and some levels of the page table are updated far more often than others.

The implementation combines the best of shadow and nested paging by using: **(I)** shadow paging for its fast TLB misses response related to parts of the guest page table that remain static, and **(II)** nested paging for its fast pages updates for the parts of the guest page tables that dynamically changes. The table 2.1.6.3 show the performance comparison of the [66] implementation of the dynamic switching paging technique with the nested paging, shadow paging and the natively one.

Even with hardware and software techniques the overhead of virtualizing memory can still be high, and the dynamic Switching paging technique can reduce the virtualization overhead imposed by the traditional techniques. The infrastructure currently present in today's top architectures and hypervisor makes it easier to implement the dynamic Switching paging technique.

2.1.7 I/O Virtualization

There are a number of different applications where the support for Input/Output (I/O) virtualization it is or will be a highly desired feature. There are two key requirements that are common in all those different use-cases scenario [67]: the first one is the protected access to the I/O resources such that there is no dispute between guest OS in accessing the I/O resource while the second requirement is the ability to share the I/O resource among the multiple guest OS. Virtualizing the I/O infrastructure in a system follows the same principles used when virtualizing the CPU or memory, i.e., it uses the abstraction layer (hypervisor) to isolate and monitors the access to the I/O devices between the different guest virtual I/O devices. Furthermore, like the CPU virtualization an I/O device can be fully-virtualized where the hypervisor traps the guest OS access to the I/O devices and emulates the underlying hardware operations intended by the guest. The guest OS interface with the I/O device can be achieved either through memory-mapped I/O (MMIO) or

Port-mapped I/O (PMIO). If the I/O device is accessed through MMIO the virtualization can be accomplished by applying the memory virtualization techniques presented previously in the regions where the device is mapped. Access to PMIO are made through special instructions designed specifically for performing I/O operations. To virtualize devices that are accessed through PMIO the instructions to perform I/O operations are trapped (sensitive instruction) when being executed by the guest OS and the execution is passed to the hypervisor that will decode and emulate the instruction. To fully virtualize the I/O devices, the hypervisor needs also to be compatible with the software device drivers used in the guest OS to access the I/O device, as well as able to emulate I/O features like interrupts, data buses, and resource sharing of the physical hardware. When emulating I/O resources which generates interruption the hypervisor uses mechanism to inject interrupt into the guest OS. This can be accomplished by emulating the programmable interrupt controller (PIC). The trap-and-emulate virtualization approach may require minimal to none changes in the guest OS, but it has a low performance due to the high amount of overhead imposed by the costly interception and validation by the virtualization layer for security, isolation, data multiplexing and demultiplexing purpose.

Relying on paravirtualization is another method used to virtualize I/O resources [67, 68, 69, 70]. The goal is minimizing the hypervisor's intervention presented in the full I/O virtualization technique when the guest OS is accessing the I/O resources. The guest OS is aware of the presence of the abstraction layer on the contrary of the full virtualization technique where the guest runs as it would without virtualization. The guest OS is altered (e.g., addition of a paravirtual driver) in order to use a set of I/O type-specific APIs provided by the hypervisor to interface the I/O resources. The guest uses the paravirtual driver (e.g., added to the guest) to communicate with the hypervisor using APIs. The actual device driver that will access the I/O resources can reside in the hypervisor or in a secure device driver domain only visible to the hypervisor. Instead of emulating the PIC as done in the full virtualization method, with paravirtualization event or callback can be used to signal the guest OS [67]. This technique can remove the interaction of the PIC hardware, with the event or callback the process is direct. The hypervisor can directly hand the work to the guest OS. The problem with this approach is that the Guest OS interrupt handling mechanism must be altered. The drawback in using a paravirtualized system is that by altering the guest OS it limits the guest portability/migration and also the guest is aware of the virtualization layer, and so, it fails the Popek and Goldberg equivalence requirement. Even with the pitfall shown above the paravirtualization technique is used by the major hypervisor vendors [71] when virtualizing the I/O devices due to a better performance and the reduction of complexity and cost of the I/O resource virtualization comparing to the emulation technique.

Other implementations uses direct assignment, a method that can be used when a I/O resource can be owned by an specific guest OS, i.e., a particular I/O device is assigned to be used by an specific host removing the hypervisor's intervention. In doing so, the guest OS is allowed

to interact with the I/O device as it would be in a native system. This method can simplify the hypervisor's implementation by moving the I/O device drivers from the hypervisor to the guest. This method is only appropriate for some implementations since the hypervisor can only allocate as many devices as they are physically available in the platform. Furthermore, even with the direct assignment of an I/O resource to the guest OS, the hypervisor may still have to intervene in operations such as routing the interruption to the guest [71] and/or the guest address translation. This is required because the I/O works with the host physical address (hPA) and the guest OS perception of physical address is the gPA and the hypervisor needs to intervene to apply memory virtualization techniques to solve the memory access.

The three techniques shown above are the most common ways of implementing I/O resources virtualization. The article [72] shows an interesting implementation of I/O resources virtualization known as Hypervisor-bypass, which takes advantage of the features presented in modern high speed devices to allow time-critical operations to be carried out directly between the guest OS and the I/O device, and so, removing the overhead caused by hypervisor intervention and at the same time maintaining the system's integrity and isolation intact. In this implementation only the time-critical operation are bypassed while the remaining are assisted by the hypervisor. This implementation was based on the OS-bypass I/O technique. The OS-bypass I/O is a mechanism developed to allow processes in the user space access I/O resources (traditionally accessible in kernel mode). It provides user-level-communication APIs that allows the user space processes to access the I/O devices without the need of a context switching. To support accesses from multiple user processes is used I/O pages that are mapped directly in the address space of the process. The hypervisor-bypass follows the same principle as OS-bypass I/O where the guest OS in the hypervisor-bypass behaves as the OS-bypass user space application and the hypervisor as the OS kernel in the OS-bypass. The hypervisor provide the guests mechanisms to directly access the time-critical I/O resources and the others resources are managed using one of the three mechanisms exposed above (paravirtualization, emulation or direct assignment).

2.1.7.1 DMA Protection

The majority of modern I/O devices utilizes direct memory access (DMA) to move data between the host's main memory and the device's on board buffers. The usage of DMA is to reduce the overhead on the CPU when using the I/O devices, allowing it to not be occupied during the entire read/write operation. To initialize a DMA operation the device driver in the OS creates DMA descriptors that refer to region of memory. Each descriptor typically includes the address, the length and some device-specific flags. When created the device driver passes the descriptor to the device to transfer data to or from it and when the operation is finished an interruption is used to signalize the CPU. This mean that in a virtualized system the guest OS can program the I/O to uses the DMA to read or write to any address, which can compromise the system's security

its isolation capability. When the I/O is virtualized using the emulation technique, this problem is solved by the hypervisor since all the sensitive instructions are trapped to the hypervisor that can monitor the address passed by the guest. The same happens when the I/O is virtualized using paravirtualization. Since the device driver is in the hypervisor or in a secure device driver domain only visible to the hypervisor, the guest is altered to communicate with the hypervisor when accessing the I/O device. This gives the hypervisor means to monitor the addresses used by the guest OSes. The problem arises when the guest can directly access the I/O resource, e.g., using direct assignment method as in this case the access to the device driver is completely owned by the guest OS. The majority of modern architectures possesses hardware such as ARM SMMU[73], Intel IOMMU[74], AMD IOMMU[75] that operate as the input/output memory management unit (IOMMU) to assist and facilitate the I/O devices access to the memory. There are several advantages using the IOMMU to assist in I/O devices virtualization, being the most relevant in this scenario the memory protection.

The IOMMU allows all the I/O devices to be mapped to a page table in the main memory and on every memory access by the I/O device the IOMMU consults the I/O device's designated page table to determine if the access should be allowed or blocked. Taking in consideration this capability, the hypervisor can make sure the well being of the device being assigned to particular guest OS by allowing the device to only access memory mapped to the guest where the device is assigned. The IOMMU must only be accessed by the hypervisor. To architectures where there is no hardware support to the I/O devices memory access the protection must be ensured using software techniques. This technique is carried out by the creation of the actual DMA descriptor, i.e. once before the guest OS uses the I/O device the hypervisor will take ownership of the driver's DMA descriptor region and the driver's status region, revoking write permissions from the guest. This prevents the guest from independently creating or modifying DMA descriptors, or modifying the status region [76]. This can be accomplished by mapping the descriptor memory region in the hypervisor's memory region. The software implementation is described in greater detail in [77].

3. Related Work and Background

In this chapter it is surveyed previous and current state-of-art of Virtualization techniques. Since the main focus of this work is on the embedded domain it will be pictured in this section the current state-of-art of virtualization of embedded devices. Hypervisors designed for general purpose virtualization cannot be directly adopted for ES. This chapter starts by showing well-known hypervisors widely used for cloud computing that were not ported into the embedded world according to the evolution of embedded devices and the necessity of virtualization to manage the increasing complexity of today's embedded applications. We start by describing Xen, a well known type-1 virtualization platform that have made its way from the GPOS to the Embedded devices with real-time requirements, then is described the KVM hypervisor, a major hypervisor that have also found itself ported into embedded devices.

Then XtratuM will be presented as a different hypervisor implementation compared to the hypervisors presented before since it was originally designed for real-time applications. Basically, this the section describes the XtratuM architecture and to finalize the chapter is presented another hypervisor designed for embedded applications.

To finish the chapter is depicted the architectural features presents in ARM processors such a TrustZone and the Virtualization Extension that served as backbone for LTZVisor's implementation. It is also surveyed some state-of-art of Virtualization techniques assisted by ARM TrustZone.

3.1 Embedded Hypervisors

Since the early 2000s we have seen the virtualization technology gaining ground in the into the embedded domain, bringing new solutions and expanding the scope and capabilities of the embedded applications. Bellow is presented some of the many embedded virtualization architecture available.

3.1.1 Xen

Xen is a type-1, open-source hypervisor, for both 32 and 64-bits processor architecture [78] created at the University of Cambridge. Xen was first introduced as a virtualization solution for

x86 architectures and since then the platform has been ported to the majority of major processors technology such as Itanium, Power PC, and ARM [79]. Xen supports Linux, NetBSD, FreeBSD, Solaris, Windows, and other common operating systems as guests running on top of it. At the heart of the Xen virtualization platform is the Xen hypervisor. It runs in the highest privilege level and controls the system's hardware. Xen hypervisor design aims to create a lightweight hypervisor that performs only the primary activities such as scheduling and virtualizing the CPUs and memory. The remaining part of the TCB are held in a secure container outside the hypervisor know as the Dom0 (domain 0). Dom0 is usually a modified Linux kernel or a privileged guest laying on to of the hypervisor that has special rights to access physical resources as well as interact with the other guests running in the system. The Dom0 VM is a requirement in the Xen virtualization platform as it possesses all the hardware device drivers and plays a major role in the creation of the unprivileged guest. The guest OSes are held in the less privileged domain know as DomU (DomainU). Xen hypervisor supports multiples virtualization approaches for the DomU guests, including full hardware virtualization and paravirtualization. The DomU paravirtualized guest are know as PV guests and the full virtualized guest are referred to as HVM guests. The advantages of moving services from the hypervisor to the dom0 were listed previously in the Microkernel subsection. Similar benefits are achieved by the Xen hypervisor architecture design and the success of this technique as led to studies beginning to break up the Dom0 from a single guest into a series of mini-OS guests each with a specific purpose and responsibility, which drives better performance and security in a virtualization environment [80]. Dom0 gives access for unprivileged guests to the hardware with paravirtualized backend drivers. Moreover, the modified guests (unprivileged domains) get access to a set of virtual devices using the paravirtualized frontend drivers. As an effort to improve overall performance and reduce the load on the Dom0 Guest, Xen hypervisor uses pass-through technology [81, 82] which allows guest domain to speak directly with a specific piece of hardware directly without having to send communication to and from the Domain0. One of the Xen hypervisor's main tasks is abstracting the system's physical CPU, creating Virtual CPU (VCPU). [83] states that "the main scheduling abstraction in Xen is the virtual CPU". By default Xen uses the credit scheduler whose features and behavior are well documented in [83, 84, 85, 86]. [87] list others scheduling available for the Xen hypervisor such as Real-Time Deferrable Server(RTDS) for soft and firm real-time tasks and ARINC653 for hard real-time tasks. Many other scheduling algorithms implementation [83, 84, 88, 89, 90, 91, 92] are used in the Xen hypervisor, leaving the selection policy to the intended application.

3.1.2 KVM

Kernel-based Virtual Machine (KVM) is a partially monolithic type-2 hypervisor that supports both full virtualization as well as paravirtualization. KVM is an established system virtualization solution, implemented as a software-only driver running within the Linux Kernel, which effectively

turns the Linux kernel into an hypervisor [93]. This approach takes advantage of the existing mechanisms within the Linux kernel, such as the scheduler, and memory management. Like Xen, KVM was first developed for x86 processors and since it has been supported S/390, PowerPC, IA-64 and ARM processors. KVM support a variety of guest including Linux, Solaris, OS X and Windows. The virtualization design approach used in KVM it is very relevant in the embedded world. One may think that when virtualizing embedded devices it is more attractive the adoption of a type-1 hypervisor for its independence of a host environment and the resource limitations in the embedded devices, but this type-2 hypervisor suits well in the embedded world due to the vast number of embedded devices already supported by support Linux. Having KVM as a module in the Linux OS makes KVM a highly portable hypervisor. The KVM system architecture implementation had to be redesigned for the ARM architectures by introducing the hypervisor mode (in order not to cause confusion between the ARM hypervisor mode and the hypervisor virtualization layer it is used Hyp mode to name the hypervisor mode). The Linux kernel runs in the supervisor mode that is a less privileged mode than the hyp mode, which means that in order to design the KVM hypervisor the entire Linux kernel would had to be redesign to run in the hyp mode (see details in [94]). The design approach used is known as split-mode virtualization, because it splits the core hypervisor so that it runs across different privileged CPU modes to take advantage of the specific benefits and functionality offered by each mode. Basically, the hypervisor is split in two modules:

1. The Lowvisor runs in the hyp mode to leverage the ARM hardware virtualization support enabled by hyp mode. The lowvisor manages the different execution contexts through appropriate configurations of the hardware, enforced protection and isolation between different execution contexts, world switches between the hypervisor and the VM context and the virtualization trap handler.
2. The Highvisor is embedded into the host kernel as a linux driver. Being in the kernel space allows it to directly leverage existing Linux functionality such as scheduler, memory management while easily implementing high level functionalities.

Using Linux APIs simplifies the hypervisor's highvisor implementation but at the same time because the hypervisor is split across supervisor and Hyp mode, switching between a VM and the highvisor involves multiple mode transitions (VM $\Leftrightarrow$ Lowvisor (hyp mode) $\Leftrightarrow$ Highvisor (supervisor mode)). For instance, in the arise of an hardware interrupt, a stage-2 page fault, or a sensitive operation during the KVM guest execution, the CPU will trap to the lowvisor (hyp mode) that examine and perform the necessary task and then switches back resuming the VM execution. The intended task could be performed by the lowervisor or the highvivisor (supervisor mode), for example, the emulation is performed by an application in the user space, usually KEMU, with the lowvisor switching to the highvivisor to perform the necessary emulation. Details in implementation

of KVM to ARM architectures can be found in [94, 95] and the documents describes in details the implementation of CPU, Memory and I/O virtualization using methods described in the previous section.

KVM is the mainline Linux ARM hypervisor and the first system that can run unmodified guest operating systems on ARM multicore hardware. Its split-mode virtualization makes it possible to use ARM hardware virtualization extensions while leveraging Linux kernel mechanisms and hardware support [94]. However, there remains concerns regarding the responsiveness of the real-time tasks running on top of a virtual machine. Many studies have tackle this issue proposing solutions [96, 97, 97, 98, 99] to enhance the platform with real-time capabilities.

3.1.3 XtratuM

Different from the hypervisors presented so far, XtratuM was originally designed to meet real-time requirements, specifically as a substitution of the RTLinux to achieve temporal and spatial requirements[100, 101]. XtratuM is a type 1 hypervisor that uses paravirtualization to abstract the hardware to the guest OSes. By using paravirtualization part of the XtratuM guests hardware abstraction layer (HAL) are replaced with hypercalls, which allows the guests to access privileged functionalities and at the same time helps the hypervisor in the management of the guest access to those functionalities. [102] states that main idea behind of the design of the architecture of XtratuM is to virtualizes the minimal possible parts of the hardware to achieve the execution, in a concurrent way, of several OSes, where at least some of them could be real-time kernels. XtratuM virtualizes the main hardware in the system and its internal architecture includes: memory management, scheduling (fixed cyclic scheduling), interrupt management, clock and timers management, partition communication management, health monitoring and tracing facilities [103]. XtratuM allows the definition of two types of VM, system VM(also know as supervisor VMs) and normal VM. System's VMs are allowed to manage and monitor the state of the system and other partitions. The purpose of the health monitoring is to discover errors at an early stage and try to solve or confine the fault subsystem in order to avoid a failure or reduce the possible consequences.

Time and space partitioning (TSP) based architecture has been identified as the best solution to ease and secure reuse, enabling a strong decoupling of the generic features to be developed, validated and maintained [100, 103, 104]. XtratuM is used as an enabling technology for TSP developments, being designed to meet real-time requirements it is an excellent solution for embedded virtualization, by virtualizing only the necessary hardware in the system, leading to a small footprint. It is classified as a nanokernel architecture. Which conceive a trustworthy execution environment to its guests by enforcing: (I) a strong spatial isolation by executing the hypervisor and the guest in different CPU privileges and address space, (II) temporal isolation achieved by using a fixed cyclic scheduler as well as (III) robust communication mechanisms between the

guests, communication between guests are achieved by using specific services provided by the hypervisor. The hypervisor implements the link (channel) between two ports or more ports based on the ARINC-653 standard [105]. A detailed description of the XtratuM hypervisor design and implementation is provided in [106, 107].

3.1.4 Xvisor

eXtensible Versatile hypervISOR (Xvisor) is a type-1 monolithic hypervisor that supports both full virtualization and paravirtualization. It aims to provide a lightweight hypervisor that can be used within embedded systems with less overhead and small memory footprint. Xvisor primarily provides fully virtualized guest and provides paravirtualization in the form of optional VirtIO devices [108, 109]. Xvisor provides high performance and low memory footprint virtualization capabilities for various ARM architecture (with and without virtualization extensions) and for other CPU architectures including x86. The hypervisor allows the execution of multiple guest VMs, to each guest is assigned one or more virtual CPUs (vCPU) by the Xvisor scheduler and all background processing for device drivers and management purposes are run on the Orphan vCPUs with highest privilege. Xvisor vCPU scheduler offers various scheduling policies (e.g., fixed-priority[110], round-robin[111]). Virtualization of peripheral devices is achieved via emulation with pass-through access available for some devices. When compared to Xen or KVM, Xvisor differs from them by having a single software layer where all virtualization related services are provided [108, 112]. For example, KVM depends on Qemu for device emulation and Xen depends on dom0 for I/O. Thus, the Xvisor's context switches are lightweight, resulting in instruction trap handling, host interrupts and guest I/O events. Also, the Xvisor's scheduler is per-CPU and does not do load balancing [113] for multiprocessor systems. Instead, load balancing is performed by a distinct entity in Xvisor.

3.2 ARMv7-A architecture

The architectural simplicity of ARM RISC processor have allowed ARM based devices to have a very low power consumption, with high performance at a reduced silicon cost. The architecture have evolved significantly since its introduction, eight major versions of the architecture defined to date, denoted by the version numbers 1 to 8. Among them, version 7 is the relevant in this project. ARMv7 provides three profiles: ARMv7-A, ARMv7-R, ARMv7-M[114] details the difference between the different ARMv7 profiles and the processor used in this project inhabit in the profile A. ARMv7-A architectures supports both the ARM and Thumb instruction set. The ARM instruction set is a set of 32-bit instructions providing comprehensive data-processing and control functions.

Developed with a subset of ARM instruction set functionalities, the Thumb instruction set is 16-bits wide that provides significantly improved code density but with some performance reduction compared to the ARM ISA. A processor that support both ISA can change between the two when necessary. The continuous evolving nature of ARM's architecture led to incremental changes to the architecture and the introduction of new features. These changes can happen at the ISA level, a phenomenon titled ISA Extension or in the architecture level called Architecture Extension.

ISA Extension ranges from Jazelle, ThumbEE, floating-point (also known as VFP extension), Advanced Single Instruction Multiple Data (SIMD) [115].

ARMv7 implements the following **Architecture Extension**:

1. Security Extensions

Also known as TrustZone extension, when implemented, the Security Extensions integrate hardware security features into the architecture, to facilitate the development of secure applications. Due to its high relevance to the project, ARM Security extension is detailed in the next section.

2. Multiprocessing Extensions

VE provides a set of features that enhance multiprocessing functionality and additional registers are included in the SoC's core hardware to support virtualization of the cores.

3. Large Physical Address Extension (LPAE)

This feature is only included in architecture that implements the multiprocessing extension and the extension adds a new translation table format, named the Long-descriptor format. LPAE comes with a 3-level page table format, allowing up to 40-bit physical address space. The architecture in which the implementation does not include the Large Physical Address Extension uses the short-descriptor format.

4. Virtualization Extensions (VE)

Virtualization extension is only included in the architectures that implement the security extension, LPAE and the multiprocessing extension. When implemented, the virtualization extensions provide a set of hardware features that support virtualization, many features of the architecture are extended to integrate with the virtualization extensions. ARM VE provides 3 levels of execution: EL0, user mode, EL1, kernel mode, and EL2, hypervisor mode as shown in the figure 3.1.

ARMv7 architectures provides more execution modes than the ones shown in the figure 3.1 and the complete list of the processors execution modes are listed as followed:

- (a) User Mode: An operating system runs applications in user mode to restrict the use of system resources. Software executing in user mode executes at PL0. Execution in

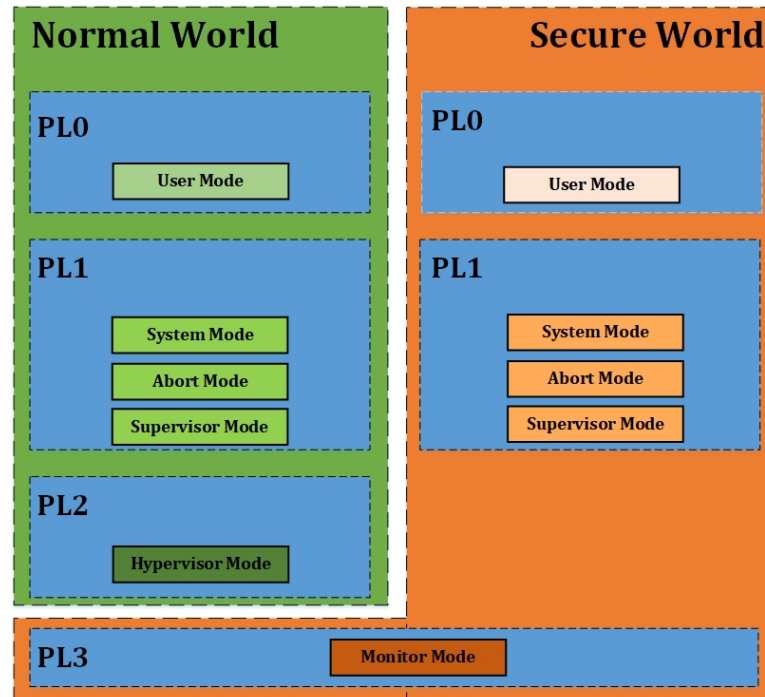


Figure 3.1: TrustZone privilege levels.

user mode is sometimes described as unprivileged execution. Application programs normally execute in user mode.

- (b) **Supervisor Mode:** this is the default mode to which a supervisor call exception is taken. Executing a SVC (supervisor Call) instruction generates an supervisor Call exception, that is taken to supervisor mode. A processor enters supervisor mode on Reset.
- (c) **System Mode:** software executing in system mode executes at PL1. System mode has the same registers available as user mode, and is not entered by any exception.
- (d) **Abort Mode:** abort mode is the default mode to which a data abort exception or prefetch abort exception is taken.
- (e) **Undefined Mode:** undefined mode is the default mode to which an instruction-related exception, including any attempt to execute an UNDEFINED instruction, is handled.
- (f) **FIQ Mode:** FIQ mode is the default mode to which an FIQ interrupt is handled.
- (g) **IRQ Mode:** IRQ mode is the default mode to which an IRQ interrupt is handled.
- (h) **Hypervisor Mode:** hypervisor mode is the non-secure PL2 mode, implemented as part of the virtualization extensions, this mode is entered on taking an exception from non-secure state that must be taken to PL2 and it can be caused by an hypervisor call or an hypervisor trap implemented by the virtualization extension.

- (i) **Monitor Mode:** monitor mode is the mode to which a Secure monitor call exception is taken. In a PL1 mode, executing an SMC (secure monitor Call) instruction generates a secure monitor call exception. Monitor mode is a Secure mode. Software running in monitor mode has access to both the Secure and Non-secure copies of system registers. This means monitor mode provides the normal method of changing between the secure and non-secure security states.

Mode changes can be made under software control, or can be caused by an external or internal exception. The VE extension introduces a new instruction, HVC, to switch between kernel mode and hypervisor mode, from reset, the HVC instruction is UNDEFINED in Non-secure PL1 modes, meaning entry to Hyp mode is disabled by default. To permit entry to Hyp mode using the Hypervisor Call exception, Secure software must enable use of the HVC instruction by setting the Secure Configuration register (SCR).

An exception causes the processor to suspend program execution to handle an event, which can be generated by internal and external sources. Usually, when an exception is taken the processor state is preserved immediately, before handling the exception so that when the event has been handled, the original state can be restored and program execution resumed from the point where the exception was taken. The exception handlers are registered in the Vector Table that is a set of exception vector comprised in consecutive word-aligned memory addresses, starting at an exception base address. The security extension implementation includes three vector table: the Non-Secure world Vector Table and the Secure world Vector Table to independently handle the worlds exceptions and the Monitor Vector Table used for the exceptions routed to the monitor mode, as shown in the table 3.1.

Vector Tables			
Offset	Secure	Non-secure	Monitor
0x00	Reset	Not Used	Not Used
0x04	Undefined Instruction	Undefined Instruction	Not Used
0x08	Supervisor Call	Supervisor Call	Secure Monitor Call
0x0C	Prefetch Abort	Prefetch Abort	Prefetch Abort
0x10	Data Abort	Data Abort	Data Abort
0x14	Configurable ¹	Not Used	Not Used
0x18	IRQ Interrupt	IRQ Interrupt	IRQ Interrupt
0x1C	FIQ Interrupt	FIQ Interrupt	FIQ Interrupt

Table 3.1: ARMv7-A Vector Tables.

5. Performance Monitors Extension

Most processors nowadays have special, on-chip hardware that monitors micro architectural events like elapsed cycles, cache hits, cache miss, etc. It is a subsystem which helps analyzing how an application or operating systems are performing on the processor. ARM Performance Monitoring Unit (PMU) provides counters to gather statistics on the operation of the processor and memory system. Each counter can count any of the events available in the processor. It also provides a single 32-bit cycle counter with support for scaling and filtering on the processor mode and security state. If an implementation includes the virtualization extensions and also includes the performance monitors extension, then it must implement PMUv2. In PMUv2, an implementation that includes the virtualization extensions, non-secure software executing at PL2 can trap any attempt by the Guest OS to access the PMU. This means the hypervisor can monitor the access to the PMU and dictate its correct usage.

3.3 ARM Security Extension - TrustZone

Design security is often an afterthought. But, with the regularity of security breaches impacting an array of industries, it is now more of an imperative to build security into design early on [116]. While software-based security mechanisms are deemed to be cost effective and easy to implement and update, this technique's success relies entirely on the underlying software layer implemented. In most cases this software layer is an operating system, hypervisor and in the embedded world a real-time application or OS. Meaning that this underlying software layers will have to implement means to guarantee the system's reliability and trustworthiness. Adding security mechanisms within the management layer (OS or RTOS) increase the system overall complexity and size, a condition not always desired when dealing with resources constraints of embedded devices. This has led to exploration of other security approaches, in this case hardware-based security mechanisms. Shipping the security mechanism implementation to hardware relieves the software of time and space consuming security tasks and responsibilities.

Hardware-based security uses a device's on-board security to perform security measures to ensure the correct functioning of the system. Hardware-based security is, indeed, more robust than its software-based counterpart. It is self-contained and does not require the help of any additional software. Therefore, it is essentially free from the possibility of contamination, malicious code infection, or vulnerability. Hardware-based security is achieved by implementing a well known security concept: a Trusted Execution Environment (TEE). The TEE is an isolated environment

¹The vector at offset 0x14 in the Hypervisor vector table is used for exceptions that cause entry to Hypervisor mode. The Hypervisor Configuration Register (HCR) provides configuration controls for virtualization, including defining whether various Non-secure operations are trapped to Hypervisor mode. Various Non-secure operations are trapped to Hypervisor mode.

that runs in parallel with the operating system, providing security for the rich execution environment (REE). The wide acceptance of TEE as led board manufacturer to include tools to enable the easy development and deployment of applications targeting the creation of a TEE.

The ARM TrustZone approach aims to enable a trusted computing within the embedded world by extending the security infrastructure throughout the system design, enabling an end-to-end security solution. TrustZone technology is often used to enable trusted boot and run a trusted OS, creating a trusted execution environment. The security is achieved by partitioning all the SoC's hardware and software resources so they are accessible in one or two worlds (two virtual cores)[117]: the Secure World and the Non-Secure (Normal) World as illustrated in the figure 3.2. The secure world is used to execute the TEE with full access to the system's hardware and software and the normal world is used for the rest of the subsystems with access only to the non-secure resources usually a REE. The TrustZone extension allows a single CPU to run both secure and non-secure code, which is achieved with the assistance of a robustly context switch mechanism known as the Monitor mode.

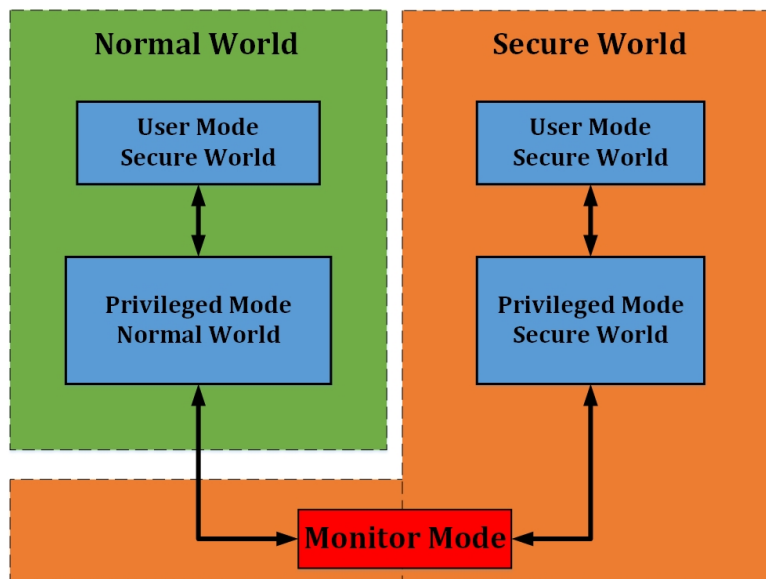


Figure 3.2: TrustZone General Architecture adapted from [117]

3.3.1 Context-Switching

By allowing the same CPU to execute both secure and non-secure code, the TrustZone extension allows execution switching between worlds through the monitor mode. TrustZone provides means for both secure and non-secure world to access the monitor mode. By executing the Secure Monitor Call (SMC) instruction the CPU traps to the monitor exception handler that performs the context-switching. The role of the monitor handler software in a design is to provide a robust gatekeeper which manages the switches between the Secure and Non-secure processor states.

In most designs this functionality will be similar to a traditional operating system context switch, ensuring that state of the world that the processor is leaving is safely saved, and the state of the world the processor is switching to is correctly restored[118]. As a security measure, the monitor mode entry from the non-secure world is tightly controlled by the secure world application. Monitor mode entry can also be achieved via interruption and external aborts. The secure World can also access the monitor mode by directly writing the register that holds the CPU current privilege mode. This functionality its not available to the non-secure software due to the register privatization, i.e., only accessible when the CPU is in secure state.

3.3.2 Interruption

ARM processor that implements the TrustZone extension comes with the interrupt controller that supports the interruption grouping, i.e., the interrupt controller allows each interrupt to be configured as secure or non-secure. Secure interrupts are usually handled as Fast Interrupt Request (FIQ) (Group 0) and non-secure as Interrupt Request (IRQ) (group 1). The interrupt controller also allows secure interrupt to be handled as IRQ but this use case scenario breaks the interrupt isolation between worlds. Dedicating FIQ to handle the secure interrupts and IRQ for non-secure debilitate the non secure interruptions to interrupt the secure world execution and that way it isolates the secure world form non-secure interruptions. ARM recommends using FIQ to handle secure interruptions, which is not nested, have the highest priority among all interruptions and also has lower latency [119] compared to the IRQ interruption handling. This particularity makes FIQ handling highly recommended to handle secure interrupts. TrustZone-enabled processor implements three sets of exception vector tables. One of these tables is for the normal world, one is for the secure world, and the other is for monitor mode, meaning that each interrupt can be configured to be handled in one of three vector table, i.e., an interrupt can be configured to be: secure interrupt, non-secure interrupt or be interrupt routed to monitor.

In a multiprocessing system, this feature works equally and independent for each physical CPU available, meaning that each processor can be configured to handle interruptions independently form each other.

3.3.3 Processor Configuration

ARM processors possess a set of co-processors used to extend the processor instruction set, accessed through a group of dedicated ARM instructions that provide a load-store type interface. This co-processors are used to configure a set of functionalities presented in the SoC, for example, the co-processor 15 (CP15) is used to configure the overall system control and configuration of the processor, cache configuration and management, memory Management Unit (MMU) configuration and management and the system performance monitoring [120]. The security is maintained

by allowing only secure access to sensitive co-processor registers, and so, only the trusted secure world application can manage the SoC sensitive configurations. In a multiprocessing system when a co-processor register is private to each CPU it means that there are multiple copies of co-processor registers (banked in the hardware) and co-processor registers where there is only one copy, these are shared between the processors (global registers). The configurations chosen by the secure software, by changing the co-processor's registers, directly affects the non-secure world applications. CP15 holds various highly valuable registers in the TrustZone architecture including the Secure Configuration register (SCR) that allows the world context to switch changing the state of the NS-Bit (secure NS-bit=0 and non-secure NS-bit=1). The NS-Bit can only be set in the monitor mode and the world context switching is performed when the processor leaves the monitor mode.

3.3.4 Level One Memory System

ARM processors that implements the security extension provides two virtual MMU for each physical core, one virtual MMU for each world, meaning that each world possesses independent control over their address mapping. ARMv6 and ARMv7 L1 translation table descriptor format includes an NS field used to access the physical mapping associated with the associated with that table descriptor, i.e., a software running in the secure world uses the NS-bit field in the table descriptor format to access the secure or the non-secure address mappings. The non-secure world can only access the non-secure mappings and when running in non-secure the NS-bit in the address format is always 1. The design also supports data on the cache from both worlds simultaneously, meaning that in a context switching it is not required to flush the cache.

3.3.5 SoC Partitioning

ARM provides mechanism to separate the SoC's hardware resources by implementing transaction checking in the system and peripheral buses. The ARM Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in SoC designs. It facilitates development of multiprocessor designs with large numbers of controllers and peripherals. The AMBA buses used in the ARM processors that implements the security extension where designed with mechanism that allows the the control of every transactions, ensuring that no secure world resources are accessed by non-secure world components. AMBA also includes an input signal for each peripheral connected to the bus and this signal is used to determine the security configuration of the peripheral (secure or non-secure). AMBA bus signals are dynamically controlled by a secure peripheral that allows the secure world do configure the peripheral security, and it is known as TrustZone Protection

Controller (TZPC). It allows the secure world to dynamically reconfigure SoC's peripherals security access at run-time, meaning that a particular peripheral can be secure in an instance and non-secure in another.

3.3.6 Memory Partitioning

Protecting the memory region access is one of the main concern when implementing isolated systems. In the previous chapter some methods used to virtualize memory were presented in the memory virtualization section. The techniques presented in this subsection aims to provide a secure and isolated execution environment for the different subsystems (in this case virtual machines) within the system. Those techniques comes at a price, the necessity of an underling software layer that monitors the memory access by different subsystems thus, adding overhead to the entire system.

The mainstream hardware-assisted memory virtualization relies on the MMU support for the 2-level address mapping and translation. This feature is a key enabler to run unmodified partition Oses, and also to implement isolation between partitions. ARM TZASC is an Advanced Extensible Interface (AXI) [121] component which partitions its slave address range into a number of memory regions. TZASC allows the partitioning of the memory into multiple segments, which can be exploited to guarantee spatial isolation between the non-secure VM and the secure one, by adequately configuring the security state of the memory segments of respective partitions. The non-secure VM should have its own memory segment(s) configured as non-secure, and the remaining memory as secure. Any attempt to access a secure memory region by the non-secure world raises an exception that is trapped in by monitor. This solution is attractive to TrustZone-enabled SoCs with MMU support for the straightforward approach and it is a requirement for the TrustZone-enabled SoCs which are not VE-enabled and only capable of providing MMU support for single-level address translation.

For on-SoC memory, solutions like TrustZone Memory Adapter (TZMA) is available. The TZMA allows a single memory to be partitioned in two region where the lower part is secure and the upper non-secure. TZMA only allows the memory to be partitioned in two regions, with sizes always as a multiple of 4KB and controlled by the TZPC peripheral.

3.4 TrustZone-Based Virtualization

ARM TrustZone can be viewed from two prospectives, as virtualization solution or as mechanism to implement functionality similar to Trusted Platform Modules (TPM) [122]. Pinto et al. [21] have shown an implementation where TrustZone is exploited to implement the basic building blocks of a Trusted Execution Environment (TEE) as a lower priority thread of a RTOS with the RT

properties of the RTOS remaining practically unaffected, by only inducing a small overhead on latency. When regarded as a virtualization solution, TrustZone technology virtualizes a physical core as two virtual cores, providing two completely separate execution domains. The non-secure world acts as a virtual machine (VM) under the control of a hypervisor running in the secure world side. ARM TrustZone, when adequately exploited, opens up a number of opportunities for (real-time) virtualization, despite some researchers still arguing that perceiving TrustZone as a virtualization mechanism is very limiting and ill-guided [123, 124].

With the continue spreading of the TrustZone in ARM devices, one of the main goals of the LTZVisor project is to demonstrate that when adequately exploited **TrustZone is the Key**[125] to extend the virtualization technology to the variety of ARM devices that implements the security extension, ranging from applications processors to the smallest of microcontrollers.

3.4.1 Related Work

This section surveys previous and the current state-of-art of TrustZone-Based Virtualization techniques. The first hypervisor described is the SierraVisor, an open source hypervisor designed to take advantage of the ARM TZ and VE, it is presented as a Dual-OS Infrastructure for Mixed-Criticality Systems on ARMv8 Platforms, which uses the ARM trusted firmware as the monitor of two OSes running in each world. Then the section is finished with the vTZ, that is a technique used to maximize the system security and reliability capabilities by abstracting the physical TrustZone extension from the hypervisor.

3.4.1.1 SierraVisor

The SierraVisor is a TrustZone-assisted hypervisor that allows multiple operating systems, such as Linux, BSD, Android, and other real-time operating systems to run concurrently on most ARM device [126], providing both paravirtualization and full hardware virtualization. The hypervisor leverages hardware security extensions included in ARM TrustZone-enabled devices to run the guest operating systems concurrently. Because of enhancements built for TrustZone technology, SierraVisor can operate efficiently with minimal impact on performance. In addition, each high-level operating system instance can still access a secure operating system (or TEE [127]) for sensitive applications and data. It offers robust multicore management, offering both symmetric and asymmetric multiprocessing for managing system utilization.

3.4.1.2 Dual-OS Infrastructure for Mixed-Criticality Systems on ARMv8 Platforms

With the proliferation of ARM devices in the mobile and embedded world and the introduction of VE in the ARM devices Spyridakis et. al. [128] proposed the use of a Dual-OS infrastructure with a GPOS, such as Linux, coexisting with other operating systems usually an RTOS to efficiently

leverage ARM multicore hardware platforms in mixed-criticality use cases. The ARM Trusted Firmware (ATF) is used as the layer that uses the TrustZone security extensions to isolate resources into the two worlds. ATF [129] is a secure world software implementation provided as a reference firmware infrastructure, and it meant to be a modular framework for handling the boot procedure, interrupt management and world switching on all available SoC cores. To test the proposed infrastructure is used the the ARMv8 Juno development platform running the Linux kernel with the KVM hypervisor in the non-secure world while the secure world runs an RTOS and the ATF as the monitor.

3.4.1.3 vTZ

vTZ is a technique used to virtualize the ARM TrustZone extension as a solution to remove the security accountability on the hypervisor, i.e., by virtualizing the TrustZone extension, vTZ is capable of maintain the system's reliability even when the hypervisor is compromised. vTZ is based on ARM architectures that implements the TrustZone extension together with the virtualization extension (VE). vTZ [130] provides transparent virtualization of TrustZone while still maintaining strong isolation among guest trusted execution environment (TEEs) with minimal software TCB. The key idea is separating functionality from protection, i.e., vTZ relies on the normal world hypervisor to virtualize functionality of guest TEEs but leverages the physical TrustZone to enforce protection. To enforce protection using the TrustZone extension the memory management and the world switching modules are implemented in the secure world. vTZ provides a set of Constrained Isolated Execution Environments (CIEEs) running in privileged mode, which is used to implement some key logic that emulate the TrustZone functionalities used by the hypervisor. The CIEEs are isolated from each other as well as from the hypervisor, therefore removing them from the TCB since a compromised CIEE cannot affect the security of other CIEEs or guest TEEs. vTZ ensures the control and data flow integrity for code fragment within each CIEE by applying mechanism like Control Flow Locking (CFLock), used to provide non-bypassable hook for CIEE and the secure world modules ensuring the execution conclusion of the CIEEs. CIEEs and the hypervisor are maintained in the TrustZone non-secure world. Resources virtualization are easier with the ARM TZ extension, since it provides mechanisms to easily configure the systems peripherals for both secure and non-secure world. vTZ take advantages of this mechanisms to virtualize system's resources. vTZ also provide various VM management operations including VM suspending and resuming while preserving the security properties. vTZ was implemented based on Xen-ARM and the performance evaluation shows that the average applications overhead introduced by vTZ is 3% compared to Xen [130]

4. System Specification

In this chapter is described the LTZVisor's specification. It is first presented the hypervisor's architecture followed by the description of the multicore extension. Afterwards, it is listed the assets that influenced the adoption of Linux and FreeRTOS as the non-secure and secure guest. Later, it is given an overview of the i.MX 6Quad platform, and described the component used in LTZVisor's implementation.

4.1 LTZVisor

As a TrustZone-assisted hypervisor, LTZVisor relies heavily on hardware support offered by the TrustZone extension to create a secure environment to run its privileged trusted world, isolated from the unprivileged normal world. The design of LTZVisor follows three principles[125]:

1. **The principle of minimal implementation:**

In an effort to reduce the system's attack surface, LTZVisor relies on TrustZone to carefully design and configure its components with the goal of minimizing the system's TCB.

2. **The principle of least privilege:**

The security extension by design guarantees different level of privilege within the system as shown in the figure 3.2. This feature is exploited to implement a well-layered virtualization mechanism, promoting privileged execution and hardware-based isolation of the secure world from the non-secure world.

3. **The principle of asymmetric scheduling:**

Adopting the hypervisor's scheduling policy is a big issue in the presence of real-time requirements. LTZVisor opts to an asymmetric scheduling policy, where the secure environment has a higher execution privilege than the non-secure. By having the secure environment with an higher execution priority, this solution will ensure that timing requirements are met even with the CPU being virtualized.

4.1.1 General Architecture

The virtualization provided by LTZVisor relies on the two virtual execution environments created by the TrustZone hardware extension. The secure world host the privileged VM, while the non-secure world host the non-privileged VM. Figure 4.1 depicts the proposed virtualization architecture. In the figure, three main software components can be identified: the hypervisor, the secure VM and the non-secure VM.

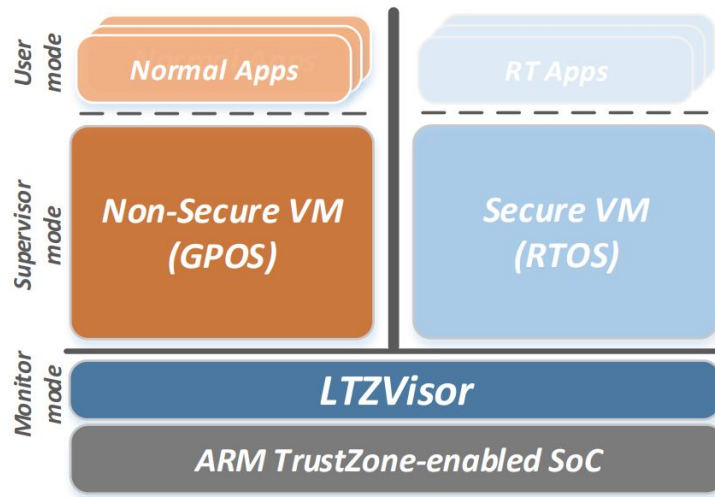


Figure 4.1: LTZVisor General Architecture.

By running with the highest privilege, LTZVisor has full control over all hardware resources. Thus, it is LTZVisor responsibility to setup each VM's memory, interrupts, assigned devices, as well as managing the VMCB during partition switching. During a context-switching, LTZVisor saves the running virtual machine state in the VMCB before restoring the context of the machine about to be executed, from the corresponding VMCB to the physical processor context. To ensure that the visibility provided to the secure world VM over the normal VM will not compromise or interfere with the normal world execution, the secure world application should have a small footprint and be aware of the virtualization. This attributes suits well with an RTOS and the higher execution privilege will help meeting the demanded time requirements. Additionally, RTOS typically possesses a small memory footprint. Since the secure VM is part of the system's TCB, a small memory footprint is also attractive as it follows LTZVisor's principle of minimal implementation. The non-secure VM runs in the supervisor mode of the non-secure world. This VM is ideal to host a general purpose guest OS which is useful to run human-machine interfaces or Internet-based applications and services. TrustZone guarantees that software running in the non-secure world is completely isolated from the secure world software. Any attempt from the non-secure guest to access any secure resource, triggers an exception to the hypervisor.

4.1.2 LTZVisor-AMP (Multicore Extension)

The multicore extension leverages LTZVisor toward multicore configuration. Pinto et. al. [20] have implemented and demonstrated LTZVisor in a multicore configuration. A GPOS is hosted side-by-side with an RTOS on Xilinx Zynq platform, in a dual-core configuration. The GPOS runs in the de-privileged non-secure core, while the RTOS runs in the privileged secure core. The implementation conducted in this thesis extends this approach, toward a quad-core deployment. The quad-core implementation conducted here will explore two approaches regarding the system's cores partitioning:

4.1.2.1 Dedicated Core Configuration

Similar to the dual-core configuration implemented in [20], one core is dedicated to run in the secure privileged world, while the remaining cores are assigned to the de-privileged non-secure world. Figure 4.2 depicts the proposed virtualization architecture based on three main software components: the hypervisor, the real-time guest OS, and the general-purpose guest OS.

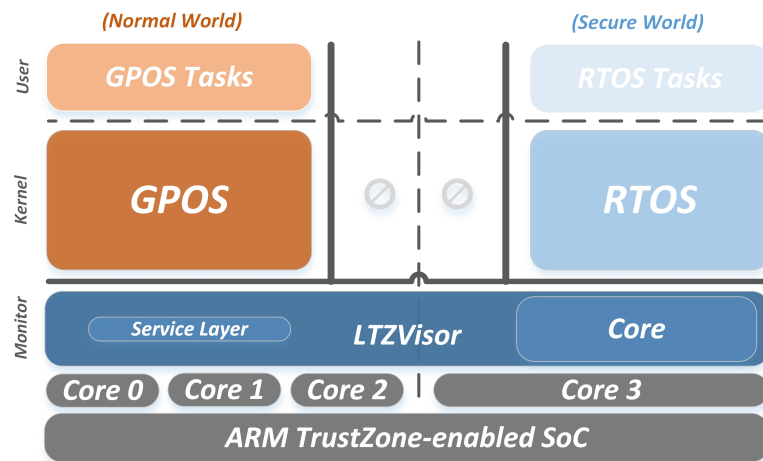


Figure 4.2: LTZVisor-AMP Dedicated Core Configuration General Architecture.

4.1.2.2 Shared Core Configuration

This configuration tries to take as much advantage of the system's processors. With the goal of creating an environment as close as possible to the native version, during the secure OS idle time the secure core is assigned to the non-secure world, following the same scheduling policy as the single-core configuration. This configuration architecture is depicted in figure 4.3. Equally to the other's configurations, the system is based on three main software components: the hypervisor, the real-time guest OS, and the general-purpose guest OS.

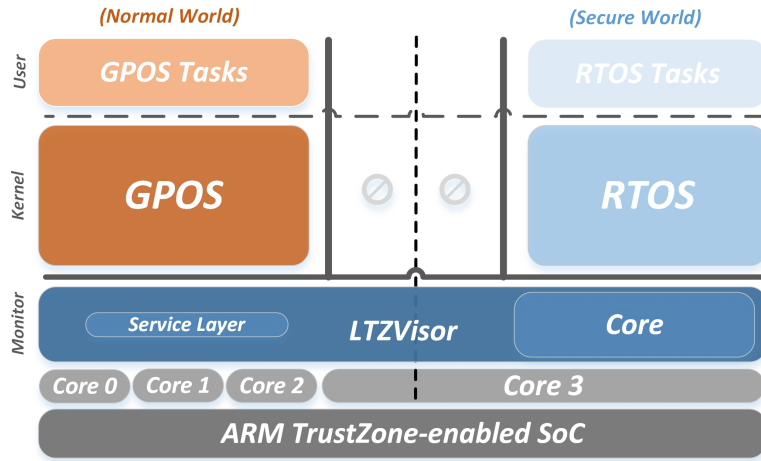


Figure 4.3: LTZVisor-AMP Shared Core Configuration General Architecture.

Following the AMP schema, LTZVisor performs the same tasks done in the single-core configuration: guests memory, interrupts and devices assignment.

4.1.3 Secure Guest

The OS chosen to run in LTZVisor's secure world is the market leading real-time OS FreeRTOS. Its small size and simplicity makes it perfect for embedded devices. FreeRTOS kernel has been ported to a variety of microcontrollers based on its services for multiple prioritized tasks, mutexes, semaphores and software timers. Many features and attributes provided by FreeRTOS makes it ideal to be adopted as LTZVisor's secure world OS. Below is presented a list of peculiarities that led the adoption of FreeRTOS as LTZVisor's secure guest OS.

1. Open Source:

Because the virtualization technique used here is the paravirtualization, accessing the secure guest OS source code is a requirement in order for this to be fit to run in the LTZVisor's secure world. The changes made in the OS are very small and will be detailed later in the implementation chapter. Therefore, the granted permission of free of charge, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of FreeRTOS software, makes FreeRTOS an attractive OS for LTZVisor's secure world.

2. Documentation:

Since its launch, FreeRTOS has grown to be one of the most popular real-time operating system as it can be built with approximately twenty different compilers, and run on more than thirty different processor architectures. This popularity has led to the development of an enormous amount of documentation that supports the variety of FreeRTOS based application. The educational nature of LTZVisor makes documentation an high asset.

3. **Small Memory Footprint:**

FreeRTOS is a scalable real-time kernel designed specifically for small embedded systems as it is constructed to be simple and easy to use. Its kernel binary image size is around 4K to 9K bytes while its APIs are designed to be simple and intuitive. Having a small footprint makes FreeRTOS an attractive choice for embedded application and this in the case for LTZVisor.

4.1.4 **Non-Secure Guest**

Linux was adopted as LTZVisor's non-secure GPOS. The complexity increase in today's embedded application has led Linux to become one of the main operating systems of choice in the embedded world. Other than its popularity below is presented some main reasons of its choice as LTZVisor's non-secure OS:

1. **Open-Source:** Although Linux under LTZVisor is not aware of the virtualization, some modifications are required in order to run the OS in non-secure world. The security extension implementation should not increase the software design cost of applications where this feature is not used. With this in mind, OSes for platforms where the security extension is implemented are designed to run as secure. This way, in order to run in LTZVisor's non-secure world, Linux must be able to run as non-secure. Thus, being open-source allows the adaptation of the OS to run as non-secure.
2. **Documentation:** Being open-source has led the creation of a wide range of Linux distribution tackling a variety of fields and application. This, comes altogether with the expansion of the Linux documentation availability. For i.MX 6Quad, ARM and NXP provide a ton of documentation regarding the distribution used in the board. A well documented OS is attractive to LTZVisor for reasons such as: (I) LTZVisor being an open-source approach to understand, evaluate, and encourage research towards TrustZone assisted virtualization dictates documentation as a key part of the project, (II) the imminent necessity to adapt the OS to run as non-secure, requires a deep understanding of the OS and its subsystems.
3. **Embedded Oriented Distribution:** Linux distribution targeting embedded devices is an acclaimed concept in the embedded world known as Embedded Linux. Embedded Linux is specifically customized for embedded systems, bearing features such as a small memory footprint, components and processing power. Many platform manufacturers maintain their own Linux distribution and provides documentation, assistance and support for Embedded Linux developers.

4.2 Development Platform

This work aims to port LTZVisor to the i.MX 6Quad Sabre Devices Board. i.MX, is a family of Freescale Semiconductor microcontrollers for multimedia applications based on ARM processors and focused on low-power consumption. The platform features an advanced implementation of the quad ARM Cortex-A9 core, operating at speeds up to 1.2 GHz. The family provides power-efficient processing capabilities covering a variety of user experience that ranges from automotive infotainment, human-machine interface, in-flight entertainment, industrial control systems, IoT gateways, portable medical and mobile devices. Figure 4.4 shows the functional modules in the i.MX 6Quad processor system.

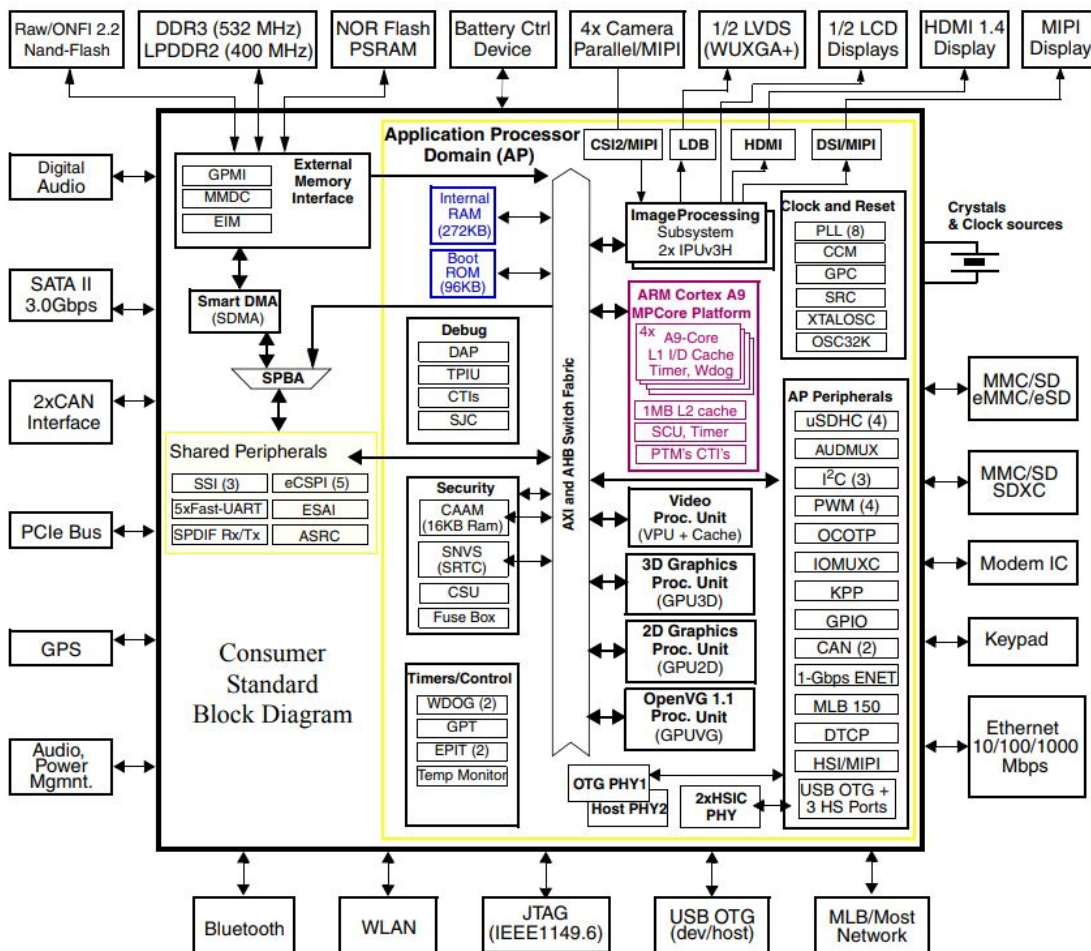


Figure 4.4: i.MX 6Dual/6Quad Consumer Grade System Block Diagram from [131].

The variety of hardware components integrated in the i.MX6 SoC, establishes a rich environment capable of matching and satisfy the diverse requirements of today's embedded system's applications. The hardware set integrated in the i.MX6 SoC builds a platform capable of supporting resources virtualization with rather small effort (assisted-by-hardware), while delivering high performance.

Due to the considerable amount of architectural components and for the sake of clarity, in this section it is only presented the platform resources relevant to the project. For a more detailed platform specification is recommend consulting the platform Application Processor Reference Manual provided by the platform manufacturer.

4.2.1 ARM Cortex A9 MPCore Quad-Core

In the heart of the i.MX 6Quad is the ARM Cortex-A9 MPCore, a multicore 32-bit processor core implementing the ARMv7-A architecture [114] with four cache-coherent cores. The MPCore implements the virtualization extension used to assist the LTZVisor. The ARM Cortex-A9 MPCore features relevant to LTZVisor implementations will be described below.

4.2.1.1 4 Cortex-A9 processors

Each processor in the ARM Cortex-A9 MPCore includes a Neon co-processor, a private timer and watchdog as well as 32 KB + 32 KB L1 data and instruction caches.

4.2.1.2 General Interrupt Controller (GIC)

The GIC controller in the Cortex-A9 MPCore implements 16 Software Generated Interrupts (SGI), 5 Private Peripheral Interrupt (PPI) and 128 Shared Peripheral Interrupt (SPI). The controller also includes 31 Lockable Shared Peripheral Interrupt (LSPI), that allows locking the interrupt against further changes. The security extension implemented in the MPCore interrupt controller, permits all implemented interrupts to be individually defined as secure or non-secure. Secure interrupt can be programmed to be handled either as IRQ or FIQ while non-secure interrupts are always handled as IRQ. LTZVisor handle secure interrupt as FIQ and the non-secure as IRQ.

4.2.1.3 Snoop Control Unit (SCU)

SCU holds the virtualization extension registers that allow access to secure processor's features such as global timer, L2 cache to the non-secure World.

4.2.2 Central Security Unit (CSU)

One of the hypervisor's main goal is guaranteeing a secure isolated execution environment for its guests. LTZVisor follows the same approach and to securely isolate the guests, the Central Security Unit (CSU) is used to impose the peripheral access policy control. CSU manages the system security policy for peripheral access on the SoC, allowing trusted code to set individual security access privileges on each of the peripherals, using one of the eight security access privilege levels. It sets access control policies between bus masters and bus slaves, allowing

peripherals to be separated into distinct security domains, protecting them against unauthorized access. The peripheral access policy is set using the Configure Slave Level (CSL) registers.

The CSL Register value defines the peripheral output signal `csu_sec_level`. The signal determines the masters to which the peripheral is accessible. The relationship between `csu_sec_level` signal and security operation mode is shown in the table 4.1.

CSU SEC LEVEL[2:0]	NS USR Mode	NS SVC Mode	Sec USR Mode	Sec SVC Mode
0x000	RW	RW	RW	RW
0x001	None	RW	RW	RW
0x010	RO	RO	RW	RW
0x011	None	RO	RW	RW
0x100	None	None	RW	RW
0x101	None	None	None	RW
0x110	None	None	RO	RO
0x111	None	None	None	None

Table 4.1: Permission access table.

4.2.3 Pin Multiplexing - IOMUX

i.MX 6Quad SoC possesses a ton of functionality but a limited number of pins (or pads). Even though a single pin can only perform one function at a time, they can be configured internally to perform different functions. This is called pin multiplexing. Most pins or pads have multiple signal options, which are selected by the input/output multiplexer (IOMUX) in order to facilitate the connection of on-chip peripheral signals to the external system. The IOMUX is also used to configure other pin characteristics, such as voltage level, drive strength, and hysteresis. The IOMUX module allows up to 8 internal signals to be connected to a single I/O pad on a device. If only one functional mode is required for a specific pad, there is no need for IOMUX and the signals can be connected directly from the module to the I/O. The main groups of IOMUXC registers are the general purpose registers used to select operating modes for general features in the SoC, the Software MUX control registers used to connect the pad to a given port in a module and the PAD settings registers used to control the pad configuration.

4.2.4 On-Chip Memory

The on-chip RAM (OCRAM) is implemented as a slave module on the 64-bit system AXI bus. Designed as a simple on-chip memory block, it has one bank of single port SRAM and supports one AXI port. i.MX 6Quad possesses a 256 Kbytes on-chip RAM, organized as 16K x 64 bits,

mapped from 0x00900000 to 0x0093FFFF. OCRM supports TrustZone and non-TrustZone accesses, allowing the memory region to be split in two areas, secure and non-secure area. When the OCRM TrustZone feature is disabled the entire OCRM space is available for all access (secure/non-secure). When enabled, the RAM space is split in two as explained previously while describing TrustZone Memory Adapter (TZMA). To configure the OCRM access policy is used the 10th IOMUX general purpose registers (IOMUXC_GPR10). GPR10 has two fields used to configure the OCRM access policy:

1. **OCRAM_TZ_EN** : OCRM TrustZone enable.
2. **OCRAM_TZ_ADDR** : The field specifies the start address of the secure memory region within the OCRM memory space. The start address affects the OCRM transactions only if OCRM_TZ_EN bit is set. The OCRM TZ ENDADDR is not configurable and is set to the end of OCRM memory space.

The register also provides bits to lock the configuration from further changing. When the TrustZone feature is enabled the OCRM can only be accessed (both read and write) according to the execution mode policy defined in the CSU_SEC_LEVEL field shown in the table 4.1.

4.2.5 Off-Chip Memory

i.MX 6Quad comes with a 3840 MBytes off-chip RAM, mapped from 0x10000000 to 0xFFFFFFFF. The memory can be mapped either to a single or dual-channel in the DDR Controller (MMDC). The single channel, maps the entire DDR to MMDC channel 0. The two channel configuration can be implemented in two ways: two separate 32 bit DDR memory devices with fixed mapping¹ or two separated 32 bits with an interleaved mapping². The i.MX 6Quad uses two TZASC instances, one on each DDR channel (MMDC0 and MMDC1). TZASC1 provides protection on memory accessible via MMDC0, while TZASC2 provides protection for memory accessible via MMDC1. Each TZASC allows the DDR to be partitioned in up to 16 regions with independent access control.

4.2.6 Enhanced Periodic Interrupt Timer (EPIT)

EPIT is a 32-bit set-and-forget timer, capable of providing precise interrupts at regular intervals with minimal processor intervention. It bears a 32-bit down counter with clock source selection, a 12-bit prescaler and a counter register that can be programmed on the fly.

¹ In the fixed mapping the addresses 0x10000000 to 0x7FFFFFFF (2048 MBytes) are mapped to MMDC1 and the addresses 0x80000000 to 0xFFFFFFFF (1792 MBytes) are mapped to the MMDC0

² In the 4KB interleaving mode the system bus maps each consecutive 4KB region to a 4KB region in the other MMDC port, such that "even" 4KB spaces are mapped to MMDC0, and "odd" 4KB regions mapped to MMDC1

4.2.6.1 Operating Modes

EPIT supports two modes of operation:

1. **Set-and-forget:** In this mode, the counter obtains its data from the load register. Whenever the counter reaches zero, load register is loaded into the counter.
2. **Free running:** In this mode, the counter rolls over from 0000_0000h to FFFF_FFFFh. After reaching zero, the counter continues counting down from FFFF_FFFFh.

4.2.6.2 Compare Events - Interrupt

When EPIT interrupt is enabled, its generated an interrupt anytime the counter value is equal to the compare register value. EPIT its used to implement the secure OS tick.

4.2.7 Universal Asynchronous Receiver/Transmitter(UART)

The i.MX 6Quad has 5 UART ports with features such as speed up to 4.0 Mbits/s, provides RS232 interface, supports 9-bit RS485 multidrop mode. One of the five supports 8-wire (UART 1) while the other four support 4-wire (Due to SoC IOMUX limitation). In LTZVisor UART is configured in the terminal mode (DTE), serving as the input/output of messages to the console.

5. Implementation

In this chapter, it is detailed the LTZVisor's implementation. Beginning with the implementation of the board support package needed by LTZVisor to configure the platform's hardware, then is presented the implementation of LTZVisor's modules, which starts with context switching mechanisms followed by the guest paravirtualization and then the multicore extension.

5.1 LTZVisor: Board Support Package (BSP)

In this section is depicted the implementation of the architecture-specific modules used by LTZVisor. It is shown the driver implemented to interface the peripherals along with the chosen configuration.

5.1.1 Pin Assignment (IOMUX)

The i.MX 6Quad possesses thousands of pins and even more signals options to be connected to pins. Configuring each pin manually would require a lot of effort and the possibility of human-made mistakes could compromise the entire system. To automate and speed the procedure, NXP provides tools to make the pins assignments. The i.MX Pin Mux tool [132] is an application that allows the developer to easily assign the internal signals to the external devices pins. The interaction with the application is through a graphical user interface (GUI) which makes the works easy and straightforward. The application identifies and assists in the resolution of conflicts when making the assignments in real-time.

The tool generate C code to configure the IOMUXC registers according to the users design. This alleviates developers from the tedious task of comparing schematics to the i.MX Reference Manual to establish connections to peripherals and also serves as supplementary documentation of a system. A complete and detailed specification of the i.MX Pin Mux tool is provided in [133].

In LTZVisor, the tool is used to generated the C code to configure the IOMUXC registers of the following peripherals: GPIO, EPIT 1 and 2, UART 1, 2, 3, 4 and 5, CCM, and SRC. The code generated is used by the LTZVisor when initializing each of the peripherals listed above.

5.1.2 UART Driver

This section shows the implementation of the UART driver, which is a low-level driver able to handle most common uses of a RS-232 serial interface. All UART ports are controlled through this driver and all functions can be called from any place in the code.

The UART driver consists of three main functions: The `ltz_uart_init` used to initialize the UART serial communication device, the `ltz_uart_getchar` used to read a character from the UART receiver buffer and the `ltz_uart_putchar` used to write a character to the UART transmitter buffer.

Initializing the UART driver

Before using the UART port, LTZVisor must initialize the UART by calling the `ltz_uart_init` function. The function receives the parameter `baudrate` that specifies how fast data is sent over the serial line. LTZVisor uses by default 115200bps. The algorithm 1 shows the procedure used to initialize the UART driver.

Algorithm 1: UART Driver Initialization algorithm.

Function `ltz_uart_init(Baudrate: uint32_t)`

```

    Enable the source clocks to the UART Port;
    Empty TxFIFO;
    Disable UART;
    Set OperatingMode  $\leftarrow$  DCE/DTE mode;
    Select the UART source clock;
    Set Parity  $\leftarrow$  No Parity;
    Set StopBit  $\leftarrow$  One Stop Bit;
    Set DataSize  $\leftarrow$  8 Bits;
    Set Parity  $\leftarrow$  No Parity;
    Enable UART;
    Perform software reset;
    Clear status flags;
    Reset the transmit;
    Set Baudrate;

```

end

Reading the UART driver

To read the Data from the UART port it is used the `ltz_uart_getchar` function. The algorithm is depicted in algorithm 2.

Writing the UART driver

To output the data, the driver implements the `ltz_uart_putchar`, a function that when called writes a byte into the UART transmitter buffer. The function's behavior is presented in algorithm 3. This function is used to implement the `printf` function based on the C `printf` function.

Algorithm 2: UART Driver GetChar algorithm.

```

Function ltz_uart_getchar(void)
    if Received Data then
        | Return Received Char;
    end
    else
        | return EMPTY_CHAR;    /* 0xFF is the value of an Empty Char */
        | */
    end
end

```

Algorithm 3: UART Driver PutChar algorithm.

```

Function ltz_uart_putchar(data: uint8_t)
    while FIFO Buffer is Full do
        | wait;
    end
    Write TxFIFO  $\leftarrow$  data;
end

```

5.1.3 EPIT Driver

This section details the implementation of the EPIT driver. This low-level driver configure the EPIT timer to periodically interrupt the CPU and delivers the system tick for the real-time OS scheduler running in the secure world. There are two instances of EPIT timer and the driver allows the usage of both. By default LTZVisor uses the first instance (EPIT1).

Configuring the EPIT timer

The EPIT driver implementation separates the timer configuration from the counter enabling. In doing so, the driver can be configured once and enabled or disabled when necessary. The EPIT configuration follows the steps shown in algorithm 4.

The EPIT compare register is kept to 0 and reload register is used to setup the interrupt. This way an interrupt is always raised when the counter reaches zero.

EPIT timer Enable and Disable function

²The variable Instance is used to define the registers base address used in the timer configuration, the EPIT1 base address is at 020D0000h while the EPIT2 is at 20D4000h.

²To minimize going too deep in the EPIT driver implementation, the calculations necessary to select the intended EPIT timer clock is not presented here, but the end result is that the clock is selected such that every counts its precisely 1 microseconds.

Algorithm 4: EPIT Driver configuration algorithm.

Function `ltz_epit_init`(Instance¹, Load_Value: uint32_t, uint32_t)
 Enable EPIT source clock;
 Reset the EPIT module;
 Set EPIT source clock ²;
 Set EPIT clock prescaler ;
 Set *ReloadMode* $\leftarrow$ Set-And-Forget mode;
 Set *PowerMode* $\leftarrow$ Enabled in Wait mode and Stop mode;
 Write *LoadRegister* $\leftarrow$ Load_Value;
end

As said before enabling and disabling the timer are implemented separately from the timer configuration. Below is presented the algorithms to implement both functions, the EPIT enable function in algorithm 5 and EPIT disable function in 6.

Algorithm 5: EPIT enable algorithm.

Function `ltz_epit_enable`(Instance: uint32_t)
*StatusRegister*³ $\leftarrow$ 1;
 Enable EPIT interrupt;
 Enable the EPIT counter;
end

Algorithm 6: EPIT disable algorithm.

Function `ltz_epit_disable`(Instance: uint32_t)
*ReloadCounter*⁴ $\leftarrow$ 0xFFFFFFFF;
 Disable EPIT interrupt;
 Disable the EPIT counter;
StatusRegister $\leftarrow$ 1;
end

To change the reload register it is provided the `ltz_epit_reload` function. Since the configuration used for LTZVisor is using the Set-And-Forget mode and an interrupt is raised when the counter reaches zero, it is also implemented the function `ltz_epit_get_compare_event` that is called to clear the status register when the interrupt is handled.

³ EPIT Status Register possesses the compare event flag and writing one to the register clear the bit in case a compare event occurred previously

⁴The i.MX 6Quad Reference Manual suggests setting the counter to the maximum value so that it ensures that the counter could not reach 0 when it is disabled.

5.1.4 GIC Driver

This section presents the implementation of the GIC Driver. The general interrupt controller (GIC) supports up to 128 shared peripheral interrupt (SPI) sources and 16 software generated interrupt (SGI) sources. The GIC is split into a main interrupt distributor block common to all the cores and individual core interface blocks, one for each Cortex-A9 core present in the system.

GIC Core Interface (GIC CI)

The GIC Core Interface main functions deliver interrupts to the CPU while acknowledging the distributor when an interrupt has been handled. When enabled, the interface takes the highest priority pending interrupt available from the distributor and determines whether the interrupt source has sufficient priority to interrupt the core to which it is connected. The GIC CI initialization follows algorithm 7.

Algorithm 7: GIC CI initialization algorithm.

```

Function gic_int_cpu(void)
    Set PriorityMask  $\leftarrow$  0xFF5;
    Enable preemption;
    Enable secure interrupt signaling;
    Enable secure interrupt to be routed as FIQ;
end

```

When handling an interrupt, GIC CI must send an acknowledge to the GIC interrupt distributor (GIC ID). This is accomplished by calling the driver's function `gic_read_irq_ack` that reads the interface Acknowledge register that automatically sends the acknowledge to the distributor. The function returns the ID of the interrupt. Similarly, when the interrupt service is completed, the core interface must send an end-of-interrupt signal to the distributor, this is accomplished by calling the function `gic_write_end_of_irq`, that writes the CPU Interface End of interrupt register.

GIC Interrupt Distributor (GIC ID)

The GIC ID main goal is to distribute every interrupt in the system to the correspondent GIC core interfaces. It performs the interrupt prioritization and security. The distributor receives interrupts and provides the highest priority interrupt to the corresponding CPU interface. An interrupt with a lower priority is forwarded to the appropriate CPU interface when it becomes the highest priority pending interrupt. In LTZVisor the secure interrupts are handled as FIQ and the non-secure as IRQ, meaning that the secure interrupt are always handled before the non-secure. The GIC driver implements the `gic_set_irq_priority` that receives the ID and the interrupt priority as parameters

⁵A CPU Interface contains a programmable interrupt priority mask and it only accepts pending interrupts if the priority of the interrupt is higher than the programmed interrupt priority mask. In this implementation the value is set to lowest priority level (0xFF).

and sets the interrupt priority for the intended interrupt source. The i.MX 6Quad GIC controller implements 256 priority levels and for each interrupt ID its dedicated one byte of the Interrupt Priority Registers in a systematized way (first byte to interrupt ID 1, second byte to interrupt ID 2, etc.). The main functionality of `gic_set_irq_priority` is writing the priority level into the received interrupt ID field.

The driver also implements a function to set the interrupt security level of a particular interrupt, the `gic_set_irq_security`, which receives the interrupt ID and the security level (secure or non-secure) and according to the security level the function it writes the Interrupt Group Registers field to the selected interrupt. The GIC controller possesses 8 32 bits Interrupt Group Register and each bit is dedicated to one interrupt ID, being the first bit of Interrupt Group Register[0] dedicated to the interrupt ID 1 and the first bit of Interrupt Group Register[1] dedicated to the ID 33, etc.. Each bit is set to 0 for secure interrupts and 1 for non-secure.

Each interrupt source can be targeted to a single or multiple cores. Hardware ensures that if an interrupt is targeted to several of the available cores, only one of the cores handles it. This feature is implemented in `gic_set_cpu_target`, which receives the interrupt ID and the CPU ID where its intended to target the interrupt. The function writes the Interrupt Processor Targets Registers that store the list CPUs for which the interrupt can be Pending. The Interrupt Processor Targets Registers are byte accessible and each byte corresponds to the core targets of an interrupt source. The bits in each byte is dedicated to each core (eg., bit 0 core0, bit 1 - core1, etc.). This set allows the GIC distributor to distribute the load between cores effectively. The GIC Controller initialization is achieved following the algorithm 8.

Algorithm 8: GIC Controller initialization algorithm.

Function `gic_init(void)`

```

    Disable distributor;
    Clear all pending interrupt;
    Call gic_init_cpu function;
    Set all interrupt as non-secure6;
    Enable distributor;
```

end

In an effort to centralize the driver implementation and reduce the number of functions called to configure and initialize the GIC driver, functions are implemented to combine the different functions presented above to achieve the intended configuration.

⁶When configuring the GIC Controller, all interrupt is set as non-secure. Later when initializing the secure modules the interrupt security of secure interrupts is changed to secure. This way the interrupts not used by LTZVisor or the secure guest will be available to the non-secure guest.

5.1.5 Security Features

One of the main security support is given by the Central Security Unit (CSU). The CSU enables secure software to set bus privilege security policy within the platform. It sets access control policies between bus masters and bus slaves, allowing peripherals to be separated into distinct security domains (Secure and non-secure) domains. i.MX 6Quad possesses 40 Config Security Level Register(CSL), each consisting of two fields used to determine the read and write access permissions for a slave peripheral. The first slave's field are from the first 9 bits (CSL[8:0]) and the second by the first 9 bits of the register second word (24:16). The first 8 bits of the slave field determines the slave access control and the 9th bit locks the configuration from further changes. This registers are only accessible in privileged mode. LTZVisor sets the secure peripherals access mode to secure and the remaining as non-secure, following the algorithm 9. The list of secure peripherals depends on LTZVisor and the secure guest implementation.

Algorithm 9: CSU configuration algorithms.

Function ltzvisor_csu_config(void)

```

    Set all Peripheral's Access as non-secure;
    Change all secure peripherals access to secure;
    Lock the Secure peripherals configuration bits;

```

end

TrustZone Address Space Controller (TZASC)

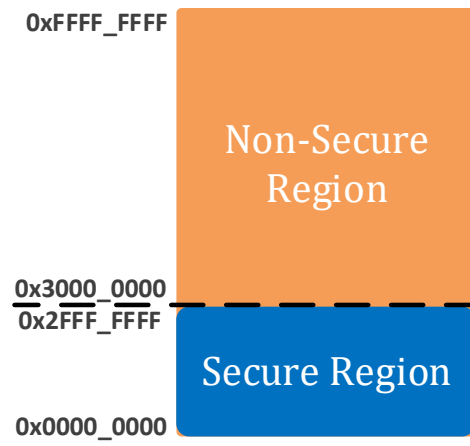
The i.MX 6Quad utilizes bus muxing logic to route DDR memory traffic through or bypass the TZASC modules. TZASC modules are bypassed by default, in order to route the DDR traffic through the TZASC module, with the module clock gated on. TZASC registers also implements feature that allows the TZASC configuration to be locked from changes. TZASC allows the memory to be partitioned in multiple regions, each region is programmable for its size, base address, enable and security. TZASC always provides a minimum of two regions, as contiguous areas of the memory address space. Except the region 0, TZASC allows the other regions to be enabled and disabled and prioritized. The priority of a region is fixed and determined by the region number. The priority of a region increases with the region number meaning that when a transaction is received it is first checked if its address matches the higher region. Since LTZVisor only needs two address domains, the TZASC is programmed to create two address space one for each world. TZASC also implements a feature that allows the security inversion of a region, meaning that the non-secure region can be configured to become inaccessible to the secure world application. The configuration used in the LTZVisor is such that one region is set as secure (region 1) and the other as non-secure (region 0). The priority inversion is not used in the non-secure region, meaning that the secure world have full access of the non-secure memory space. The steps used to configure TZASC to implement the memory protection required by LTZVisor follows the algorithm 10.

Algorithm 10: DDR TZASC configuration algorithm.

```

Function ltzvisor_ddr_tzasc_config(void)
    Set TZASC NumberOfRegions  $\leftarrow$  2;
    Set Region BaseAddress  $\leftarrow$  07;
    Select SecureRegionSize  $\leftarrow$  512MB ;
    Set region 0 AccessPermission  $\leftarrow$  non-Secure Access;
    Set region 1 AccessPermission  $\leftarrow$  Secure Access;
    Lock regions from further changes;
    Enable region 1;
end

```

**Figure 5.1:** LTZVisor's worlds memory partitioning.

i. MX 6Quad possesses 3840 MBytes off-chip RAM, mapped from 0x1000_0000 to 0xFFFF_FFFF. In the implementation 512MB is dedicated to the secure world application mapped from 0x1000_0000 to 0x2FFF_FFFF, leaving the non-secure world with the remaining 3328 MBytes mapped from 0x3000_0000 to 0xFFFF_FFFF, as shown in the figure 5.1. This configuration can be changed at anytime in case it is required additional memory space to the non-secure OS.

Snoop Control Unit (SCU)

The SCU connects the cores to the memory systems through AXI interface, it is responsible to maintain the cache coherency between the system processors. It manages the access to the Accelerator Coherency Port (ACP) and L2 cache. SCU with the virtualization extension also controls the non-secure access to the global and private timers. These MPCore components are by default secure. Allowing non-secure access to the CPU features mentioned above requires changing the configuration of the SCU Access Control Register and the SCU Non-secure Access Control Register. Both registers are changed to allow non-secure access to the MPCore components. This step

⁸Setting the region 1 base address to 0 means that the secure region will start at the begin of the DDR memory address, since the region 0 covers the entire memory space, the non secure address space start at the end of the secure address space.

is demanded when running Linux in the non-secure world due to its dependence on the global timer.

5.2 LTZVisor

This section details the LTZVisor's implementation by showing the implementation of the different hypervisor's modules and functions such as: memory isolation, MMU and cache management, SoC partitioning and interrupt management.

A well known concept used when implementing a system where security is a requirement, is the implementation of the Chain of Trust, as a measure to guarantee that the system is not compromised during its initialization. With this in mind, TrustZone-enabled processor always starts in the secure mode. Having the processor in secure mode after the power-on allows the implementation of a secure foundation which the LTZVisor relies on during its initialization. The first step towards the LTZVisor initialization is the configuration of the CPU registers. The configuration prior LTZVisor starts by disabling the TRE bit of System Control Register(SCR) that control the TEX remap functionality in the MMU. The SCR vector bit is also disabled to allow the exception vector address to be re-mapped. The second register configured is the Non-secure Access Control Register (NSACR), which is responsible to set the non-secure access permission for the co-processors. NSACR is configured so that the non-secure world is permitted to access both co-processors 10 and 11, allocate lockable TLB and change SMP bit of the Auxiliary Control Register (ACR). Enabling the non-secure access SMP bit is demanded in the multicore configuration where Linux is runs in a SMP. Another crucial task performed during the processor configuration is the stack setting for the different CPU modes. Stack Pointer (SP) for each mode is points to memory addresses previously reserved for each mode. Each stack is configured with 1024 Bytes. At this point all the configurations made was through the secure supervisor mode (secure EL2). For the subsequential configurations the CPU is changed to the monitor mode (EL3). Exception handlers necessary to handle the monitor exceptions are installed in the Monitor Vector Base Address Register(MVBAR). LTZVisor's MVBAR implements two exceptions handlers: (I) SMC handler, used to handle the secure monitor calls from both secure and non-secure world and (II) FIQ handler, used to serve the secure OS tick interrupt when the processor is in non-secure state. Afterwards, it is configured the virtual memory, disabling the the MMU and the L1 cache (data and instruction) and invalidating the instruction cache, data cache, unified TLB inner shareable and the branch predictor array. Subsequently the execution branches to the LTZVisor main.

The main tasks performed by the LTZVisor is initialization and configuration of hardware peripherals as well as the non-secure guest creation. The hardware peripherals configuration follows the algorithm 11.

Algorithm 11: LTZVisor Hardware initialization algorithm.

```

Function ltzvisor_hw_init(void)
    Configure Peripherals IOMUX Registers;
    Initialize the UART Port;
    Initialize the GIC Controller ;
    Configure the Secure Guest Timer;
    Configure Peripherals Access through CSU;
    Set memory regions and configure its access policy;
    Configure CPU Components Access through SCU;
end

```

The non-secure guest is included in LTZVisor as a blackbox and the only parameters of the non-secure world guest available to LTZVisor are: Load Address, Guest Start Address and the Guest End Address. Because, not all the CPU registers are banked in both worlds it is used the VMCB to store the context of the two worlds when the respective VM is not executing. The VMCB of the non-secure world is composed by 27 registers: 13 General Purpose Registers (R0-R12), the Stack Pointer (SP), the Linker Register (LR) and Saved Program Status Register (SPSR) for the following modes: Supervisor, System, Abort and Undef. The “high” General Purpose Registers (R8-R12), as well as the SP, LR and SPSR of the FIQ and IRQ modes are not included, as they are mutually exclusive for each world. Among the co-processor registers, almost all of them are banked: only the SCTLR and the ACTLR need to be preserved. For optimization purposes, the VMCB of the secure world is composed of only 24 registers: 13 General Purpose Registers (R0-R12), the Stack Pointer (SP), the Linker Register (LR) and SPSR for the System mode and 5 FIQ mode banked General Purpose Registers (R8-r12) SP, LR and SPSR. The first step when creating the non-secure guest is initializing the non-secure core registers (R0-R12, SPSR and LR and R13 from the different modes) in the VMCB. The non-secure guest Supervisor mode LR¹⁰ in the VMCB is loaded with the Load Address value (non-secure guest entry point). The co-processors registers are also initialized with zero with the exception of the SCTLR. The hypervisor fill some registers of the non-secure VMCB with a specific initialization value during the VM creation. SCTLR is configured to set Round-robin as the replacement strategy for caches, BTAC, and micro TLBs as well as disabling hardware management flag. If necessary by the non-secure guest, L2 and L1 prefetch bits in the ACTLR are also configured at the guest creation. After initializing the non-secure guest VMCB registers, the non-secure guest is reallocated to the non-secure memory space and at this point on, the non-secure guest is ready to run and the VMCB is fully configured. The VMCB banked registers are written to their non-secure copies. With the non-secure co-processor copies written, the address of VMCB data struct is stored in the Software Thread ID register

¹⁰LTZVisor is implemented such that the non-secure entry point is always the LR value, after the context switching the first instruction executed in the non-secure world OS is always the LR, for this reason the Load Address is set to LR

TPIDRPRW which is a register used for OS management purposes. It provides a location where privileged software can store thread identifying information that is not visible to software execution at unprivileged level. With the non-secure VMCB saved, LTZVisor passes execution to the secure guest running in the supervisor mode.

The implementation of the non-secure guest creation detailed above is listed in 5.1.

```

1 uint32_t ltzvisor_nsguest_create(struct nsguest_conf_entry *g)
2 {
3     /** Init Guest attributes */
4     NS_Guest.booting = 1;
5     NS_Guest.id = g->gce_id;
6     strcpy(NS_Guest.name, g->gce_name);
7     /** Guest Core Registers */
8     /* Clean core registers */
9     memset(&NS_Guest.core.vcpu_regs_core,0,sizeof(struct core_regs));
10    /* lr = start_addr & spsr=supervisor */
11    NS_Guest.core.vcpu_regs_core.lr_mon      = g->gce_bin_load;
12    NS_Guest.core.vcpu_regs_core.spsr_mon    = 0x13;
13    /** Guest CP15 Registers */
14    /* Clean CP15 registers and set SCTLR register in VMCB*/
15    memset(&NS_Guest.core.vcpu_regs_cp15,0,sizeof(struct cp15_regs));
16    NS_Guest.core.vcpu_regs_cp15.c1_SCTLR = 0x00c50078;
17    /** Load Guest bin to Non-Secure Memory */
18    memcpy((uint32_t*) g->gce_bin_load,(uint32_t *)g->gce_bin_start,(
        (uint32_t)g->gce_bin_end - (uint32_t) g->gce_bin_start));
19    if(g->gce_trd_init) {
20        memcpy((uint32_t *)g->gce_trd_load,(uint32_t *)g->
            gce_trd_start,(g->gce_trd_end - g->gce_trd_start));
21    }
22    /** Get NS_Guest ready to run */
23    cp15_restore(&NS_Guest.core.vcpu_regs_cp15);
24    set_guest_context( (uint32_t)&NS_Guest);
25    printk("      * NS_Guest ready to run - OK  \n\t");
26    return TRUE;
27 }

```

Listing 5.1: Non-secure guest creationg.

5.2.1 Context-Switching

Real-time tasks running in the RTOS are under control of the RTOS scheduler. Virtualizing the RTOS adds another scheduling domain to the system, creating a well-known hierarchical scheduling problem. LTZVisor overcomes this problem by implementing an asymmetric scheduler. This scheduling policy guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and the secure guest OS can preempt the execution of the non-secure one. In this scheduling policy the guests are scheduled not by the hypervisor, but the secure OS. Although this can seem contradictory, the concept of ring protection is never jeopardized, as the LTZVisor continues executing in a most privileged mode. After configuring the processor, peripherals and the non-secure guest creation, LTZVisor's only task is performing the world switching and assist the non-secure OS in the secure access through APIs. The world-switching in the secure side is done via an SMC call by executing the arm instruction SMC#0. When executed, an exception is raised to the monitor that performs the world-switching. Arguments can be passed to SMC handler using registers, which allows the implementation of SMC calls for different purpose and responsibility, each with a specific ID. Except the secure context-switching, all the SMC calls are implemented as an API that brings limited privileged functionalities to the non-secure world. When the non-secure world OS needs to access a secure register this can be accomplished by creating an API that uses the SMC call to take the processor to the monitor mode then accessing the register that can be returned to the non-secure using the general purpose registers. The SMC handler differentiates between the different calls implemented by looking at the ID passed in R0. When the SMC is called from the secure world it uses the ID 0xFFFFFFFF1. During a context-switching first is saved the secure world. Unlike the non-secure world where the VMCB is saved in the Thread ID Register, secure world VMCB is saved in the monitor stack. With the secure context saved in the monitor stack, it is then restored the non-secure context from the Thread ID Register. After restoring the non-secure context and having both data and instructions synchronized, the monitor exception handler jumps to the non-secure world guest execution. The process of switching to the non-secure world is done following the algorithm 12.

For optimization purpose, the partitions context-switch were implemented using ARM assembly. The implementation of the algorithm 12 is presented in the listing 5.2.

```

1  @ ----- Save Secure World -----
2  @ Save MON/SYS state
3  stmfd  sp,{r0-r14}^    @ Push SYS regs to monitor stack
4  sub    sp, sp, #(15*4) @ Point stack to the top
5  mrs    r0,spsr         @ Read the SPSR
6  stmfd  sp, {r0,lr}     @ Push SPSR_mon & lr_mon
7  sub    sp, sp, #(2*4)  @ Update stack pointer
8  mov    r0, sp          @ Save pointer to monitor stack
9  @ ----- Save FIQ state -----

```

Algorithm 12: Non-secure context-switching algorithm.

```

Read VMCB from Thread ID Register;
Restore General Purpose Registers (r0-r12);
Restore Supervisor Mode;
Restore System Mode;
Restore Abort Mode;
Restore undefined Mode;
Restore IRQ Mode;
Set SRC  $NS - Bit \leftarrow 1$ ;
Route FIQ to Monitor;
Perform Instruction synchronization barrier;
Perform Data synchronization barrier;
Jump to the Non-Secure Guest;

```

```

10  cps      #Mode_FIQ
11  stmfd    r0,{r8-r13}      @ Save R8-R14
12  sub      r0, r0, #(6*4)
13  mrs      r1,spsr          @ Read the SPSR_FIQ
14  stmfd    r0, {r1,lr}      @ Push SPSR_FIQ and lr_FIQ
15  sub      r0, r0, #(2*4)    @ Update stack pointer
16  cps      #Mode_MON
17  mov      sp, r0
18  @----- Restore NS state -----
19  GET_NS_WORLD_CONTEXT
20  WORLD_CONTEXT_RESTORE
21  SCR_NS_SET
22  isb
23  dsb
24  @ ---- Return from exception (MON -> NS-SVC) ----
25  movs     pc, lr          @ Leave monitor mode

```

Listing 5.2: Non-secure context-switching.

In the listing 5.2, it is used 3 macros: (I) the GET_NS_WORLD_CONTEXT to read the non-secure VMCB, (II) the WORLD_CONTEXT_RESTORE to restore the non-secure world context and (III) SCR_NS_SET to change the NS_BIT required in order to change CPU to non-secure when the monitor mode is exited. Their implementation are presented in listings 5.3, 5.4 and 5.5.

```

1  .macro GET_NS_WORLD_CONTEXT
2  mrc p15, 0, r0, c13, c0, 4 @ Read PL1 only thread ID register
3  add r0, r0, #0              @ Add offset
4  .endm

```

Listing 5.3: GET_NS_WORLD_CONTEXT macro implementation.

```

1 .macro WORLD_CONTEXT_RESTORE
2 @ ----- Restore registers (r0-r12, lr) -----
3 mov      r4, r0
4 ldmia r4!, {r0 - r3}
5 push  {r0, r1}
6 mov   r0, r4
7 ldmia r0!, {r4 - r12}
8 @ ----- Restore Modes Banked Registers -----
9 ldmia  r0!, {r1, lr}
10 msr    spsr_cxsf, r1
11 cps    #Mode_SVC @ Restore supervisor mode
12 ldmia r0!, {r1, r13, lr}
13 msr    spsr_cxsf, r1
14 cps    #Mode_SYS @ Restore system mode
15 ldmia r0!, {r1,r13, lr}
16 msr    spsr_cxsf, r1
17 cps    #Mode_ABT @ Restore abort mode
18 ldmia r0!, {r1, r13, lr}
19 msr    spsr_cxsf, r1
20 cps    #Mode_UND @ Restore undefined mode
21 ldmia r0!, {r1, r13, lr}
22 msr    spsr_cxsf, r1
23 cps    #Mode_IRQ @ Restore IRQ mode
24 ldmia r0!, {r1, r13, lr}
25 msr    spsr_cxsf, r1
26 cps    #Mode_MON @ Get back to monitor mode
27 pop    {r0, r1}
28 .endm

```

Listing 5.4: WORLD_CONTEXT_RESTORE macro implementation.

```

1 .macro SCR_NS_SET
2 push  {r0}
3 mrc    p15, 0, r0, c1, c1, 0 @ Read SCR
4 orr    r0, r0, #SCR_NS_BIT @ Enable NS bit
5 orr    r0, r0, #SCR_FIQ_BIT @ Enable routing FIQs to Monitor
6 mcr    p15, 0, r0, c1, c1, 0 @ Write SCR
7 pop    {r0}
8 .endm

```

Listing 5.5: SCR_NS_SET macro implementation.

The non-secure guest execution is performed until the CPU gets interrupted by the timer tick interruption. In the event of a timer tick interrupt if the active guest is non-secure, it is handled by

the monitor FIQ handler that performs the non-secure context-switching. The switching process in the non-secure side reverse the operations done in the SMC Handler and the specific steps to perform the non-secure context switching is shown in the algorithm 13.

Algorithm 13: Secure context-switching algorithm.

```

Disable Routing FIQ to Monitor;
Set SRC NS – Bit  $\leftarrow 1$ ;
Handle The FIQ Interrupt;
Read VMCB from Thread ID Register;
Save General Purpose Registers (r0-r12);
Save IRQ Mode;
Save Undefined Mode;
Save Abort Mode;
Save System Mode;
Save Supervisor Mode;

```

In the listing 5.6 is presents LTZVisor’s FIQ monitor handler. It performs the switching from the non-secure world to the secure world. The listing includes all the activities performed by the handler, such as (I) acknowledging the tick interrupt, (II) save the non-secure world context and (III) restore the secure world context.

```

1 global NS_MON_FIQ_HDLR
2 NS_MON_FIQ_HDLR:
3 sub    lr, lr , #4
4 push   {r4,lr}
5 push   {r0-r3}
6 @ ----- START Handling Interrupt -----
7 stmfd  sp!, {lr}
8 stmfd  sp!, {r0-r12}
9 beq    FreeRTOS_TICK
10 ldmfd  sp!, {r0-r12}
11 ldmfd  sp!, {lr}
12 @ ----- Change CPU to secure Interrupt -----
13 mrc    p15, 0, r0, c1, c1, 0
14 bic    r0, r0, #NS_BIT      @ Clear NS bit
15 bic    r0, r0, #SCR_FIQ_BIT @ Disable routing FIQ to Monitor
16 mcr    p15, 0, r0, c1, c1, 0
17 @ ----- Save Non-Secure World Context -----
18 GET_NS_WORLD_CONTEXT
19 WORLD_CONTEXT_SAVE
20 @ ----- Restore Secure World Context -----
21 pop     {r1-r2}  @ Pop SPSR_FIQ and LR_FIQ
22 mov     r0, sp   @ Update stack pointer

```

```

23 @ ----- Restore FIQ state -----
24 cps      #Mode_FIQ
25 msr      spsr,r1
26 mov      lr, r2
27 ldmfd    r0, {r8-r13}      @ Restore FIQ regs (R8-R14)
28 add      r0, r0, #(6*4)    @ Update stack pointer
29 @ ----- Restore MON state -----
30 @ ----- And jump to the Secure World -----
31 cps      #Mode_MON
32 mov      sp, r0
33 pop      {r0-r1}
34 msr      spsr,r0
35 mov      lr, r1
36 ldmfd    sp, {r0-r14}~
37 add      sp, sp, #(15*4)
38 @ Return from exception (MON -> S_SVC)
39 movs     pc,lr

```

Listing 5.6: Monitor FIQ handler implementation.

The implementation of the `WORLD_CONTEXT_SAVE` macro, used in the monitor FIQ handler to save the context of the non-secure world is shown in the listing 5.7.

```

1 .macro WORLD_CONTEXT_SAVE
2 @ ----- Save registers (r0-r12, lr) -----
3 mov      r4, r0
4 pop      {r0-r3}
5 stmia    r4, {r0-r3}
6 mov      r3, r4
7 add      r4, r4, #16
8 mov      r0, r4
9 pop      {r4,lr}
10 stmia    r0!, {r4-r12}
11 @ ----- Save Modes Banked Registers -----
12 mrs      r1, spsr
13 stmia    r0!, {r1, lr}
14 cps      #Mode_SVC @ Save supervisor mode
15 mrs      r1, spsr
16 stmia    r0!, {r1, r13, lr}
17 cps      #Mode_SYS @ Save system mode
18 mrs      r1, spsr
19 stmia    r0!, {r1,r13, lr}
20 cps      #Mode_ABT @ Save abort mode
21 mrs      r1, spsr
22 stmia    r0!, {r1, r13, lr}

```

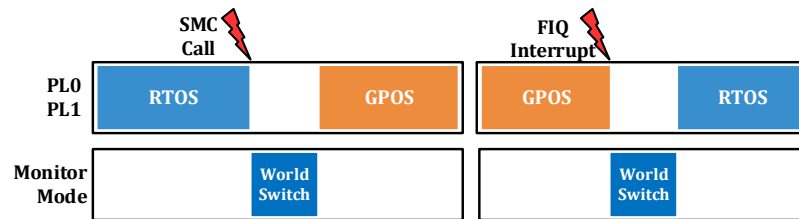


Figure 5.2: LTZVisor context switching cases.

```

23 cps    #Mode_UND @   Save undefined mode
24 mrs    r1, spsr
25 stmia  r0!, {r1, r13, lr}
26 cps    #Mode_IRQ @   Save IRQ Mode
27 mrs    r1, spsr
28 stmia  r0!, {r1, r13, lr}
29 cps    #Mode_MON @   Get back to monitor mode
30 .endm

```

Listing 5.7: WORLD_CONTEXT_SAVE macro implementation.

After saving the non-secure guest context the secure guest context is restored from the monitor Stack. First is restored the FIQ banked General Purpose Register, SPSR and LR, then the same process is repeated to the System Mode General Purpose Registers, SPSR and LR. LR is used as the return address back to the secure world.

To finalize the section on LTZVisor context-switching, the figure 5.2 presents a blueprint of the LTZVisor context-switching use cases. The system schedule is completely controlled by the secure world OS that configures a periodical FIQ interrupt that forces the non-secure world context-switching. When in the idle state, it grants execution to the non-secure OS by calling the secure monitor that performs the world context-switching.

5.2.2 Secure Guest

Using FreeRTOS kernel allows the secure world software to be completely event-driven, with the secure guest OS executed only when necessary. No processing time is wasted by polling for events that have not occurred. When there is no task to be executed the CPU runs the idle task, created automatically when the kernel scheduler is started. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

The paravirtualization technique used in the LTZVisor secure guest aims to create an asymmetric idle scheduler, where the non-secure guest OS is only scheduled during the idle periods of the secure guest OS. This is implemented by changing FreeRTOS so that during its idle time

the secure core is granted to the non-secure world OS. Using the Idle Task Hook API¹¹ it is implemented a function that makes a monitor call (SMC) with the scheduling ID. Since the Idle Task Hook function is only called during the idle task cycle, the non-secure guest only executes during FreeRTOS idle time. Since the native version of FreeRTOS uses IRQ to handle the interrupts, this was changed to FIQ to support the interrupt policy implemented in LTZVisor in which secure interrupts are handled as FIQ and non-secure as IRQ.

5.2.3 Non-Secure Guest

In order to run in LTZVisor's non-secure world the virtualized Linux was slightly changed. This is required in order to get rid of the secure access needed by Linux. As explained previously in the section 4.1.4, the native Linux was built to run in the secure world. Similar to the de-privileging phenomenon often found in software-based OS virtualization techniques, shipping the Linux OS to the non-secure world, undergo similar behavioral consequences as those in software based de-privileging. That is, the inability to execute privileged instructions, in this case access privileged resources. Since accessing these resources are vital to the OS, the solution used in LTZVisor is, identify all the secure access made by the Linux Kernel and replace it with hypercalls. To each implemented hypercall it is associated a procedure in the hypervisor level that implement the native operation. Upon its execution the hypercalls are trapped to the hypervisor (monitor mode) that performs the procedure attached to the hypercall requested by the non-secure guest. Hypercall in LTZVisor are implemented solely on SMC calls. Due to the multiple category of SMC calls, to each system call is attached an ID used by the monitor to distinguish the call source. The hypercalls implemented in LTZVisor are presented in the table 5.1. The table also includes the ID associated to each hypercall.

ID	Description
0xFFFF_FFE2	L2 Cache Enable
0xFFFF_FFE1	L2 Cache Disable
0xFFFF_FFE0	LTZVisor Read hypercall
0xFFFF_FFDF	LTZVisor Write hypercall

Table 5.1: LTZVisor hypercalls Table.

The implementation of these hypercalls takes advantage of the common general purpose registers (usually r0 to r3) to assist in data exchange between Linux Kernel and the Hypervisor:

1. L2 Cache Enable: In this hypercall it is only used the general purpose R0 that holds the hypercall's ID. When LTZVisor receives a call and R0 holds the value 0xFFFF_FFE2, it is

¹¹This FreeRTOS API is used to implement functionalities into the idle task. The idle task hook is always called during each cycle of the idle task, meaning that any functionality implemented in the hook is executed during the idle task cycle.

performed the procedure to enable the L2C-310 cache controller and the cache is initialized with a predefined set of configuration: 16 ways and 1024KB.

2. L2 Cache Disable: When a secure monitor is performed and R0 holds the value 0xFFFF_FFE1, the procedure to disable the L2C-310 cache controller is performed.
3. LTZVisor Read hypercall: This hypercall allows the non-secure guest to perform a secure read under the supervision of the LTZVisor. When this hypercall is performed R0 holds the hypercall's ID and R1 the read address. At the end of the procedure R0 is used to return the read value.
4. LTZVisor Write hypercall: When called R1 holds the address of the write operation and R2 the value to be written, R0 as usual holds the hypercall ID.

To implement the hypercalls described above, it was defined the functions listed in 5.8 in the Linux Kernel 'thead.S' file.

```

1  .section ".text" , "ax"
2  .arch_extension sec
3
4  ENTRY(ltzvisor_l2_cache_enable)
5      mov r0, =0xFFFFFFE2
6      SMC #0
7  ENDPROC(ltzvisor_l2_cache_enable)
8  ENTRY(ltzvisor_l2_cache_disable)
9      mov r0, =0xFFFFFFE1
10     SMC #0
11  ENDPROC(ltzvisor_l2_cache_disable)
12  ENTRY(ltzvisor_read_register)
13     mov r1, r0
14     mov r0, =0xFFFFFFE0
15     SMC #0
16  ENDPROC(ltzvisor_read_register)
17  ENTRY(ltzvisor_write_register)
18     mov r1, r0
19     mov r0, =0xFFFFFDF
20     SMC #0
21  ENDPROC(ltzvisor_write_register)

```

Listing 5.8: Hypercalls definitions.

The hypercalls listed above can be referenced or called in every part of the Kernel. For example, both `ltzvisor_l2_cache_disable` and `ltzvisor_read_register` hypercalls are referenced in Kernel's `system.c` file in order to initialize the L2 cache controller. In the listing 5.9 is presented

the adjustment made in the `imx_init_l2cache` function, enabling the non-secure to write the L2 cache enable bit.

```

1 void __init imx_init_l2cache(void)
2 {
3     void __iomem *l2x0_base;
4     struct device_node *np;
5     unsigned int val;
6     np = of_find_compatible_node(NULL, NULL, "arm,pl310-cache");
7     if (!np)
8         return;
9     l2x0_base = of_iomap(np, 0);
10    if (!l2x0_base)
11        goto put_node;
12    if (!(ltzvisor_read_register(l2x0_base + L2X0_CTRL) &
13        L2X0_CTRL_EN)) {
14        /* Configure the L2 PREFETCH and POWER registers */
15        - val = readl_relaxed(l2x0_base + L310_PREFETCH_CTRL);
16        - val |= L310_PREFETCH_CTRL_DBL_LINEFILL |
17        - L310_PREFETCH_CTRL_INSTR_PREFETCH |
18        - L310_PREFETCH_CTRL_DATA_PREFETCH;
19        /* Set prefetch offset to improve performance */
20        - val &= ~L310_PREFETCH_CTRL_OFFSET_MASK;
21        - val |= 15;
22        - writel_relaxed(val, l2x0_base + L310_PREFETCH_CTRL);
23        + ltzvisor_l2_cache_enable();
24    }
25    iounmap(l2x0_base);
26    put_node:
27    of_node_put(np);
28 }

```

Listing 5.9: Hypercalls definitions.

One can see by the number of hypercalls implemented that the changes performed in the Linux Kernel are minimal, which is due to the secure guest small resources utilization. Since its not necessary the protection of the resources not used by the secure OS, these can be configured to become accessible in the non-secure state. The changes required in the non-secure guest varies depending on the features/resources of the board used in the secure world and non-secure guest. For instance, since the implementation conducted here serves as a demonstration of LTZVisor the resources footprint in the secure OS are minimal. Thus, hypercalls are only required in the

L2 cache enabling. Additional changes would be required if any resource used by the non-secure world is configured as secure.

5.2.4 Multicore Extension

The foundation created in the single-core implementation allows the migration from the single-core configuration to a multicore implementation as a straightforward process. The Cortex-A9 MPCore by default only initializes the core 0, while the remaining cores are kept disabled, i.e., in the single core configuration these are ignored. The multicore extension of LTZVisor implements procedures to enable and configure the secondary cores. In both multicore configurations after the core is enabled the process of configuring the core is the following: (I) first is set each stack of the processor's modes, (II) then it is configured the co-processors registers such as the NSCAR, ACR and SCR, following the same setup made to the processor 0 explained previously. From this point on the execution flow of the secondary cores varies depending on the intended multicore configuration:

1. Shared configuration: In this configuration all cores are configured in the same way. After configuring the co-processors all cores are prepared to run in the non-secure Linux OS: (I) both L1 and L2 caches are initialized and invalidated, (II) An SMC call is performed to switch the CPU to non-secure and (III) finally, the CPU jump to the Linux Entry point.
2. Dedicated configuration: In the dedicated configuration processors 1 and 2 are configured the same way as in the shared configuration since those are the cores used together with the processor 0 to run Linux in a SMP. The 4th processor is kept in the secure mode and setup to run the FreeRTOS guest by: (I) first is set each stack of the processor's mode, (II) the co-processors registers are configured and (III) FreeRTOS hardware specific devices for the 4th core such as the timer tick and the GIC controller are configured.

Natively, the process of enabling the secondary cores is entitled to the Linux Kernel, since the registers used to enable to cores are only accessible in secure mode. Therefore, an additional hypercall is created to enable Linux to initialize the secondary cores (the cores 1-3 in the shared configuration and 1-2 in the dedicated configuration). With the ID 0xFFFF_FFDE this hypercall passes the ID of the processor to be enabled in R1 and the core Linux entry point on R2.

To implement the hypercall to assist the non-secure OS in accessing secondary cores, the function listed in 5.10 is added to the Linux Kernel 'thead.S file.

```

1  ENTRY(ltzvisor_wake_secondary_cpu)
2  mov r1, r0
3  mov r0, =0xFFFFFDE
```

```
4  ENDPROC(ltzvisor_wake_secondary_cpu)
```

Listing 5.10: ltzvisor_wake_secondary_cpu hypercall.

The hypercall is used when enabling the secondary cores as shown in listing 5.11. The `imx_set_cpu_jump` function defined in the Linux Kernel `platsmp.c` file is changed in order to allow the non-secure OS to wake secondary cores.

```
1  static int imx_boot_secondary(unsigned int cpu, struct task_struct
    *idle)
2  {
3      imx_set_cpu_jump(cpu, v7_secondary_startup);
4      - imx_enable_cpu(cpu, true);
5      + ltzvisor_wake_secondary_cpu(cpu);
6      return 0;
7  }
```

Listing 5.11: Linux secondary cores enabler function.

The native function `imx_enable_cpu`, is substituted with LTZVisor's `ltzvisor_wake_secondary_cpu` hypercall. Upon a call, an SMC hands the execution to the hypervisor that will enable and configure the secondary core.

6. Evaluation

The LTZVisor implementation carried out in this thesis was evaluated in i.MX 6Quad Sabre Devices Board which is based on the ARM Cortex-A9 MPCore (MCIMX6Q-SDB). The evaluation was performed with both, LTZVisor single implementation and LTZVisor including the multicore extension. In both implementations the processors are running at 792 MHz. LTZVisor and both guests were compiled using the GNU Arm Embedded Toolchain. Since turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program, LTZVisor was evaluated with compilation optimizations disabled (-O0), with the objective of witnessing LTZVisor in the worst case scenario. The secure guest is evaluated with FreeRTOS Kernel (v10.0.1) running with MMU, L1 data and instruction cache enabled with 1MB page tables entries, L2 cache and branch prediction disabled. The non secure partition was evaluated with the Freescale i.MX Linux Tree (v4.15.0).

The evaluation performed focused on three metrics: (i) memory footprint by measuring the amount of memory used by LTZVisor, (ii) performance overhead by evaluating the overhead added to the guests execution comparatively to their native version, (iii) interrupt latency by calculating the time that elapses from when an interrupt is generated to when the source of the interrupt is serviced.

6.1 Memory Footprint

Memory Footprint refers to the amount of main memory that a program uses or references while running. LTZVisor being an lightweight hypervisor, aims to withhold a small memory footprint. LTZVisor's memory footprint was evaluated with the assistance of ARM Embedded Toolchain's size tool, that lists the individual section sizes and the total size for each of the object or archive files. LTZVisor total size is presented in the table 6.1, the performed measurement only includes the hypervisor specific files, i.e., excluding the board specific files. The hypervisor only utilizes 4180 bytes (4.08203KB) of memory. The main reasons behind such a low memory footprint are related to the principle of minimal implementation followed during LTZVisor design which (I) relies on hardware support provided by the TrustZone and (II) the careful design and configuration of its component.

It was also evaluated the secure world OS (FreeRTOS) memory footprint before and after its paravirtualization. In every configurations it is used the FreeRTOS version 10.0.1 including: task.o, timer.o, heap_2.o, port.o, queue.o, list.o, and portASM.o. Table 6.1 depict the memory footprint of: (I) native FreeRTOS handling interrupt as IRQ, (II) native FreeRTOS handling interrupt as FIQ, (III) paravirtualized FreeRTOS running on LTZVisor's secure world (vFreeRTOS).

	Memory Footprint			Total
	.text	.data	.bss	
LTZVisor	3315	0	865	4180
FreeRTOS IRQ (v10.0.1)	29129	0	10828	39957
FreeRTOS FIQ (v10.0.1)	29326	0	10828	40154
vFreeRTOS FIQ (v10.0.1)	30781	0	11128	41909

Table 6.1: LTZVisor Memory Footprint (bytes)

As the data shows, the transition from the native version using IRQ to the configuration where interrupts are served as FIQ presents a slight increase of 0.5005% in the memory footprint. From the native IRQ configuration to the paravirtualized configuration there is an increase of 4.885%. The slight increase in the secure guest memory footprint is completely acceptable and encompasses the small modifications and adaptations made in the virtualized FreeRTOS.

6.2 Performance

In this section is presented the LTZVisor's performance evaluation performed on both the secure and the non-secure guests. The secure guest evaluation is divided in two experiments: (I) the measurement of the secure OS interrupt latency by evaluating the micro-operations¹ of the guest-switching operation, (II) the measurement of the secure guest performance is evaluated with the Thread-Metric Benchmark Suite. The non-secure guest performance is evaluated with the LMBench Suite Tool.

6.2.1 Partition Context-Switching

LTZVisor partition context-switching is evaluated using the platform General Purpose Timer (GPT), that measure the time consumed by each partition context-switching micro-operation. For each micro-operation it was collected 1000 samples, and the presented result represents the average and the standard deviation of the collected samples. In the evaluation, the partition context-switching process was decomposed in 8 micro-operations that together represents the entire partition context switching process from both secure to non-secure and from non-secure to secure worlds is illustrated in the figure 6.1.

¹The micro-operations designation refers to the atomic operations of a full partition context-switching. The collection of all micro-operations is equivalent of a full partition switching

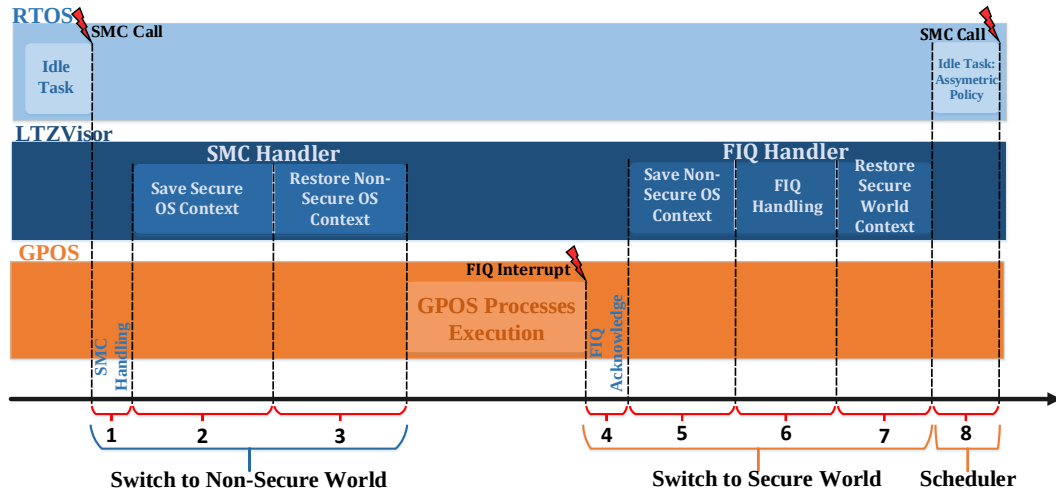


Figure 6.1: LTZVisor Partition Context Switching Process flow.

The absolute context-switching time is calculated by combining the individual micro-operations execution time. The micro-operations evaluated in LTZVisor partition context switching are:

1. SMC Handling: The secure guest OS schedules the idle task. The idle task performs a secure monitor call that is responsible for invoking the hypervisor (SMC). SMC Handling is the time since the processor performs the SMC call until LTZVisor completes the SMC handling;
2. Save secure guest OS context: LTZVisor handles the SMC request and saves the context of the secure guest OS. This micro-operation represents the time to save the current state of the secure guest OS to its respective VMCB;
3. Restore non-secure guest OS context: LTZVisor saves the context of the secure guest OS and then restores the context of the non-secure guest OS. It represents the time to restore the state of the non-secure guest OS from its respective VMCB;
4. FIQ acknowledge: The non-secure guest OS is running while a secure interrupt is triggered (e.g., RTOS timer tick). It represents the time since the interrupt occurs until LTZVisor acknowledges the FIQ in the monitor's vector table;
5. Save non-secure guest OS context: LTZVisor acknowledges the FIQ request and then saves the context of the non-secure guest OS. It represents the time to save the current state of the non-secure guest OS to its respective VMCB;
6. FIQ handling: LTZVisor saves the context of the non-secure guest OS and then immediately handles the FIQ request. It represents the time since the hypervisor save the current state of the non-secure guest OS until LTZVisor completes the FIQ handling;

World Switch	Micro-operation	Performance		Time @66MHz
		$\bar{x}$	s	
Switch to non-secure	(1) SMC handling	85.584	1.991	1296.72(ns)
	(2) Save S Guest OS context	204.74	9.527	3102.15(ns)
	(3) Restore NS guest OS Context	195.71	4.927	2965.35(ns)
Switch to secure	(4) FIQ Acknowledge	144.95	13.990	2196.24(ns)
	(5) Save NS guest OS Context	267.07	7.313	4046.57(ns)
	(6) FIQ Handling	1021.57	15.129	15478.38(ns)
	(7) Restore S Guest OS Context	133.04	7.578	2015.77(ns)
Scheduler	(8) Assymetric Policy	206.55	2.064	3129.66(ns)

Table 6.2: LTZVisor performance statistics (clock cycles).

7. Restore secure guest OS context: LTZVisor handles the FIQ and then restores the context of the secure guest OS. It represents the time to restore the state of the secure guest OS from its respective VMCB.
8. Scheduler: LTZVisor restores the execution of the RTOS. The RTOS continues executing the idle task loop and verifies if there are real-time tasks to run. If not, the idle task performs a system call (SMC) that is responsible for invoking the hypervisor. It represents the time since the processor restores the idle task execution until it execute the secure monitor call.

The collected data from the micro-operations evaluation are presented in the Table 6.2. As it can be seen, the complete partition context-switch operation takes around 34.227 microseconds, assuming that there's no real-time tasks ready to run once the RTOS is rescheduled. The process of verifying, from the RTOS prospective if there are no real-time tasks to run and hence, triggering the switch to the non-secure world, takes around 3.129 microseconds. The process of switching from the RTOS to the GPOS takes just 7.363 microseconds, and is the most deterministic activity of the partition switching operation. The evaluation demonstrated less than ten timer tick cycles of deviation from the average value (for each individual activity). Once the GPOS is executing and a FIQ is triggered, the hypervisor ensures a 6.242 microseconds of interrupt latency, and 17.493 microseconds for the RTOS to be restored. The FIQ handling operation is the major source of non-determinism of the partition-switching operation.

6.2.2 Secure OS

The RTOS running in LTZVisor secure world is evaluated using the Thread-Metric Benchmark Suite that consists of a set of benchmarks properly conceived to evaluate RTOS performance. The suite comprises 7 micro-benchmarks:

1. Cooperative Scheduling (CS): This test consists of 5 threads created at the same priority that voluntarily release control to each other in a round-robin fashion. Each thread increments its counter and relinquish the execution to the next thread. The results are presented as a sum of the individual counters.
2. Preemptive Scheduling (PS): This test consists of 5 threads each with unique priority. All threads except the lowest priority thread are left in a suspended state. The lowest priority thread will resume the next highest priority thread. That thread will resume the next highest priority thread and so on until the highest priority thread executes. Each thread will increment its run count and then call thread suspend. Eventually the processing will return to the lowest priority thread and once again resume the next highest priority thread starting the whole process over once again.
3. Interrupt Processing (IP): This test consists of a single thread. The thread will cause an interrupt, which will result in a call to the interrupt handler. The interrupt handler will increment a counter and then post to a semaphore. After the interrupt handler completes, processing returns to the test thread that initiated the interrupt. The thread then retrieves the semaphore set by the interrupt handler, increments a counter and then generates another interrupt.
4. Interrupt Preemption Processing (IPP): This test is similar to the previous interrupt test. The difference is that interrupt handler in this test resumes a higher priority thread, which causes thread preemption. The higher priority thread then suspends, resuming lower priority thread.
5. Synchronization Processing (SP): This test consists of a thread getting a semaphore and then immediately releasing it. After the get/put cycle completes, the thread will increment its counter.
6. Message Passing (MP): This test consists of a thread sending a 16 byte message to a queue and retrieving the same 16 bytes message from the queue. After the send/receive sequence is complete, the thread increments its counter.
7. Memory Allocation (MA): This test consists of a thread allocating a 128-byte memory block and releasing the same block. After the block is released, the thread will increment its run counter.

The value of the counter in each micro-benchmark test represents the RTOS impact on the running application. The higher the value the smaller is the impact and better performance result. The micro-benchmarks were executed in the native version of FreeRTOS (N_IRQ) where interrupts are handled as IRQs, the modified version of FreeRTOS where interrupts are handled as FIQs (N_FIQ),

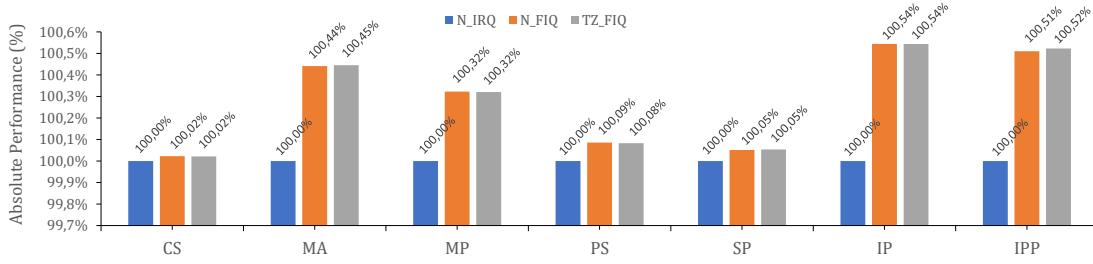


Figure 6.2: Thread Metrics benchmark results on LTZVisor's single-Core and shared multicore configuration.

and then compared against the virtualized version (TZ_FIQ). This experiment was conducted on various configurations of LTZVisor: single-core, shared multicore and dedicated multicore.

6.2.2.1 LTZVisor's Single-Core and Shared Multicore Configuration

The secure OS in the single-core and shared core configurations undergo the same conditions in this experiment and for this reason the result of both configurations are combined in the same graphic. Figure 6.2 presents the obtained results, corresponding to the average relative performance (as well as the average absolute performance) of 100 collected samples for each micro-benchmark. Each sample reflects the benchmark score for a 30 seconds execution time, encompassing a total execution time of 50 minutes, per benchmark.

As shown in figure 6.2, in every micro-benchmarks test there is a slight performance increase when FreeRTOS interrupts are handled with FIQ (N_FIQ) compared to the native version of FreeRTOS (N_IRQ). This performance increase is totally understandable considering FIQ's lower hardware interrupt latency in comparison to IRQ. Figure 6.2 also illustrates the performance results of FreeRTOS under LTZVisor where the (TZ_FIQ) performance is equal to the N_FIQ. This accomplishment is due to TZ_FIQ control over the hypervisor scheduling, as thread Metrics execution flow never gets to idle task and so, the paravirtualized OS is never interrupted by the hypervisor. This shows that as long as the secure OS possesses real-time tasks, LTZVisor's performance overhead is null.

The normalized variation of the collected Thread Metrics data over the different versions of FreeRTOS are illustrated in Figures 6.3, 6.4 and 6.5.

Figure 6.3 depicts the variations in the micro-benchmark results conducted in the native version of FreeRTOS (N_IRQ) and the standard deviation of the different results samples from its average value are the following: (I) CS: 0,01495%, (II) PS: 0,1027%, (III) IP: 0,01954%, (IV) IPP: 0,0039%, (V) SP: 0,0195%, (VI) MP: 0,0677% and (VII) MA: 0,1027%.

The modified FreeRTOS, handling interrupts as FIQ (N_FIQ) results variations is presented in figure 6.4. The results shows a slight variation reduction in the micro-benchmarks except the cooperative scheduling test, the main source of determinism deprivation in N_FIQ FreeRTOS. The standard deviation of the different micro-benchmarks samples from its average value are: (I)

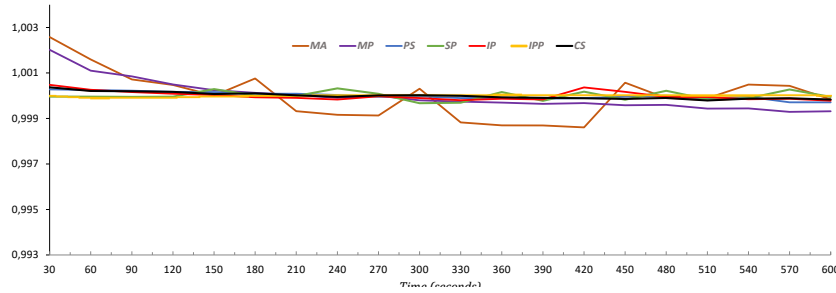


Figure 6.3: N_IRQ Thread Metrics Benchmark Results Variation.

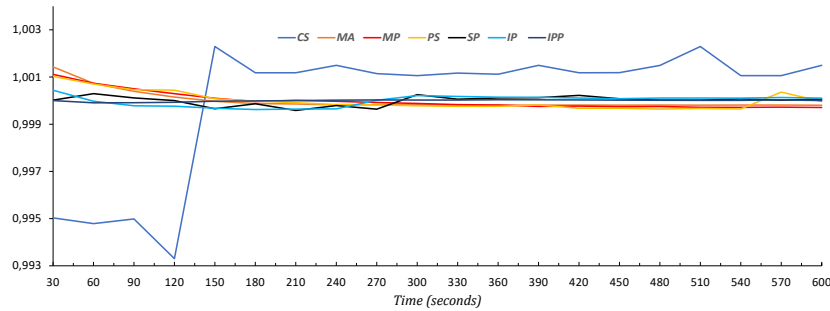


Figure 6.4: N_FIQ Thread Metrics Benchmark Results Variation.

CS: 0,2848%, (II) PS: 0,0392%, (III) IP: 0,0228%, (IV) IPP: 0,0039%, (V) SP: 0,0198%, (VI) MP: 0,0382% and (VII) MA: 0,0412%.

The same happens in the TZ_FIQ version of FreeRTOS presented in the figure 6.5. As a result of the similarities in N_FIQ and TZ_FIQ, the variations in the benchmarks results in both configurations are similar. The standard deviation for the TZ_FIQ FreeRTOS are: (I) CS: 0,2847%, (II) PS: 0,0389%, (III) IP: 0,0229%, (IV) IPP: 0,0040%, (V) SP: 0,0196%, (VI) MP: 0,0383% and (VII) MA: 0,0423%.

6.2.2.2 LTZVisor's Dedicated Multicore Configuration

The thread metrics results obtained in the dedicated multicore configuration are depicted in the figure 6.6. The results shows a decrease of performance in the TZ_FIQ version of FreeRTOS when compared to the native version.

The decrease of performance varies from 0.039% in the message passing test to 7.86% in the cooperative scheduling test. Although the performance decrease occurs in the virtualized version of FreeRTOS (TZ_FIQ), this matter is not a result of the virtualization techniques applied to the guest, but a result of the multicore performance asymmetry phenomenon observed in the hardware multiprocessing architecture where the primary processor (core 0) possesses a better performance than the secondary processors (core 1,2 and 3). Since in the dedicated multicore configuration FreeRTOS was evaluated in the processor 3, this phenomenon is the root cause of the TZ_FIQ performance loss. One may question, why not the adoption of core 0 for FreeRTOS

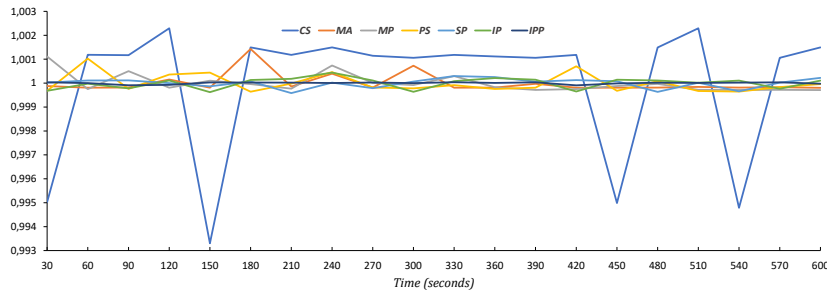


Figure 6.5: TZ_FIQ Thread Metrics Benchmark Results Variation.

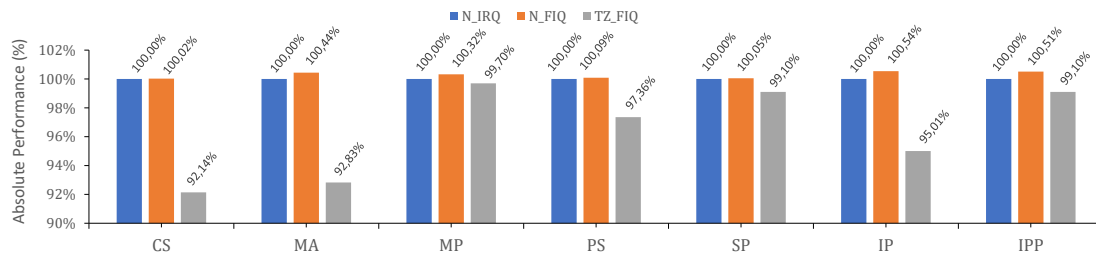


Figure 6.6: Thread Metrics benchmark results on LTZVisor's dedicated multicore configuration.

when designing and implementing the dedicated multicore configuration due to its superior performance capabilities. The answer is based of a trade-off between the design/implementation cost with the secure OS performance. Because Linux is designed and implemented to use the platform's core 0 as its primary core, dedicating core 0 to FreeRTOS would require an adaptation of the Linux Kernel to run in the secondary cores. For this reason during the hypervisor's design it was decided for the usage of FreeRTOS in the core 3 and maintain Linux using core 0 as its primary core.

Figure 6.7 illustrate the normalized variation of the collected micro-benchmarks data in the TZ_FIQ FreeRTOS. The standard deviation of the different results samples from their average values are the following: (I) CS: 0,0569%, (II) PS: 0,0363%, (III) IP: 0,0184%, (IV) IPP: 0,0039%, (V) SP: 0,0058%, (VI) MP: 0,0771% and (VII) MA: 0,0722%.

6.2.2.3 Interrupt Latency

Interrupt latency is the measurement of the system's response-time to an interrupt, which corresponds to the elapsed time between interrupt assertion and the instant when a response happens. It may be affected by architecture design, interrupt controllers, interrupt masking, and the operating system's (OS) interrupt handling methods. Equation 6.1 expresses LTZVisor's latency when handling an interrupt: (I) τ_H is the hardware dependent time which depends on

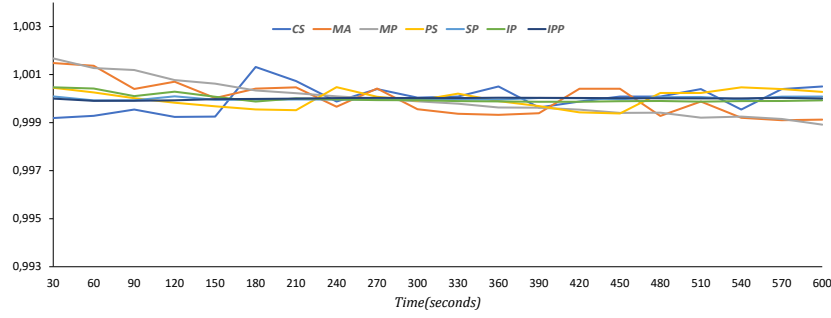


Figure 6.7: TZ_FIQ Thread Metrics Benchmark Results Variation.

the interrupt controller, (II) τ_{OS} is the OS-specific induced overhead, (III) τ_{HYP} is the hypervisor-specific induced overhead.

$$\tau_{IL} = \tau_H + (\tau_{HYP}) + \tau_{OS} \quad (6.1)$$

Applying the equation in the evaluation process, the following conclusions were achieved:

1. **τ_H :** In the evaluation it was concluded that the interrupt latency in the native FreeRTOS (N_IRQ) is 2,206 microsecond. The adjustment made in the native FreeRTOS by changing interrupt handling to FIQ instead of IRQ improved FreeRTOS interrupt latency to 2,196 microseconds. This slight decrease in the interrupt latency is a result of FIQ lower interrupt hardware latency compared to IRQ. This conclusion is common in all LTZVisor's implemented configurations.
2. **τ_{HYP} :** Latency overhead induced by LTZVisor only occurs during RTOS idle time (has no real-time ready-to-run tasks) in LTZVisor's single-core and shared multicore configurations. Being the two configurations where secure processor is shared with the non-secure world, additional overhead is added during the world switch. Since LTZVisor runs with all interrupt sources disabled, the worst case scenario happens when the FIQ interrupt arrives while a context switch from the secure to the non-secure world is starting. In this case, the request is handled with a worst case interrupt latency of 12,128 microseconds. This is a very rare situation, because two asynchronous and independent events need to occur at the same time: (I) an asynchronous FIQ needs to be triggered while (II) a world switch is happening. Nevertheless, since the overhead introduced on latency has a deterministic upper bound (12,128 microseconds), it can be taken into account when designing the real-time system. In LTZVisor's multicore configuration τ_{HYP} is null, since the processor 3 is dedicated to the secure OS there's no additional overhead in the interrupt handling induced by the Hypervisor.
3. **τ_{OS} :** The overhead induced by the FreeRTOS is common to every configuration of LTZVisor.

6.2.3 Non-Secure OS

The Linux OS running in the non-secure world was evaluated with the widely used ANSI/C micro-benchmarks for UNIX/POSIX, Lmbench. The suite provides benchmarks that measure the most commonly found performance bottlenecks in a wide range of system applications. It measures the latency and bandwidth of data movement among the processor and memory, network, file system, and disk. The benchmarks focus on latency and bandwidth because performance issues are usually caused by latency problems, bandwidth problems, or some combination of the two. The benchmarks version 3 used in the evaluation comprises more than fourty micro-benchmarks. Under LTZVisor it is focused in two specific benchmarks:

1. **lat_ops:** Arithmetic operations latency, to evaluate general CPU performance (VFP and Neon are disabled);
2. **bw_mem:** Memory operations bandwidth for different blocks size (1K, 100K, 1M and 10M), to evaluate the interference of TZASC, L2 cache as well as Level 1 (4-way set-associative 32 KB) and data caches.

The non-secure OS evaluation was conducted through two experiments. In the first experiment the non-secure OS is evaluated without the interference of the secure world. The secure OS is configured with no real time task and 1 millisecond tick rate. Under this condition the non-secure world OS is only interrupted by the monitor in the tick interrupt and since there is no real time task, the execution is always redirected to the non-secure OS. The experiment is performed in every LTZVisor's implemented configurations. Due to the similarities in the `bm_mem` test result, this is only presented once and it is common to all configurations. For each micro-benchmarks it was performed 10 times 1000 consecutive experiments. Prior each consecutive experiments it was performed 100 warm-ups (-W 100 -N 1000), meaning that for each micro-benchmark it was performed 1000 warm-ups and collected 10000 samples.

Figures 6.8a, 6.8b, 6.8c and 6.8d presents the average absolute and relative performance marks of the native Linux version (N) and the virtualized version (TZ) running the `bm_mem` test. The figures also includes the variations of the individuals micro-benchmarks samples collected.

In 6.8a it is presented the results of native and virtualized Linux's memory bandwidth benchmark for 1 KBytes of memory, showing a slight decrease of performance in the virtualized version comparatively to the native version. Specifically: (I) 0,0812% in rd, (II) 0,111% in wr, (III) 0,0959% in rdwr, (IV) 0,147% in cp, (V) 0,128% in fwr, (VI) 0,229% in fcp and (VII) 0,580% in bcopy. In the bcopy benchmark the virtualized Linux outperforms the native one by 19,609%. Even with some smalls difference, throughout these experiments the performance between the native and virtualized Linux are proportional.

In figure 6.8b, 6.8c and 6.8d it is used 100 KBytes, 1 MBytes and 10 Mbytes memory blocks. It is verified a loss of performance in all micro-benchmarks in a factor of more than

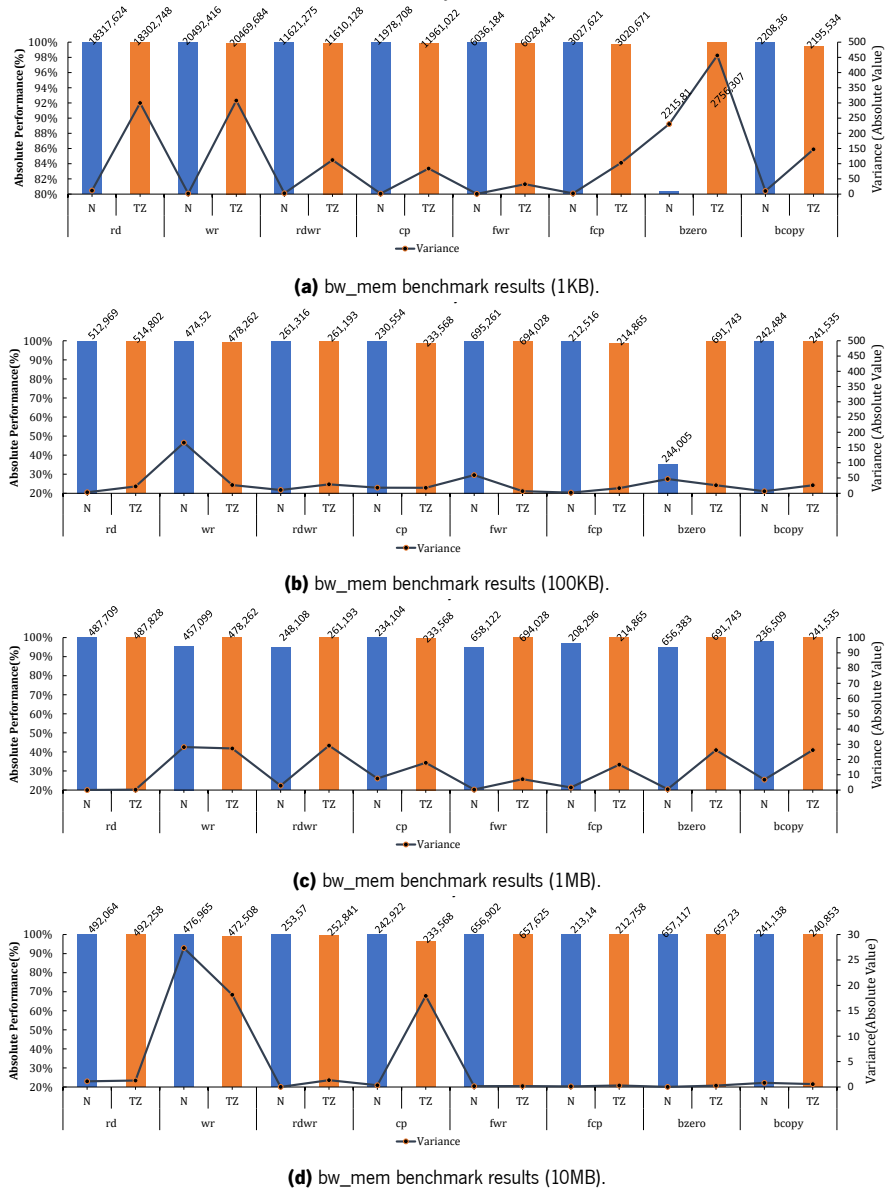


Figure 6.8: LMBench memory bandwidth (bw_mem) benchmark results.

97% in some benchmarks compared to using 1 Kbyte block. The enormous loss of performance verified shows the effects of using the cache on the memory bandwidth performance. Since L2 cache was disabled and L1 cache only has 32 KBytes, by using a block memory bigger than L1 cache the performance drops drastically, and the higher the memory block gets the greater is the performance loss. To test the effect of L2 cache, the same experiments performed above were repeated but with L2C-310 cache controller enabled with 16 ways and 1024 kB and the result of the experiment is depicted in figures 6.9a, 6.9b, 6.10a, and 6.10b. The results shows a great improvement in performance when handling bigger memory block compared to the setup in 6.8. The configuration with L2 cache enabled outperforms the setup with L2 disabled by a factor of: (I) up to 1.642% when using 1KByte of memory, (II) up to 77.7% when using 100 KBytes

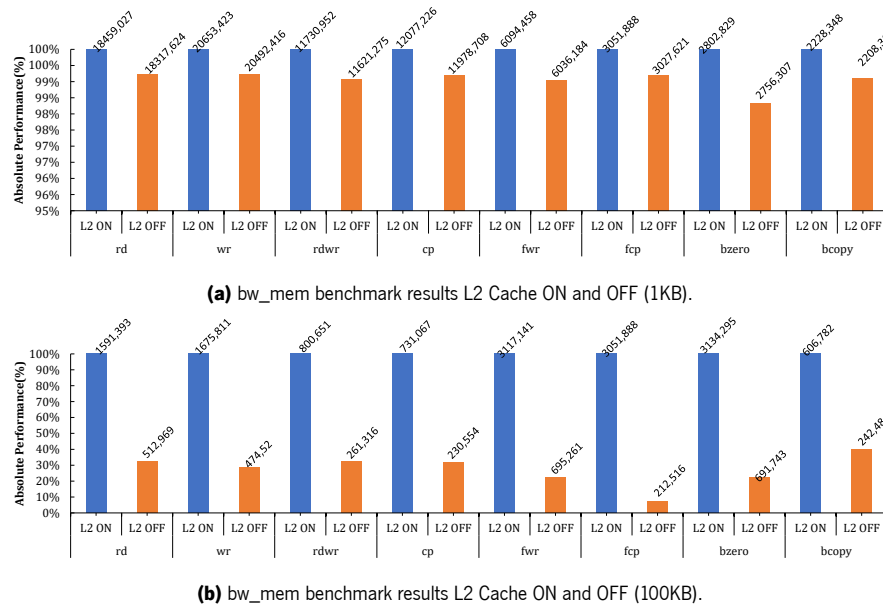


Figure 6.9: LMBench memory bandwidth benchmark L2 Cache Effect (part 1).

of memory, (III) up to 74% when using 1MBytes of data and (IV) up to 64,7% when handling 10 MBytes of memory. With this experiment's results, it's clear that L2 cache possesses an important role regarding the system's memory performance.

The next experiment done during the non-secure GPOS evaluation was measuring the latency of basic CPU operations using Lmbench3 lat_ops command. The experiment was conducted individually in each configuration of LTZVisor, and the achieved results are presented in the figures: (I) 6.11a and 6.11b for the single-core configuration, (II) 6.12a and 6.12b for the shared multicore configuration, (III) 6.13a and 6.13b for the dedicated multicore configuration. The values on top of each bar correspond to the average operation latency in nanoseconds. As depicted in figures 6.11a and 6.11b the virtualized Linux in the single-core configurations has a slight increase of overhead in the CPU operations compared to the native single core configuration of Linux and it is up to 12% in the worst case scenario. In the shared configuration CPU latency measurements also verified a slight increase in the operations latency overhead in the virtualized Linux. In the dedicated configuration the virtualized Linux running in SMP mode with 3 cores (core 0, 1 and 2) is compared to the native Linux running with 3 cores. Even having the 4th core dedicated to the secure world running FreeRTOS with no real-time task the virtualized Linux performance matches the Native configuration with the highest latency at 0.257%.

The last part of the non-secure OS evaluation is measuring the effect of the secure-OS timer tick in the non-secure world OS. For this experiment the non-secure world OS performance was measured with the secure OS tick varying from 100 microseconds to 10 seconds. For each tick rate tested, it was collected 11000 samples of the non-secure world performance metrics (10000 samples + 1000 Warm-ups). The secure OS during the experiment was configured similarly to

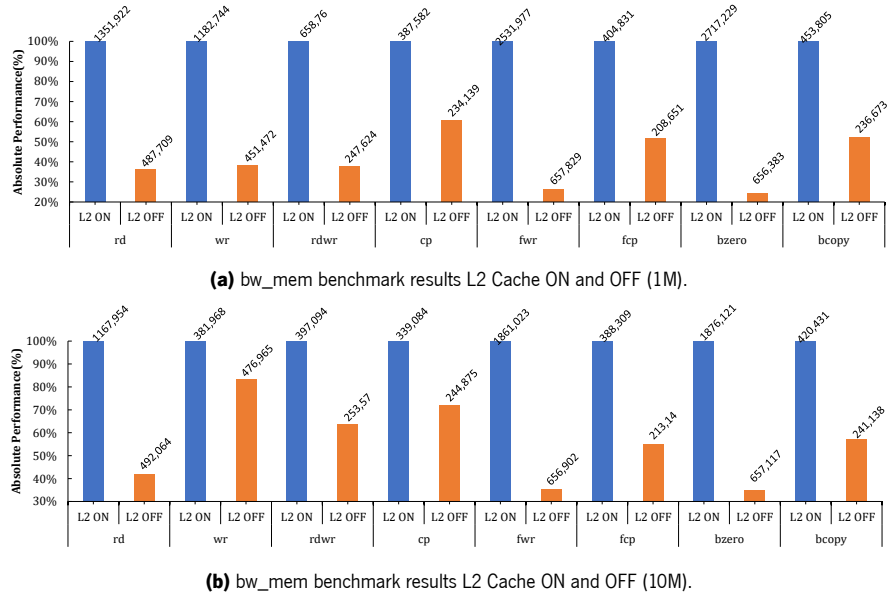


Figure 6.10: LMBench memory bandwidth benchmark L2 Cache Effect (part 2).

the previous experiments, With no real-time task the overhead added to the non-secure world is only related to the servicing of the secure OS tick interrupt. The timer tick was setup in 5 different scenarios: 100, 200, 500, 5000, and 10000 microseconds. The experiment was performed individually in every LTZVisor implemented configurations and the achieved results is depicted in figure 6.14. The results clearly show that in the configurations where the secure core is shared (sigle-core and shared multicore) the performance penalties are significant when the switching rate is high. In the single-core configuration the performance overhead varies from 0% to 35% in the worst case while in the shared multicore configuration the highest performance drops is around 29%. This test really shows the advantage of having a processor dedicated to the secure world and throughout the 5 different experiments the Linux performance only drops around 3% when the rate is 10KHz.

6.3 Discussion

From the beginning of this dissertation we have instigated that when exploited adequately, virtualization can enhance embedded application's security and consolidate real-time and non-real-time tasks in the same scope. The LTZVisor's implementation conducted in this thesis have confirmed and demonstrated this claim. Relying on hardware to assist virtualization, LTZVisor allows the consolidation of two isolated virtual machines, the first one hosting an RTOS with real-time tasks and the second VM hosting a GPOS with non-real-time tasks while supporting both single and quad-core configuration. In configurations where a single processor is used to execute

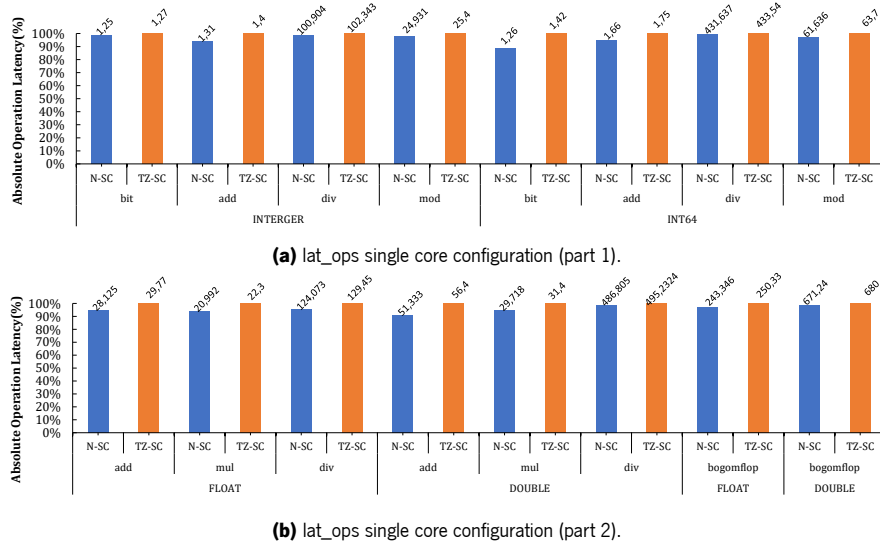


Figure 6.11: Lmbench lat_ops benchmark results.

both RTOS's real-time tasks and GPOS's non-real-time tasks, the asymmetric scheduling implemented in LTZVisor dictated by the RTOS, allows the RTOS to even by sharing the processor with the GPOS meet its timing requirements. Keeping the spotlight in LTZVisor's RTOS timing requirements, the configurations implemented and evaluated in this thesis have shown that in all them the asymmetric scheduling was capable of meeting LTZVisor's RTOS timing requirements with a slight overhead and determinism deteriorations, such deteriorations are added in the configurations where the secure core is shared with the non-secure OS due to the context-switching from the non-secure world to the secure world to service the secure interrupts. The marginal overhead and determinism deteriorations found in the single-core and shared multicore configuration are eliminated by the dedicated multicore configuration where a processor (core 3) is dedicated to the secure world RTOS. The Non-secure OS performance results showed similar results between the shared multicore where the non-secure OS uses 4 cores with the dedicated multicore configuration where is used 3 cores. Even using only 3 cores by having the RTOS on a dedicated core, the dedicated multicore can achieve similar performance results as the quad-core version. This is mainly due to having Linux main core uninterrupted.

Regarding security measurements LTZVisor is able to provide a strong spatial isolation of the secure components from the non-secure access through:

1. TZASC, LTZVisor partitioning the system's memory in two regions each assigned to one VM. The i.MX 6Quad possesses the CoreLink TrustZone Address Space Controller TZC-380 capable of dividing the main memory up to 16 regions each individually configurable according to the intended access policy. In the evaluation process, it was demonstrated that overhead added by using TZASC to virtualizes the systems memory its null. In a previous implementation of LTZVisor for the Xilinx ZC702 depicted in [125] Ngabonziza et

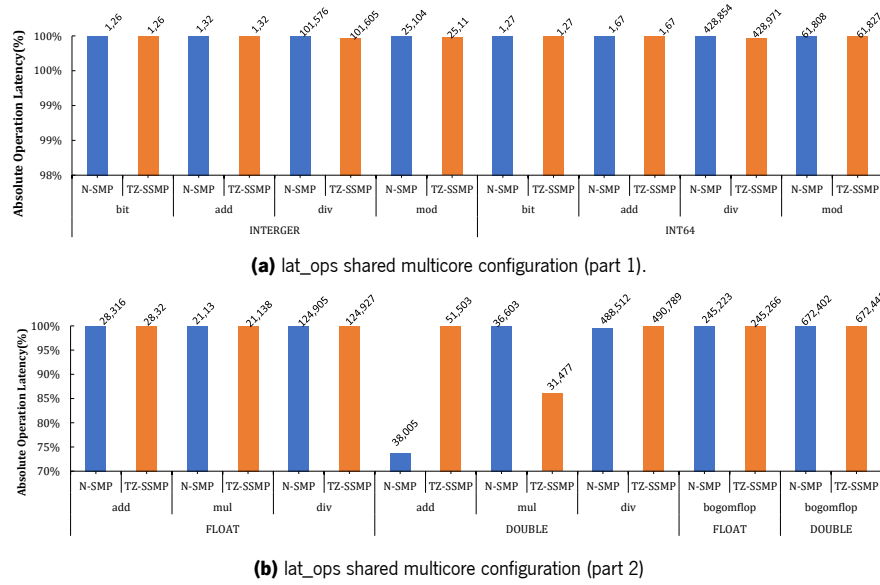
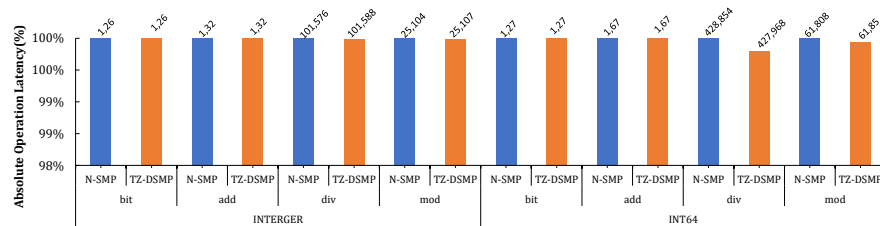


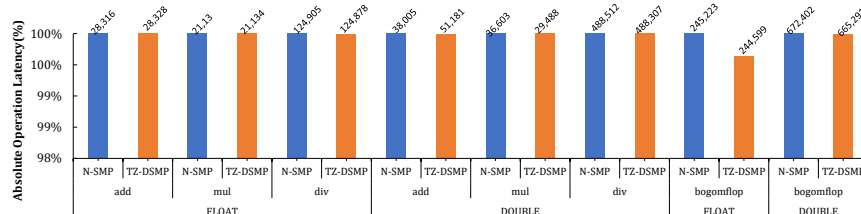
Figure 6.12: Lmbench lat_ops benchmark results.

al. [134] presented some doubts on ways how it could be prevented the secure access to the non-secure world, by having LTZVisor's secure OS with full visibility and access of the entire system as well as how can it be prevented the secure OS to access the non-secure world. Sandro et al. presented some possible solutions such as having both guests in the non-secure world as demonstrated in [135] or paravirtualizing the RTOS to run in the user mode of the secure world and while mediating each memory access through the hypervisor. Both of these solutions possesses some casualty compared to the current implementation of LTZVisor. In this implementation of LTZVisor it is possible to achieve the total isolation of the non-secure memory space from the secure OS without the need of changing LTZVisor current design. A feature present in TZC-380 known as the Security Inversion allows to configure any memory region with the security inversion. When active it enables a region to be accessible to masters in non-secure state but not accessible to masters in Secure state. So, relying on this feature present in the i.MX 6Quad TZASC controller LTZVisor can guarantee the isolation of the non-secure OS memory space by activating this feature and use the lock bits register to lock such feature from further changes. The implementation of LTZVisor conducted in this thesis allows the enabling of this feature. It is also important to reminds that this feature only isolates the non-secure world memory space while the secure world RTOS can still access non-secure peripherals. To provide a complete isolation of non-secure OS additional mechanisms are required to ensure the isolation of non-secure peripherals.

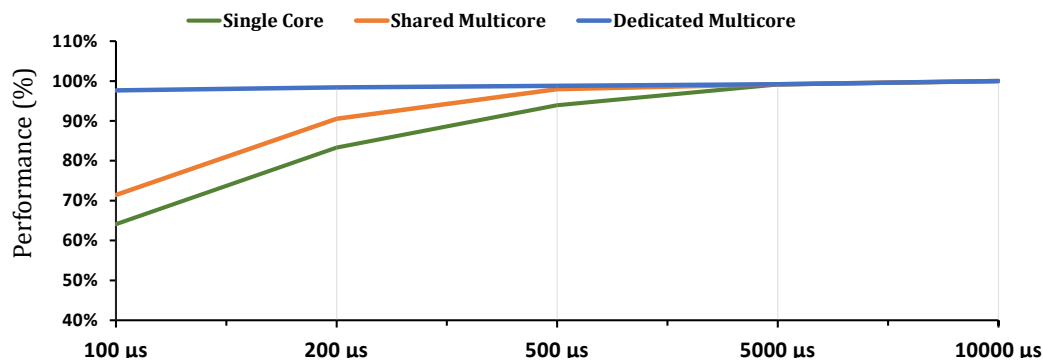
2. CSU, LTZVisor isolates the secure peripherals from non-secure accesses through i.MX 6Quad central security unit, that manages the system security policy for peripheral access



(a) lat_ops dedicated multicore configuration (part 1).



(b) lat_ops dedicated multicore configuration (part 2).

Figure 6.13: Lmbench lat_ops benchmark results.**Figure 6.14:** Guest-switching rate vs performance overhead.

on the SoC.

3. PL390, the PrimeCell Generic Interrupt Controller(PL390) presents on the Cortex-A9 MPCore support the implementation of GICv2 that allows the servicing of interrupt as GIC or FIQ. LTZVisor isolates the secure and non-secure interrupts by servicing secure interrupts as FIQ and non-secure interrupts as IRQ. This separation supports two advantageous features to the LTZVisor's design: the first one is that by using FIQ for the secure interrupt there is a slight performance improvement as a result of FIQ lower hardware latency which was seen during the secure OS performance evaluation and the second is the higher priority nature of FIQ interrupts compared to IRQ. With the secure guest using all its interrupts as FIQ it can run unbothered of the non-secure world interrupts.
4. MMU, when used in both worlds the MMU present in the Cortex-A9 MPCore allows the isolation of secure entries from non-secure access relying fully on hardware.

Another point outlined in [125] is the possibility of starvation in the non-secure OS as a result of the secure OS control over the hypervisor's scheduling policy. In the implementation conducted in this thesis such behavior arise in the single core and the shared multicore configuration where the primary non-secure OS core is the same used by the secure OS. This issue is eliminated in the dedicated multicore configuration where the secure world and the non-secure run in separated processors.

Architecturally LTZVisor is limited to two VM's one secure and the other in the non-secure world, but this can easily be changed specially in a multiprocessing architecture such as i.MX 6Quad where configuration possibilities are enormous.

7. Conclusion

Embedded software and application have already reshaped the world we live in and our daily routine to a point where its literally impossible living without benefiting from services provided by any sort of embedded application. And for the foreseeable future it is expected an expansion on the relationship between humans and machine. More and more devices are leveraging of cloud based services through IoT technology. For instance, automotive industries are relying heavily on embedded devices features for the next generation of cars and vehicles, hospitals and medical facilities are including more and more embedded devices and applications into their workspace for monitoring or performing delicate operations, military equipments are becomes more automated and embedded devices are even being incorporated in our bodies to assist and/or monitors our well being, etc.. Many of these applications possesses critical operations where security is paramount. Virtualization has been a technique widely used to enhance embedded devices security requirements through software-only or hardware-assisted approaches. Virtualization also allows the creation of heterogeneous systems by allowing the consolidation of real-time and non-real-time tasks in the same architecture.

Software-only virtualization techniques are usually deployed in the server and desktop platforms, due to is substantial virtualization overhead while in resources constrained systems with hard-real-time tasks where determinism is demanded often opted hardware-based virtualization. ARM the major embedded architecture developer provides the Security Extension (TrustZone) for its Cortex A and M architectures widely used in embedded systems.

In this thesis, LTZVisor, a lightweight TrustZone-assisted hypervisor implemented with the objective of instigating TrustZone-assisted virtualization is ported to the i.MX 6Quad SD board. LTZVisor uses TrustZone to create two isolated execution environment, the secure world used for a small TCB privileged RTOS and the non-secure world used for a rich de-privileged GPOS. Prior this Thesis LTZVisor had support to Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit, ZedBoard Zynq-7000 ARM/FPGA SoC Development Board and Zybo Zynq-7000 ARM/FPGA SoC Trainer Boards supporting single-core and dual-core implementation. The LTZVisor's implementation presented in this thesis not only ported the platform to i.MX 6Quad but also extend the multicore implementation to a quad-core configuration. Two quad-core configurations are supported: (I) a shared configuration, where the processor 0 is the secure core used to run the

secure OS while the non-secure world is executed in all the platforms processors. In order to implement that, the secure core is assigned to the non-secure OS when the secure world is in the idle state, (II) a dedicated multicore, where the secure OS is executed in the core 3 and the non-secure in the remaining cores. On the contrary of the share multicore, here the secure core only runs the secure OS. Although LTZVisor by default uses FreeRTOS as the secure world RTOS and Linux as the non-secure world GPOS, the implementation done in this thesis also support baremetal secure VM and baremetal and FreeRTOS as the non-secure VM.

The evaluation conducted in the virtualized LTZVisor's secure OS of all the individual configurations have shown that this can still perform well compared to the native version and even sometimes outperforming the native version due to the usage of FIQ to service the secure world OS instead of the native IRQ. Notice that the lower hardware latency of FIQ causes a marginal performance improvement. The virtualized non-secure OS also performs well compared to the native version. Results have demonstrated that this technology can effectively satisfy the strict requirements for virtualizing a real-time environment, while offering a low performance cost on running an unmodified guest GPOS.

7.1 Future Work

The LTZVisor's implementation conducted in this thesis have allowed the consolidation of a secure isolated RTOS with a non-secure GPOS. The implementations leverage the multiprocessing nature of the i.MX 6Quad to explore three different configurations of the VM's processor assignment.

However, there are two key improvements on which it would benefit the guest performance or bring needed functionalities into the system:

1. Regarding the guest performance, during the evaluation process it was shown the architectural performance superiority of the i.MX 6Quad primary processor (core 0) compared to the secondary cores (core 1, 2 and 3). In the dedicated multicore configuration the secure RTOS was assigned to the core 3 meaning that in that configuration if assigned to the core 0, the secure RTOS would have had a better performance (for being in the primary core) compared to being in the secondary core 3. The motive behind the assignment of a secondary processor to the non-secure OS is due to the dependency existent in the used non-secure OS (Linux) to run with the core 0 as its primary core. Since the OS used in the secure world (FreeRTOS) has no dependency on the core 0 it was decided that instead of changing the Linux kernel to run in the secondary core, the secure OS would be used in the secondary core. An interesting future improvement would be shipping Linux to a secondary core and uses the real-time OS in the processor with the highest performance.

2. Regarding functionalities, in the LTZVisor implemented in this thesis there is no communication between the two guests. So, a compelling future improvement would be the implementation of an inter-VM communication. In LTZVisor's implementation depicted in [125] it is used an adaptation of the RPMsg API of the Texas Instrument and OpenAMP group, supporting the signaling of interrupts to send events. It was designed to promote asynchronous communication and guarantee the timing requirements of the secure VM.

References

- [1] G. E. Moore, "Readings in computer architecture," pp. 56–59, 2000. [Online]. Available: <http://dl.acm.org/citation.cfm?id=333067.333074>
- [2] Y. Lai, "Techniques for Techniques High Speed A / D Converters in Real Time Systems," *Imid* 2009, no. December, pp. 1069–1072, 2009.
- [3] R. Sgandurra, "Strategies for Deploying Xilinx's Zynq UltraScale + RFSoc," Tech. Rep. 201.
- [4] "Rob Persons Senior System Architect Artesyn Embedded Technologies," Tech. Rep., 2018. [Online]. Available: <https://www.militaryaerospace.com/content/mae/en/whitepapers/2018/09/cots-bladed-server-architecture-for-high-performance-defense-applications.whitepaperpdf.render.pdf>
- [5] "Improve Visual Clarity In Live Video zmicro," Tech. Rep., 2017. [Online]. Available: <https://www.militaryaerospace.com/whitepapers/2018/04/zmicro-insight-system-white-paper.whitepaperpdf.render.pdf>
- [6] L. Pike, N. Wegmann, S. Niller, and A. Goodloe, "Copilot: monitoring embedded systems," *Innovations in Systems and Software Engineering*, vol. 9, no. 4, pp. 235–255, Dec 2013. [Online]. Available: <https://doi.org/10.1007/s11334-013-0223-x>
- [7] R. Vyas, A. Bhattacharjee, and A. Khandelwal, "Development of soc for interfacing avionics units with arinc 429 bus," in *2016 International Conference on Communication and Electronics Systems (ICCES)*, Oct 2016, pp. 1–5.
- [8] A. Alptekin, Y. Yilmazer, U. Usug, F. Koca, and K. Incki, "Design for arinc 653 conformance: Architecting independent validation of a safety-critical rtos," in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, Oct 2014, pp. 8A6–1–8A6–8.
- [9] A. Sangiovanni-Vincentelli and M. D. Natale, "Embedded system design for automotive applications," *Computer*, vol. 40, no. 10, pp. 42–51, Oct 2007.
- [10] W. H. Oh, J. H. Lee, H. G. Kwon, and H. J. Yoon, "Model-based development of automotive embedded systems: a case of continuously variable transmission (cvt)," in *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, Aug 2005, pp. 201–204.

- [11] T. Massey, F. Dabiri, R. Jafari, H. Noshadi, P. Brisk, W. Kaiser, and M. Sarrafzadeh, "Towards reconfigurable embedded medical systems," in 2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP 2007), June 2007, pp. 178–180.
- [12] T. Jingwen, L. Erhong, and G. Meijuan, "Network multi-parameters medical treatment monitoring instrument based on embedded system," in 2007 8th International Conference on Electronic Measurement and Instruments, Aug 2007, pp. 3–1005–3–1008.
- [13] W. H. Wu, A. A. Bui, M. A. Batalin, L. K. Au, J. D. Binney, and W. J. Kaiser, "Medic: Medical embedded device for individualized care," *Artificial Intelligence in Medicine*, vol. 42, no. 2, pp. 137 – 152, 2008, wearable Computing and Artificial Intelligence for Healthcare Applications. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0933365707001510>
- [14] S. Pinto, J. Cabral, and T. Gomes, "We-care: An iot-based health care system for elderly people," in 2017 IEEE International Conference on Industrial Technology (ICIT), March 2017, pp. 1378–1383.
- [15] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares, "IloTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices," *IEEE Internet Computing*, vol. 21, no. 1, pp. 40–47, jan 2017. [Online]. Available: <http://ieeexplore.ieee.org/document/7839870/>
- [16] S. Pinto, "Secure and safe virtualization-based framework for embedded systems development," mar 2017. [Online]. Available: <http://repositorium.sdum.uminho.pt/handle/1822/48679>
- [17] "AMD-V™ Technology for Client Virtualization," <https://www.amd.com/en/technologies/virtualization>.
- [18] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, May 2005.
- [19] J. Winter, "Trusted computing building blocks for embedded linux-based arm trustzone platforms," in Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, ser. STC '08. New York, NY, USA: ACM, 2008, pp. 21–30. [Online]. Available: <http://doi.acm.org/10.1145/1456455.1456460>
- [20] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares, "Lightweight multicore virtualization architecture exploiting ARM TrustZone," *Proceedings IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, vol. 2017-January, pp. 3562–3567, 2017.
- [21] S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares, "FreeTEE: When real-time and

- security meet,” IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, vol. 2015-October, 2015.
- [22] J. Winter, “Experimenting with arm trustzone – or: How i met friendly piece of trusted hardware,” in 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, June 2012, pp. 1161–1166.
- [23] S. Pinto and N. Santos, “Demystifying Arm TrustZone: A Comprehensive Survey,” ACM Computing Surveys, vol. preprint.
- [24] Amit Singh, “An Introduction to Virtualization,” <http://www.kernelthread.com/publications/virtualization/>, 2014.
- [25] R. C. D. Fernando J. Corbató, Marjorie Merwin Daggett, “An Experimental Time-Sharing System,” <http://webcache.googleusercontent.com/search?q=cache:http://larch-www.lcs.mit.edu:8001/corbato/sjcc62/>, 1962.
- [26] K. Burrage, L. Hood, and M. a. Ragan, ADVANCED SERVER VIRTUALIZATION VMware® and Microsoft® Platforms in the Virtual Data Center, 2006, vol. 7, no. 4. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/20435543>
- [27] F. Corbató, M. Daggett, R. Daley, P. Denning, D. A. Grier, R. Mills, R. Roach, and A. Scherr, “Compatible Time-Sharing System (1961-1973) Fiftieth Anniversary Commemorative Overview,” System, 1973.
- [28] R. J. Creasy, “The Origin of the VM / 370 Time-sharing System,” IBM Journal of Research and Development, vol. 25, no. 5, pp. 483–790, 1981.
- [29] John McCarthy, “Reminiscences on the Theory of Time-Sharing,” <http://jmc.stanford.edu/computing-science/timesharing.html>, 1983.
- [30] “CTSS,” <http://www.piercefuller.com/library/ctss.html>, 2004.
- [31] J. Hoopes, Virtualization for security : including sandboxing, disaster recovery, high availability, forensic analysis, and honeypotting. Syngress Pub, 2009.
- [32] Computer History: IBM 360/370/3090/390. <http://www.beagle-ears.com/lars/engineer/comphist/ibm360.htm>.
- [33] “CMPSCI 691ST Software Systems Lecture 5,” Tech. Rep., 2011. [Online]. Available: <https://people.cs.umass.edu/~emery/classes/cmpsci691st/scribe/lecture5-system360.pdf>
- [34] M. Varian, “Vm and the vm community: Past, present, and future,” SHARE, 1997.
- [35] J. R. Levine, Linkers and loaders. Morgan Kaufmann, 2000.
- [36] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” Communications of the ACM, vol. 17, no. 7, pp. 412–421, 1974. [Online].

- Available: <http://portal.acm.org/citation.cfm?doid=361011.361073>
- [37] M. Uddin and A. Abdul Rahman, "Server consolidation: An approach to make data centers energy efficient and green," *International Journal of Scientific and Engineering Research*, vol. 1, 10 2010.
 - [38] "Server Consolidation and Containment With Virtual Infrastructure," Tech. Rep., 1998. [Online]. Available: <http://www.vmware>.
 - [39] J. Shropshire, "Analysis of monolithic and microkernel architectures: Towards secure hypervisor design," in *2014 47th Hawaii International Conference on System Sciences*, Jan 2014, pp. 5008–5017.
 - [40] Sharifah Setapa, Mohd Anuar Mat Isa, Nazri Abdullah, and Jamalul-hail Ab Manan, "Trusted computing based microkernel," *Computer Applications and Industrial Electronics (ICCAIE 2010)*, no. Iccae, pp. 309–312, 2010. [Online]. Available: <http://ieeexplore.ieee.org/xpls/abs{ }all.jsp?arnumber=5771164>
 - [41] John Bell, "Operating Systems: Structures," https://www.cs.uic.edu/~jbell/CourseNotes/-OperatingSystems/2_Structures.html, 2006.
 - [42] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: Design and Implementation*. Prentice Hall, 1997.
 - [43] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Tolerating hardware device failures in software," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 59–72. [Online]. Available: <http://doi.acm.org/10.1145/1629575.1629582>
 - [44] A. Ganapathi, V. Ganapathi, and D. Patterson, "Windows XP Kernel Crash Analysis," Tech. Rep. [Online]. Available: <http://www.microsoft.com/whdc/>
 - [45] A. Kadav and M. M. Swift, "Understanding Modern Device Drivers," *ASPLOS XVII Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pp. 87–98, 2012. [Online]. Available: <http://delivery.acm.org/10.1145/2160000/2150987/p87-kadav.pdf?ip=193.137.92.214{&}id=2150987{&}acc=ACTIVESERVICE{&}key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35{&}{ }acm{ }{ }=1534990003{ }2bb63afba80f762e09ddfed2e45c6f20>
 - [46] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha, "Microdrivers: A New Architecture for Device Drivers," *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, no. May, pp. 15:1—15:6, 2007. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1361397.1361412>
 - [47] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity

- operating systems,” in Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 207–222. [Online]. Available: <http://doi.acm.org/10.1145/945445.945466>
- [48] Jochen Liedtke, “On u-Kernel Construction,” German National Research Center for Information Technology, pp. 1–14, 1995. [Online]. Available: [http://delivery.acm.org/10.1145/230000/224075/p237-liedtke.pdf?ip=193.137.92.101&id=224075&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35&acm\[\]=\[\]=1534709513_e0b1770d2b5458e6d114c713ff5caead](http://delivery.acm.org/10.1145/230000/224075/p237-liedtke.pdf?ip=193.137.92.101&id=224075&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35&acm[]=[]=1534709513_e0b1770d2b5458e6d114c713ff5caead)
- [49] —, “TOWARD REAL MICROKERNELS,” vol. 39, pp. 70–77, 1996. [Online]. Available: [http://delivery.acm.org/10.1145/240000/234473/p70-liedtke.pdf?ip=193.137.92.101&id=234473&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35&acm\[\]=\[\]=1534709401_e0b1770d2b5458e6d114c713ff5caead](http://delivery.acm.org/10.1145/240000/234473/p70-liedtke.pdf?ip=193.137.92.101&id=234473&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35&acm[]=[]=1534709401_e0b1770d2b5458e6d114c713ff5caead)
- [50] T. Xia, J. C. Prevotet, and F. Nouvel, “Mini-NOVA: A Lightweight ARM-based Virtualization Microkernel Supporting Dynamic Partial Reconfiguration,” Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2015, no. Vm, pp. 71–80, 2015.
- [51] J. S. S. Michael Hohmuth, Michael Peter, Hermann Hartig, “Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors,” EW 11 Proceedings of the 11th workshop on ACM SIGOPS European workshop Article No. 22, 2004. [Online]. Available: [http://delivery.acm.org/10.1145/1140000/1133615/p22-hohmuth.pdf?ip=193.137.92.202&id=1133615&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35&acm\[\]=\[\]=1534760018_41d12518ac86c1dec4137349d6f73975](http://delivery.acm.org/10.1145/1140000/1133615/p22-hohmuth.pdf?ip=193.137.92.202&id=1133615&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35&acm[]=[]=1534760018_41d12518ac86c1dec4137349d6f73975)
- [52] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg, “The performance of microkernel-based systems,” in Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, ser. SOSP '97. New York, NY, USA: ACM, 1997, pp. 66–77. [Online]. Available: <http://doi.acm.org/10.1145/268998.266660>
- [53] C. M. Kirsch, M. A. A. Sanvido, and T. A. Henzinger, “A programmable microkernel for real-time systems,” in Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, ser. VEE '05. New York, NY, USA: ACM, 2005, pp. 35–45. [Online]. Available: <http://doi.acm.org/10.1145/1064979.1064986>
- [54] B. Roch, “Monolithic kernel vs. microkernel,” 10 2018.
- [55] F. Bruns, S. Traboulsi, D. Szczesny, E. Gonzalez, Y. Xu, and A. Bilgic, “An evaluation of

- microkernel-based virtualization for embedded real-time systems,” Proceedings - Euromicro Conference on Real-Time Systems, pp. 57–65, 2010.
- [56] H. Massalin, “Synthesis: An efficient implementation of fundamental operating system services,” Ph.D. dissertation, New York, NY, USA, 1992, uMI Order No. GAX92-32050.
- [57] B. W. Larnpson, “Hints for Computer System Design,” Proceedings of the Seventh ACM Symposium on Operating Systems Principles, pp. 98–105, 1979. [Online]. Available: http://delivery.acm.org/10.1145/810000/806614/p33-lampson.pdf?ip=193.137.92.178&id=806614&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35&{}_{}_acm{}_{}_=1534903105{}_d5a2453f7b1f28649dbb5a852e8da098
- [58] E. A. Thomas, B. N. Bershad, E. D. Lazowska, and H. M. Levy, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism,” Proc. Thirteenth Symposium on Operating System Principles, pp. 95–109, 1991. [Online]. Available: http://delivery.acm.org/10.1145/130000/121151/p95-anderson.pdf?ip=193.137.92.178&id=121151&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.BB8FB5C9967023B9.4D4702B0C3E38B35&{}_{}_acm{}_{}_=1534903078{}_c60136872c9dc77586eca4f6a41f8332
- [59] D. R. Engler, M. F. Kaashoek, and J. O’Toole, Jr., “Exokernel: An operating system architecture for application-level resource management,” in Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, ser. SOSP ’95. New York, NY, USA: ACM, 1995, pp. 251–266. [Online]. Available: <http://doi.acm.org/10.1145/224056.224076>
- [60] K. Adams and O. Agesen, “A comparison of software and hardware techniques for x86 virtualization,” SIGARCH Comput. Archit. News, vol. 34, no. 5, pp. 2–13, Oct. 2006. [Online]. Available: <http://doi.acm.org/10.1145/1168919.1168860>
- [61] H. Lee, “Virtualization basics: Understanding techniques and fundamentals,” 2014.
- [62] Z. Amsden, D. Arai, D. Hecht, A. Holler, P. Subrahmanyam, and V. Inc, “Vmi: An interface for paravirtualization,” Linux Symposium, vol. 2, 2006.
- [63] “AMD-V™ Nested Paging,” Tech. Rep., 2008. [Online]. Available: <http://developer.amd.com/wordpress/media/2012/10/NPT-WP-11-final-TM.pdf>
- [64] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, “Efficient memory virtualization: Reducing dimensionality of nested page walks,” in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 178–189. [Online]. Available: <http://dx.doi.org/10.1109/MICRO.2014.37>

- [65] X. L. Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, "Selective Hardware/Software Memory Virtualization," in VEEACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE). New York: ACM New York, NY, USA ©2011, 2011, pp. 217–226. [Online]. Available: http://delivery.acm.org/10.1145/1960000/1952710/p217-wang.pdf?ip=193.137.92.58&id=1952710&acc=ACTIVESERVICE&key=2E5699D25B4FE09E.8B0EA5264B27B16E.4D4702B0C3E38B35.4D4702B0C3E38B35&{}_{}_acm{}_{}_=1534443138{}_5bf8ecd55333f4ce36f5992f4385092e
- [66] J. Gandhi, M. Hill, and M. Swift, "Agile paging: Exceeding the best of nested and shadow paging," IEEE Micro, pp. 1–1, 2018.
- [67] I. Corporation, "Intel ® Virtualization Technology for Directed I / O," Order A Journal On The Theory Of Ordered Sets And Its Applications, vol. 10, no. October, pp. 1–152, 2011.
- [68] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." pp. 423–436, 01 2010.
- [69] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization," in USENIX 2008 Annual Technical Conference, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1404014.1404017>
- [70] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, no. January 2010, pp. 1–10, 2010. [Online]. Available: <http://ieeexplore.ieee.org/document/5416637/>
- [71] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: <http://doi.acm.org/10.1145/945445.945462>
- [72] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance vmm-bypass i/o in virtual machines," in Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 3–3. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1267359.1267362>
- [73] A. Limited, "ARM ® System Memory Management Unit Architecture Specification SMMU architecture version 2.0," Tech. Rep., 2012. [Online]. Available: <http://www.arm.com/about/trademark-usage-guidelines.php>.

- [74] I. Corporation, "Intel ® Virtualization Technology for Directed I/O Architecture Specification," Tech. Rep., 2018. [Online]. Available: <http://www.intel.com/info/em64t>
- [75] Amd, "AMD I/O Virtualization Technology (IOMMU) Specification," Tech. Rep. [Online]. Available: <http://www.bis.doc.gov/>.
- [76] P. Willmann, S. Rixner, and A. L. Cox, "Protection strategies for direct access to virtualized i/o devices," in USENIX 2008 Annual Technical Conference, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 15–28. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1404014.1404016>
- [77] P. Willman, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel, "Concurrent direct network access for virtual machine monitors," HPCA, 2007.
- [78] D. E. Williams and J. R. García, Virtualization with Xen: including XenEnterprise, XenServer, and XenExpress, 2007. [Online]. Available: <http://books.google.com/books?hl=en&lr=&id=FfenKMYn8sUC&pgis=1>
- [79] XENProject, "XEN Hypervisor WebSite," Homepage, pp. 1–3, 2014. [Online]. Available: <http://www.xenproject.org/developers/teams/hypervisor.html>
- [80] S. Spector and X. Org, "Why Xen?" Tech. Rep. [Online]. Available: <http://www-archive.xenproject.org/files/Marketing/WhyXen.pdf>
- [81] "Device Passthrough to Driver Domain in Xen," Tech. Rep. [Online]. Available: <https://wiki.xen.org/images/1/17/Device{ }passthrough{ }xen.pdf>
- [82] S. Horman, "Xen Multi-Function PCI Pass-Through," Tech. Rep. [Online]. Available: <https://www.valinux.co.jp/english/doc/20091023{ }03.pdf>
- [83] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in 2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT), Oct 2011, pp. 39–48.
- [84] L. Zeng, Y. Wang, W. Shi, and D. Feng, "An improved xen credit scheduler for i/o latency-sensitive applications on multicores," in Proceedings of the 2013 International Conference on Cloud Computing and Big Data, ser. CLOUDCOM-ASIA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 267–274. [Online]. Available: <https://doi.org/10.1109/CLOUDCOM-ASIA.2013.40>
- [85] S. Toumassian, R. Werner, and A. Sikora, "Performance measurements for hypervisors on embedded arm processors," in 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Sept 2016, pp. 851–855.
- [86] R. Hnarakis, "In perfect xen, a performance study of the emerging xen scheduler," 12 2013.

- [87] Xen Community, "Xen Project Schedulers - Xen," https://wiki.xenproject.org/wiki/Xen_Project_Schedulers.
- [88] M. Åsberg, N. Forsberg, T. Nolte, and S. Kato, "Towards real-time scheduling of virtual machines without kernel modifications," in ETFA2011, Sept 2011, pp. 1–4.
- [89] A. Masrur, S. Drossler, T. Pfeuffer, and S. Chakraborty, "VM-Based Real-Time Services for Automotive Control Applications," in 2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications. IEEE, aug 2010, pp. 218–223. [Online]. Available: <http://ieeexplore.ieee.org/document/5591645/>
- [90] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee, "Real-time multi-core virtual machine scheduling in xen," in 2014 International Conference on Embedded Software (EMSOFT), Oct 2014, pp. 1–10.
- [91] S. Wu, L. Zhou, X. Wang, F. Chen, and H. Jin, "Doris: An Adaptive Soft Real-Time Scheduler in Virtualized Environments," IEEE Transactions on Services Computing, no. October, 2017.
- [92] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Supporting soft real-time tasks in the xen hypervisor," SIGPLAN Not., vol. 45, no. 7, pp. 97–108, Mar. 2010. [Online]. Available: <http://doi.acm.org/10.1145/1837854.1736012>
- [93] A. Motakis, A. Rigo, and D. Raho, "Platform Device Assignment to KVM-on-ARM Virtual Machines via VFIO," Proceedings - 2014 International Conference on Embedded and Ubiquitous Computing, EUC 2014, pp. 170–177, 2014.
- [94] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor." [Online]. Available: <http://dx.doi.org/10.1145/2541940.2541946>
- [95] —, "Supporting KVM on the ARM architecture [LWN.net]," <https://lwn.net/Articles/557132/>, 2018-08-30.
- [96] R. Ma, F. Zhou, E. Zhu, and H. Guan, "Performance tuning towards a kvm-based embedded real-time virtualization system," J. Inf. Sci. Eng., vol. 29, pp. 1021–1035, 2013.
- [97] G. Taccari, L. Taccari, A. Fioravanti, L. Spalazzi, and A. Claudi, "Embedded Real - Time Virtualization: State of the Art and Research Challenges," Rtlws '14, vol. 00128, no. October, 2014.
- [98] M. El and M. Aichouch, "Evaluation of a multiple criticality real-time virtual machine system and configuration of an RTOS's resources allocation techniques," Tech. Rep. [Online]. Available: <https://tel.archives-ouvertes.fr/tel-01127450>
- [99] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization," Journal of Software Engineering and Applications, vol. 5, pp. 277–290, 01 2012.

- [100] M. Masmano, I. Ripoll, A. Crespo, and J. J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," 2009.
- [101] Z. Qingguo, R. Bucher, R. Maintainer, S. Don Carr, S. Choo, W. Llc, S. Carsten Emde, G. Philippe Gerum, X. Maintainer, F. Thomas Gleixner, G. Nicholas Mc Guire, P. Mantegazza, I. Peter Okech, S. University, K. Zhou Qingguo, and B. Zagar, "Eleventh Real-Time Linux Workshop Program committee," Tech. Rep., 2009. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.385.2999{&}rep=rep1{&}type=pdf{#}page=273>
- [102] M. Masmano, I. Ripoll, and A. Crespo, "An overview of the xtratum nanokernel," 01 2005.
- [103] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The xtratum approach," in 2010 European Dependable Computing Conference, April 2010, pp. 67–72.
- [104] I. Ripoll, A. Crespo, M. Masmano, V. Brocal, S. Peiró, P. Balbastre, and M. Jean-Jacques, "Configuration and scheduling tools for tsp systems based on xtratum," European Space Agency, (Special Publication) ESA SP, vol. 682, pp. 30–, 08 2010.
- [105] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, "A Portable ARINC 653 standard interface," AIAA/IEEE Digital Avionics Systems Conference - Proceedings, no. March 2014, 2008.
- [106] M. Masmano, I. Ripoll, and A. Crespo, "XtratuM Hypervisor for LEON3 Volume 2: User Manual," Tech. Rep., 2011. [Online]. Available: <http://www.xtratum.org/files/xm-3-usermanual-022c.pdf>
- [107] M. Masmano, J. O. Coronel, and A. Crespo, "XtratuM Hypervisor for LEON3 Volume 4 : Reference Manual," vol. 4, 2011.
- [108] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor xvisor: A comparative analysis," Proceedings - 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, pp. 682–691, 2015.
- [109] X. community, "Xvisor: eXtensible Versatile hypervISOR," <http://xhypervisor.org/>.
- [110] K. Tindell and H. Hansson, "Real Time Systems by Fixed Priority Scheduling 1," Processing, 1997.
- [111] J. Khatri, "An Enhanced Round Robin CPU Scheduling Algorithm," IOSR Journal of Computer Engineering, vol. 18, no. 04, pp. 20–24, 2016. [Online]. Available: <http://iosrjournals.org/iosr-jce/papers/Vol18-issue4/Version-2/D1804022024.pdf>
- [112] C. R. Moratelli, "A lightweight virtualization layer with hardware-assistance for embedded systems," 2016. [Online]. Available: <http://repositorio.pucrs.br/dspace/handle/10923/9490>

- [113] H. Youness, M. Osama, and A. Tarek, "Load Balancing on CPU-GPU Heterogeneous System," Academia.Edu, no. December 2012, 2015. [Online]. Available: <http://www.academia.edu/download/38263417/Load{ }Balancing{ }on{ }CPU-GPU{ }HS.pdf>
- [114] A. Limited, "ARM ® Architecture Reference Manual ARMv7-A and ARMv7-R edition," Tech. Rep., 1996. [Online]. Available: <https://silver.arm.com/download/ARM{ }and{ }AMBA{ }Architecture/AR570-DA-70000-r0p0-00rel2/DDI0406C{ }C{ }arm{ }architecture{ }reference{ }manual.pdf>
- [115] —, "Cortex-A9 MPCore Technical Reference Manual," Tech. Rep., 2008. [Online]. Available: <http://www.arm.com>
- [116] G. Guez, "Why Hardware-Based Design Security is Essential for Every Application," Tech. Rep. [Online]. Available: www.maximintegrated.com
- [117] A. Limited, "ARM Security Technology Building a Secure System using TrustZone ® Technology," Tech. Rep., 2005. [Online]. Available: <http://www.arm.com>
- [118] —, "ARM7TDMI Technical Reference Manual," Tech. Rep., 2001. [Online]. Available: <http://www.arm.com>
- [119] —, "PrimeCell ® Vectored Interrupt Controller (PL190) Technical Reference Manual," Tech. Rep., 2000. [Online]. Available: <http://www.arm.com>
- [120] —, "ARM11 MPCore Processor Technical Reference Manual," Tech. Rep., 2005. [Online]. Available: <http://www.arm.com>
- [121] "AMBA ® AXI Protocol Specification," Tech. Rep., 2003. [Online]. Available: <http://www.arm.com>
- [122] "Trusted Platform Module (TPM) Summary," Tech. Rep. [Online]. Available: <https://www.trustedcomputinggroup.org/>.
- [123] P. Varanasi and G. Heiser, "Hardware-supported virtualization on arm," in Proceedings of the Second Asia-Pacific Workshop on Systems, ser. APSys '11. New York, NY, USA: ACM, 2011, pp. 11:1–11:5. [Online]. Available: <http://doi.acm.org/10.1145/2103799.2103813>
- [124] "Genode - An Exploration of ARM TrustZone Technology," <https://genode.org/documentation/articles/trustzone>.
- [125] S. Pinto, J. Pereira, T. Gomes, and A. Tavares, "LTZVisor : TrustZone is the Key," EcrtS, pp. 1–23, 2017.
- [126] Sierraware, "Open Virtualization - ARM TrustZone and ARM Hypervisor Open Source Software," <http://www.openvirtualization.org/>.
- [127] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is,

- and what it is not,” Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015, vol. 1, pp. 57–64, 2015.
- [128] A. Spyridakis, P. Lalov, and D. Raho, “Dual-OS Infrastructure for Mixed-Criticality Systems on ARMv8 Platforms,” The Ninth International Conference on Advanced Engineering Computing and Applications in Sciences, no. c, pp. 116–121, 2015.
- [129] Arm, “ARM Trusted Firmware,” Tech. Rep., 2012. [Online]. Available: <https://github.com/ARM-software/arm-trusted-firmware>
- [130] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX vTZ: Virtualizing ARM TrustZone vTZ: Virtualizing ARM TrustZone. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua>
- [131] N. Semiconductors, “i.MX 6Dual/6Quad Applications Processor for Consumer Products Data Sheet,” Tech. Rep. [Online]. Available: <https://www.nxp.com/docs/en/data-sheet/IMX6DQCEC.pdf>
- [132] “Interactive i.MX Pin Mux Tool.” https://www.nxp.com/webapp/sps/download/license.jsp?colCode=IMX6_IOMUX_
- [133] D. Dicarlo, “i . MX IOMUX Design Aid,” pp. 2009–2012, 2012.
- [134] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, “Trustzone explained: Architectural features and use cases,” in 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), Nov 2016, pp. 445–451.
- [135] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, “Towards a trustzone-assisted hypervisor for real-time embedded systems,” IEEE Computer Architecture Letters, vol. 16, no. 2, pp. 158–161, July 2017.