Università degli Studi di Padova
Università degli Studi di Napoli Federico II

Joint Research Doctorate in Fusion Science and Engineering
XXXVII Cycle

Real-Time Virtualization of Mixed-Criticality Heterogeneous Embedded Systems for Fusion Diagnostics and Control

Candidate: Daniele Ottaviano

Supervisor: Prof. Marcello Cinque
Co-supervisor: Prof. Gianmaria De Tommasi

Napoli, September 30, 2024

Abstract

The rise of complex, safety-critical systems, such as nuclear fusion reactors, has highlighted significant challenges in optimizing resource utilization on modern Multi-Processor Systems on chips. These platforms offer substantial computational power, but strict real-time guarantees often lead to overprovisioning. As a result, only a small fraction of available resources are effectively utilized, which can hinder the system's overall efficiency and longevity, especially when the hardware is expected to operate for decades in reactors.

This dissertation addresses these challenges by exploring real-time virtualization as a key solution for improving hardware utilization in mixed-criticality systems. It begins with an in-depth analysis and testing of virtualization technologies, focusing on enhancing temporal isolation between applications that share the same hardware. A systematic Design-of-Experiments approach is used to assess and optimize the isolation provided by different virtualization configurations.

Building on this foundation, the core contribution of this dissertation is the development of the Omnivisor, an innovative virtualization model that extends existing partitioning hypervisor architectures to manage asymmetric cores (such as ARM64, ARM32, and RISC-V) and control memory bandwidth allocation for accelerators, such as FPGAs and GPUs. The Omnivisor ensures isolation between virtual machines and predictable behavior across mixed-criticality applications, enhancing system reliability and resource efficiency.

Additionally, the RPUGuard communication framework, integrated into the Omnivisor, provides fine-grained control over the communication between virtualized asymmetric processors, reducing interference and ensuring real-time performance.

To ease the utilization of Omnivisor in distributed systems the RunPHI framework is introduced. RunPHI integrates the Omnivisor into a cloud-to-edge orchestration system, simplifying the deployment of distributed mixed-criticality systems across heterogeneous environments.

The contributions of this work are demonstrated through real-world use cases related to nuclear fusion, where the proposed models and frameworks facilitate the reliable and efficient operation of safety-critical control

applications. The evaluation phase shows how innovative virtualization techniques can allow critical controllers to co-exist with utility applications on the same hardware, ensuring they remain isolated while sharing resources.

Keywords: Mixed-Criticality, Embedded Virtualization, Real-Time Systems, MPSoCs.

Abstract (Italiano)

L'aumento della complessità nei sistemi critici, come quelli utilizzati nei reattori a fusione nucleare, ha messo in luce le sfide legate all'ottimizzazione delle risorse nei moderni Multi-Processor System-on-Chips (MPSoC). Sebbene queste piattaforme offrano una potenza di calcolo considerevole, le rigorose garanzie in tempo reale portano spesso a un sovradimensionamento delle risorse nei contesti critici. Di conseguenza, solo una frazione delle risorse disponibili viene effettivamente sfruttata, limitando l'efficienza complessiva del sistema, specialmente quando l'hardware è progettato per operare per decenni nei reattori.

Questa tesi affronta queste problematiche esplorando la virtualizzazione real-time come soluzione chiave per migliorare l'utilizzo dell'hardware nei sistemi a criticità mista. Inizialmente, viene condotta un'analisi approfondita delle tecnologie di virtualizzazione, con un focus specifico sull'isolamento temporale tra le applicazioni che condividono lo stesso hardware. Viene utilizzato un approccio sistematico basato su Design-of-Experiments per valutare e ottimizzare l'isolamento offerto da diverse configurazioni di virtualizzazione.

Il contributo centrale di questa ricerca è lo sviluppo dell'Omnivisor, un modello innovativo di virtualizzazione che estende le architetture hypervisor esistenti per gestire core asimmetrici (come ARM64, ARM32 e RISC-V) e controllare l'allocazione della larghezza di banda della memoria per processori e acceleratori come FPGA e GPU. L'Omnivisor garantisce un comportamento prevedibile nelle applicazioni a criticità mista, migliorando l'affidabilità e l'efficienza delle risorse del sistema. Inoltre, RPU-Guard, integrato nell'Omnivisor, offre un controllo preciso sulle comunicazioni tra processori asimmetrici virtualizzati, riducendo le interferenze e assicurando prestazioni in tempo reale.

Per abilitare l'orchestrazione dei sistemi a criticità mista, viene introdotto il framework RunPHI, che integra l'Omnivisor in un sistema di orchestrazione distribuita, semplificando il deployment di sistemi a criticità mista in ambienti eterogenei, dal cloud fino ai nodi edge.

I contributi di questo lavoro vengono dimostrati attraverso casi d'uso reali legati alla fusione nucleare, dove i modelli e i framework proposti facilitano il funzionamento sicuro ed efficiente delle applicazioni di con-

trollo critiche. La fase di valutazione mostra come queste tecniche di virtualizzazione permettano ai controller critici di coesistere con altre applicazioni sullo stesso hardware, mantenendo l'isolamento necessario pur condividendo le risorse.

Parole chiave: Mixed-Criticality, Embedded Virtualization, Real-Time Systems, MPSoCs.

Contents

Abstract III
Sinossi V
List of Acronyms XV
List of Figures XXII
List of Tables XXIV

1 Introduction **1**
1.1 Mixed-Criticality Systems 5
1.2 Nuclear Fusion Real-Time Control Systems 6
1.3 The Rise of Real-Time Virtualization 8
1.4 Objective and Scope of the Dissertation 11

2 Background and Related Work **15**
2.1 Real-Time Systems and Mixed-Criticality 16
2.1.1 Real-Time Task Model 16
2.1.2 Real-Time Frameworks for Nuclear Fusion 18
2.1.3 Mixed-Criticality Task Model 19
2.1.4 The Evolution of Mixed-criticality Systems model . 21
2.2 Multi-Processors Systems-on-Chip 22
2.2.1 Resources Contention in Modern Architectures . . . 23
2.2.2 Emerging and Established Architectural Trends . . . 26
2.2.3 Spatial and Temporal Protection Mechanisms 28
2.2.4 Real-Time Processing Units and Virtualization . . . 30
2.3 Towards Embedded Systems Virtualization 31
2.3.1 Static Partitioning Hypervisors Shortcoming 32
2.4 Related Work on Hypervisors for MPSoCs 33
2.4.1 Partitioning Systems 33

2.4.2 Asymmetric Multi-Core Architectures 34
2.4.3 MPSoCs Hypervisors 34

3 Temporal Assessment and Modeling of Mixed-Criticality Virtualized Systems 37
3.1 Systematic Approach and Workflow 39
3.1.1 Analysis 39
3.1.2 Testbed Setup 41
3.1.3 Design of Experiment (DoE) 42
3.2 Temporal Isolation Assessment 42
3.2.1 Step 1: the Standard Analysis 43
3.2.2 Step 2: Xen on ARM-based MPSoC Testbed Setup . 43
3.2.3 Step 3: Design of Experiments 46
3.3 Temporal Isolation Results and Analysis 48
3.3.1 *Credit2* Analysis 49
3.3.2 *RTDS* Analysis 51
3.3.3 *Credit2* vs *RTDS*: Many-to-Many Analysis 52
3.3.4 *Credit2* vs *RTDS* vs *null*: One-to-One Analysis . . . 54
3.3.5 *null* Scheduler Stress Analysis: DomU vs Dom0 . . . 56
3.3.6 Results and Lessons Learned 61
3.4 Mixed-Criticality Deployment Model 64

4 The Omnivisor: A Unified Approach to Virtualizing Heterogeneous MPSoCs 69
4.1 Unified Model for Heterogeneous Virtualization 69
4.2 The Omnivisor Architectural Design 70
4.2.1 Requirements Responsibilities and Features 71
4.2.2 Implementation Strategy 78
4.2.3 Omnivisor Usage Workflow 79
4.3 Real-Time Asymmetric Communication: RPUGuard 84
4.3.1 RPUGuard Design 85
4.3.2 RPUGuard Algorithm and Implementation 86
4.4 Virtualizing Co-Processors: Microcontroller Virtualization . 90
4.4.1 Lightweight and Predictable Virtual Memory 91
4.4.2 Hardware-Based LPVM Methods 93
4.4.3 LVPM Experimental Results 98

4.4.4 Balancing Partitioning and Flexibility in Heterogeneous Systems . . . 102
4.5 RunPHI framework for Omnivisor Orchestration . . . 103
4.5.1 Advanced Orchestration Primitives for Mixed-Criticality Systems . . . 105
4.5.2 RunPHI: Managing Mixed-Criticality Heterogeneous Systems . . . 106
4.5.3 Modeling Fault Tolerance and Timing Uncertainty in Heterogeneous Systems . . . 111

5 Evaluation of Nuclear Fusion Scenarios 115
5.1 Model-Based Fog Monitoring Evaluation in Nuclear Fusion 118
5.1.1 System Description and Specification. . . . 119
5.1.2 System Architecture Design. . . . 121
5.1.3 Prototype Development. . . . 124
5.2 Omnivisor Experimental Validation . . . 129
5.2.1 Boot Time Performance Assessment . . . 130
5.2.2 Omnivisor's Isolation Capability . . . 131
5.2.3 HIL Evaluation of Vertical Stabilization Controller . 138
5.2.4 RPUGuard Application and Validation . . . 145
5.3 runPHI Preliminary Orchestration Experiments . . . 150

6 Conclusions and Future Activities 155
6.1 Summary of Contributions . . . 156
6.2 Implications for Nuclear Fusion . . . 158
6.3 Future Research Directions . . . 159

A Publications 161

List of Acronyms

The following acronyms are used throughout the thesis.

VM Virtual Machine

PVM Privileged Virtual Machine

MCS Mixed Criticality System

SWaP-C Size, Weight, Power, and Cost

CPS Cyber-physical Systems

ISA Instruction Set Architecture

SoC System on Chip

MPSoC Multi-Processor Systems on Chip

SMP Symmetric Multi-Processing

AMP Asymmetric Multi-Processing

MMU Memory Management Unit

SMMU System Memory Management Unit

MPU Memory Protection Unit

PMP Physical Memory Protection

SMPU System Memory Protection Units

SPPU	System Peripheral Protection Unit
XMPU	Xilinx Memory Protection Unit
XPPU	Xilinx Peripheral Protection Unit
DMA	Direct Memory Access
TCM	Tightly Coupled Memory
FPGA	Field Programmable Gate Array
COTS	Commercial Off-The-Shelf
rCPU	remote CPU
CPU	Central Processing Unit
APU	Application Processing Unit
RPU	Real-Time Processing Unit
GPU	Graphical Processing Unit
PMU	Platform Management Unit
NPU	Neural Processing Unit
TPU	Tensor Processing Unit
PSCI	Power State Coordination Interface
ACPI	Advanced Configuration and Power Interface
RPMsg	Remote Processor Messaging
JET	Joint European Torus
CODAC	Control Data Access and Communication
PCS	Plasma Control System
VS	Vertical Stabilization

ES	Extremum Seeking
VDE	Vertical Displacement Event
SDN	Synchronous Databus Network
MARTe2	Multi-threaded Application Real-Time executor 2
RTF	Real-Time Framework
DoE	Domain of Experiment
PE	Processing Element
WCET	Worst Case Execution Time
BCET	Best Case Execution Time
ACET	Average Case Execution Time
WCRT	Worst Case Response Time
TL	Tail Latency
std	Standard Deviation
OCM	On Chip Memory
LLC	last-level cache
TLB	Translation Lookaside Buffer
LPVM	lightweight and predictable virtual memory
RTOS	Real-Time Operating System
ECU	Electronic Control Unit
SPH	Static Partitioning Hypervisor
QoS	Quality of Service
IPI	Inter-Processor Interrupt

ISR	Interrupt Service Routine
MSI	Message Signaled Interrupt
SGI	Software Generated Interrupt
GIC	Generic Interrupt Controller
EL	Exeption Level
WASM	WebAssembly
OCI	Open Container Initiative
CRI	Container Runtime Interface
GRT	Guarded Radix Tries
DAG	Direct Acyclic Graph
ST	Segment Trie
LT	Limit Table
STT	Segment Trie Table
RPM	Region Protection Memory
LUT	Look Up Tables
FF	Flip Flop
SIL	Safety Integrity Level
ASIL	Automotive Safety Integrity Level
SSIL	Software Safety Integrity Level
DAL	Design Assurance Level
TMR	Triple Modular Redundancy
OTA	Over-The-Air

HIL	Hardware-In-the-Loop
UDP	User Datagram Protocol
IoT	Internet of Things
ZCU	Zynq™ UltraScale+™
DPR	Dynamic Partial Reconfiguration
ENV	Local Environment
SS	Scheduling Scheme
CS	inter-VM communication scheme
TS	Task Set
RT	Real-Time Tasks
G	General Purpose Tasks
M	Monitoring Tasks
CR	Computational Resources
P	Pool
PS	Pool Set
DM	Deployment Module
S	Scheduling Algorithm
AS	Architectural Scenario
CC	Communication Channel

List of Figures

1.1 Years in which the microchip was first introduced (Licensed under CC-BY by the authors Hannah Ritchie and Max Roser) 2
1.2 From homogeneous to extreme heterogeneous architectures (reproduced from [1]). 3
1.3 Example of Heterogeneous MPSoCs 3
1.4 ITER System Overview 7
1.5 Example of Virtualization 8
1.6 Overview of the Proposed Architecture based on the Omnivisor model 11

2.1 Example of Periodic Task 17
2.2 Simplified Schematic of Zynq Ultrascale+ MPSoC 23

3.1 Proposed Temporal Isolation Assessment Workflow. 40
3.2 The Xen-based virtualized 2oo2-based testbed used for experimentation. 44
3.3 CPU affinity factor and its levels. 47
3.4 *Credit2* host-level scheduler analysis, by varying *rate limit* i.e., the minimum execution time interval of a vCPU before a context switch. 50
3.5 Analysis of *"RTDS"* host-level scheduler factor, while varying the *budget/period* in microseconds. 51

3.6 Analysis of *"Credit2"* and *"RTDS"* host-level scheduler factors, fixing the *CPU affinity* factor to the *Many-to-Many* level, and by enabling/disabling *stress* factor, CPU level. Parameters for both schedulers are configured based on the optimal results attained in prior experiments. 53
3.7 Analysis of *"Credit2"* and *"RTDS"* host-level scheduler factors, fixing the *CPU affinity* factor to the *One-to-One* level, and by enabling/disabling *stress* factor, CPU level. 55
3.8 Analysis of *"null"* host-level scheduler factor on DomU, fixing the *CPU affinity* factor to the *One-to-One* level, and varying all levels within *stress* factor (CPU, Cache, Device, I/O, Intr, FS, Net). 57
3.9 Analysis of *"null"* host-level scheduler factor on Dom0, fixing the *CPU affinity* factor to the *One-to-One* level, and varying all levels within *stress* factor (CPU, Cache, Device, I/O, Intr, FS, Net). 58
3.10 Distribution of maximum execution times for voter(dom0) and replica(domU) by fixing *host-level scheduler* factor to *null* level, the *CPU affinity* factor to the *One-to-One* level, and the *stress* factor to *CPU*. 59
3.11 Distribution of maximum execution times for the entire execution (replica + voter) by fixing *host-level scheduler* factor to *null* level, the *CPU affinity* factor to the *One-to-One* level, and varying all levels within *stress* factor. 60
3.12 The elements that the proposed Mixed Criticality System (MCS) deployment model handles. 64
3.13 A sample architectural scenario. 67

4.1 A block diagram illustrates the Omnivisor model, showcasing varied temporal and spatial isolation mechanisms across CPU clusters, emphasizing their heterogeneity. The arrows indicate the flow of a request from an initiator to the accessed resource (memory or I/O). 72
4.2 Omnivisor feature set (left) and remote core Virtual Machine (VM) startup process (right). 75

4.3 Architectural view of VM compiling and start procedures using the Omnivisor implementation on top of Jailhouse and the Zynq Ultrascale+ board. 81

4.4 RPUGuard Architectural Design 86

4.5 Remote Processor Messaging (RPMsg) layers 89

4.6 Table-based LPVM. The virtual address is checked against all entries, and permissions are computed thanks to a checker module. The relocation function is then applied, and the correct entry is selected. If no region is found, or a permission mismatch occurs, a fault is raised. 94

4.7 GRT-based LPVM flow diagram describing the translation process starting from a user mode access to the physical address generation. 95

4.8 Guarded Radix Tries (GRT)-based lightweight and predictable virtual memory (LPVM). The virtual address is used by an FSM to extract the current prefix and suffix. According to the current Segment Trie Table (STT) entry, checks are performed on the control word and guard word and a new request to the RPM is submitted. If no fault occurs during Segment Trie (ST) exploration, the physical address is retrieved from the Limit Table (LT). 97

4.9 Benchmarks normalized execution time. For each benchmark, the GRT-based LPVM overhead is shown compared to the baseline and the table-based LPVM overhead. All the values are normalized to the baseline to provide a fair comparison between the benchmarks. 101

4.10 Image building workflow. 107

4.11 Linux container stack (on the left) vs. runPHI images stack (on the right). 109

4.12 High-level architectural description of runPHI, highlighting the interactions of the configuration generator and partition management. 110

4.13 Example of combined $WorstCaseResponseTime$(WCRT). The fault-free execution has a long tail due to interference caused by complex hardware. F1 and F2 are two different faults that cause a fail-slow behavior. For example, F1 is a fault affecting the task itself, while F2 is a fault affecting a higher priority task affecting the preemption time. A single task re-execution may be accounted for in the $WorstCaseExecutionTime$(WCET), assuming a single transient fault. With a single WCRT we can account for all of this weighted by its occurrence probability. 113

5.1 Simplified scheme of a tokamak fusion device. 116
5.2 Normal behavior of the plasma current Ip and vertical position of the plasma centroid Zc. 120
5.3 Anomalous behavior of the plasma current Ip and vertical position of the plasma centroid Zc. 120
5.4 The AS_2 architectural scenario. 125
5.5 Density function plots of execution times of tasks C (a), (c) and DS (b), (d), per scenario. 127
5.6 Violin plots of execution times of tasks C (a) and DS (b), per scenario. 128
5.7 Comparison of boot times across heterogeneous processors in the ZCU Platform 131
5.8 Expected fault outcomes (top) and experimental configuration (bottom) of VMs running on remote CPU (rCPU) subjected to interference from various sources (APU, RPU-1, FPGA): a comparative analysis between traditional Hypervisor and Omnivisor. 133
5.9 Execution time slowdown of simple periodic task running in a VM over both Real-Time Processing Unit (RPU)-0 and RISC-V soft-core. The behavior of the applications under different sources of interference is shown first when using a plain jailhouse, then a partial Omnivisor implementation only with spatial isolation mechanisms enabled, and finally the full Omnivisor implementation. 135

5.10 Comparative evaluation of TACLeBench: The bar plots depict the execution time slowdown with and without temporal constraints. Each benchmark showcases the bandwidth limitation imposed on other managers to achieve the desired 20% maximum degradation on the VM. 137
5.11 Hardware-in-the-loop Simulation Setup 139
5.12 Input and Output of the simulated plasma model with periodic injected tension gain. In this scenario, the controller is able to stabilize the plasma. 140
5.13 Execution time of MARTe2 Controller running on a Linux Vanilla without any co-located workload. 140
5.14 Execution time of MARTe2 Controller running on a Linux PREEMPT-RT and isolated on a single cpu without any co-located workload. 141
5.15 Execution time of MARTe2 Controller running on a Linux PREEMPT-RT VM on Jailhouse hypervisor and isolated on a single cpu. The VM runs with membomb co-located workloads on other VMs on Application Processing Unit (APU)s, RPUs, and Field Programmable Gate Array (FPGA). 142
5.16 Input and Output of the simulated plasma model with periodic injected tension gain. In this scenario, the controller is not able to stabilize the plasma in time due to the deadline misses. 142
5.17 Execution time of MARTe2 Controller running on a Linux PREEMPT-RT VM and isolated on a single cpu on Omnivisor. The VM runs with membomb co-located workloads on other VMs on APUs, RPUs, and FPGA. The Omnivisor employs cache protection through cache-coloring but it doesn't provide bandwidth regulation. 143
5.18 Execution time of MARTe2 Controller running on a Linux PREEMPT-RT VM and isolated on a single cpu on Omnivisor. The VM runs with membomb co-located workloads on other VMs on APUs, RPUs, and FPGA. The Omnivisor employs cache and memory bandwidth protection through cache-coloring and mempol regulation 144
5.19 Transfers success ratio 146

5.20 Histograms of APU-RPU round-trip latency without regulation, as a function of the disturbance bandwidth 148
5.21 Histograms of APU-RPU round-trip latency with RPU-Guard, as a function of the disturbance bandwidth 149
5.22 Cumulative distribution function of the boot times of the solutions compared. The solutions show comparable boot times. 151
5.23 Cumulative distribution function of the execution times in varied stress conditions. Partitioned containers show constant times. 152

List of Tables

3.1 Temporal isolation properties across safety-related standards. 41

3.2 Experimental factors and levels 45

3.3 Means (*M*) and standard deviations (*SD*) percentage increase, comparing *One-to-One* level to *Many-to-Many* level. SD best case and worst case are highlighted in green and red, respectively. 61

3.4 Means (*M*) and standard deviations (*SD*) percentage increase, comparing *No Stress* level to *CPU Stress* level. SD best case and worst case are highlighted in green and red, respectively. 61

3.5 Means (*M*) and standard deviations (*SD*) percentage increase, comparing *No Stress* level by fixing the *host-level scheduler* factor to *null* scheduler. SD best case and worst case are highlighted in green and red, respectively. 62

3.6 Local and remote VMs, Central Processing Unit (CPU) pools, *DM*s, and scheduling schemes of the sample architectural scenario. 68

3.7 The deployment quadruplet of the sample architectural scenario. 68

4.1 Area, Power, and Frequency results for cv32e41s-based soc baseline, with different PMP and LPVM-TAB configurations and LPVM-GRT as well. Data refer to the Nexys A7-50T board. 98

5.1 Local and remote VMs, CPU pools, DMs, and scheduling schemes of the designed architectural scenario. 123
5.2 The deployment quadruplets of the designed architectural scenarios. 123
5.3 Metrics per scenario related to tasks C and DS (N/A: Not Applicable). 126

1 Introduction

COMPLEXITY and computational power of computing systems have seen extraordinary growth over the last decades, primarily driven by the rapid digitization of human activities. From managing global communication networks to supporting advanced medical diagnostics and industrial automation, modern systems handle vast amounts of data and complex operations. In less critical environments, such as consumer electronics or smart devices, these systems handle tasks like multimedia processing, resource management, and scheduling, focusing primarily on delivering a seamless user experience and high-quality of service. On the other hand, in highly critical fields such as nuclear fusion research, these systems are pushed even further. Controlling and stabilizing plasma, processing sensor data in real-time, and running predictive simulations to optimize reactor performance require cutting-edge computational capabilities and advanced architectures to meet the stringent demands of precision and reliability in this field.

In the early stages of computing, advances in processing power were largely driven by miniaturizing hardware components and increasing processor frequencies, as outlined by Moore's law [2]. This law observed that the number of transistors on integrated circuits doubled approximately

every two years, leading to faster processing speeds and lower computing costs, as shown in Fig 1.1.

Figure 1.1: Years in which the microchip was first introduced (Licensed under CC-BY by the authors Hannah Ritchie and Max Roser)

However, as we near physical and thermal boundaries, sustaining this trend has become increasingly challenging [3]. With transistors approaching atomic dimensions and fabrication costs escalating, the traditional technological force that has driven Moore's Law for the past 50 years is failing. Consequently, to continue improving the performance and efficiency of new platforms, hardware manufacturers have shifted towards integrating multiple specialized Processing Element (PE)s tailored for specific tasks on a single chip as shown in Figure 1.2. This trend resulted in the development of heterogeneous Multi-Processor Systems on Chip (MPSoC)s [1, 4].

Figure 1.3 illustrates a simplified architecture of a modern heterogeneous MPSoC. These platforms typically combine traditional CPUs with various specialized PEs tailored for specific tasks. These may include Graphical Processing Unit (GPU)s for graphics processing, RPUs for real-time tasks, Neural Processing Unit (NPU)s for AI and neural net-

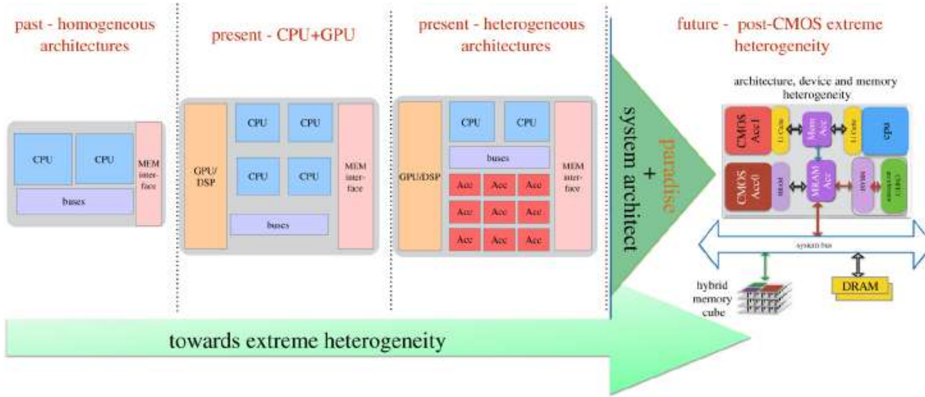


Figure 1.2: From homogeneous to extreme heterogeneous architectures (reproduced from [1]).

work computations, FPGAs for customizable, re-programmable hardware, and many others.

In safety-critical Cyber-physical Systems (CPS)s such as drones, medical devices, automotive systems, and fusion reactors, hardware is typically embedded within physical components, forming what are known as *embedded systems*. These are dedicated computing units designed to perform specific tasks within larger mechanical or electrical systems, tightly inte-

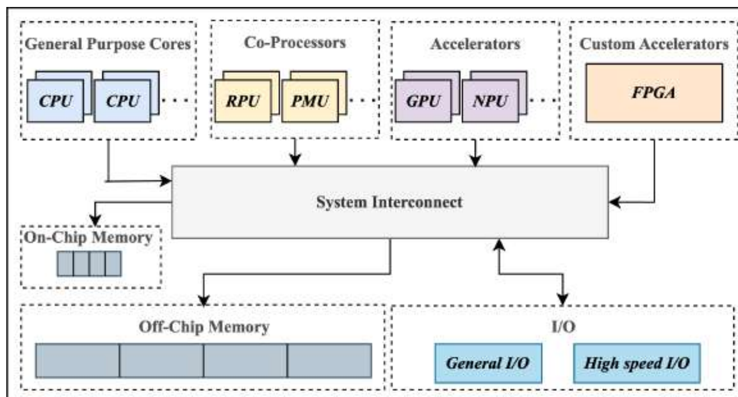


Figure 1.3: Example of Heterogeneous MPSoCs

grated into the physical processes they monitor and control. This makes them essential for real-time decision-making and operations.

Embedded systems are often constrained by factors such as power, space, and processing capabilities. However, with recent advances in hardware design, even highly complex MPSoCs can now be considered embedded systems. Despite their significant computational power and ability to run sophisticated safety-critical applications, managing and optimizing these systems remains complex due to the specialized and diverse nature of the processors involved.

While MPSoC architectures undeniably enhance application performance, achieving the necessary level of predictability to ensure safe and reliable operations is particularly challenging in CPSs where applications run at very high frequencies (e.g. tens to hundreds of kHz), using complex hardware accelerators and external devices.

Real-Time Control in Nuclear Fusion Reactors. Nuclear fusion reactors exemplify the high level of complexity that modern CPSs can achieve, as they seek to replicate the processes powering the sun, promising an almost limitless and clean energy source. A key initiative driving the development of these reactors is the ITER tokamak, which is an experimental thermonuclear reactor that plays a central role in advancing fusion technology and integrating complex control systems. Once its construction will be completed, ITER will be the world's largest fusion experiment, aiming at being the first reactor to produce a net-positive energy output through a fusion reaction.

Controlling the plasma within a fusion reactor requires extremely fast computations and strict adherence to hard real-time requirements to maintain system stability. Due to the high computational demands, modern hardware architectures are essential to meet the performance demand [5, 6, 7]. MPSoC platforms are already employed in ITER for real-time control and signal conditioning, as discussed in recent studies [8, 9].

Recent MPSoC possess sufficient single-core computational power to handle model-based control problems, such as Vertical Stabilization [10]. However, advanced methods like reinforcement learning (RL) are being explored to improve control efficiency and address complex control strategies [11]. These methods leverage hardware accelerators, which can be

seamlessly integrated into modern MPSoCs. RL-based controllers, however, introduce time unpredictability to the control due to their stochastic nature. Consequently, maintaining a traditional model-based controller as a fallback on the same platform can enhance system safety by providing a reliable backup in case of RL failure.

However, ensuring multiple applications that share resources on a single platform meet real-time constraints is still a nontrivial challenge, and *overprovisioning* is currently used as a common strategy to address the complexity of the system; each application is deployed on a separate platform to avoid interference and simplify analysis and certification. While this reduces risk, it results in three principal shortcomings:

- Under-utilization of hardware: Many PEs on the heterogeneous platforms are either idle or lightly used.
- Lack of scalability: adding new applications to the system often requires new hardware, making the overprovisioning approach inefficient and not scalable in the long term.
- Unreliable communication: when applications are spread across different boards, managing fast, reliable communication becomes more difficult, especially since many applications rely on the same sensor data

By contrast, integrating multiple applications within a single System on Chip (SoC) allows for higher resource utilization and lower-latency communication, making it a more efficient solution for real-time control in nuclear fusion systems[9].

1.1 Mixed-Criticality Systems

MCS refers to a system that integrates multiple applications with different levels of importance or criticality on the same hardware platform. These applications range from highly critical, where timing and performance guarantees are essential for safety, to non-critical tasks, where flexibility and resource efficiency are prioritized. The core challenge of MCSs lies in managing these varying levels of criticality while ensuring that the most

critical applications consistently meet their real-time requirements, even in the presence of resource contention or system degradation.

In recent years, there has been a significant shift in the industry's approach towards system consolidation and integration in the context of CPSs; combining multiple applications, previously hosted on separate hardware platforms, onto a single, more powerful platform to reduce Size, Weight, Power, and Cost (SWaP-C) of the hardware. This approach not only optimizes hardware SWaP-C but also improves efficiency and communication between applications which is of fundamental importance in high-frequency control systems. Globally, this trend has spurred significant investments in research and development, attracting both industrial [12] and academic [13] interest. As a result, the concept of consolidation and MCS is now reflected in many industrial standards, including the *EN5012X* series for railway systems [14], *DO-178B* for avionics [15], and *ISO 26262* for automotive applications [16].

Mixed-Criticality Systems Challenges. Consolidating applications with differing criticality levels onto a single platform is not an easy task. The introduction of mixed-criticality models, inspired by the pioneering work of Vestal [17] (see Section 2.1) tries to address this challenge by incorporating execution criticality into traditional real-time scheduling frameworks. Introducing such models into theoretical frameworks has unveiled a new research landscape that spans from developing innovative software architectures and hardware technologies to enhancing real-time theoretical analysis techniques. It has opened avenues for exploring how best to allocate resources dynamically, enforce timing constraints, and mitigate interference among tasks [13]. However ensuring effective resource utilization and maintaining high-quality service for non-critical applications, while simultaneously providing real-time performance guarantees for critical applications, continues to be a significant challenge in real physical systems [18].

1.2 Nuclear Fusion Real-Time Control Systems

Figure 1.4 shows a simplified system overview of one of the ITER control systems: Internet of Things (IoT) actuators and diagnostics on the reactor

(things layer), which have stringent temporal and performance demands, are connected to a plant control system (fog layer) [19, 20]. Here, MPSoCs are strategically deployed near the reactor to minimize communication latency between the controllers and the sensors while maintaining high computational performance. In this fog layer, diverse control loops and signal conditioning algorithms, each with distinct sampling times and reliability requirements, must coexist harmoniously. Data and signals processed in this layer are transmitted via a Synchronous Databus Network (SDN) to the cloud layer, where more computationally intensive control algorithms are executed. In the cloud layer, data from the fog layer are stored and further analyzed using sophisticated software, including AI algorithms and statistical tools, to derive valuable insights from the experiments.

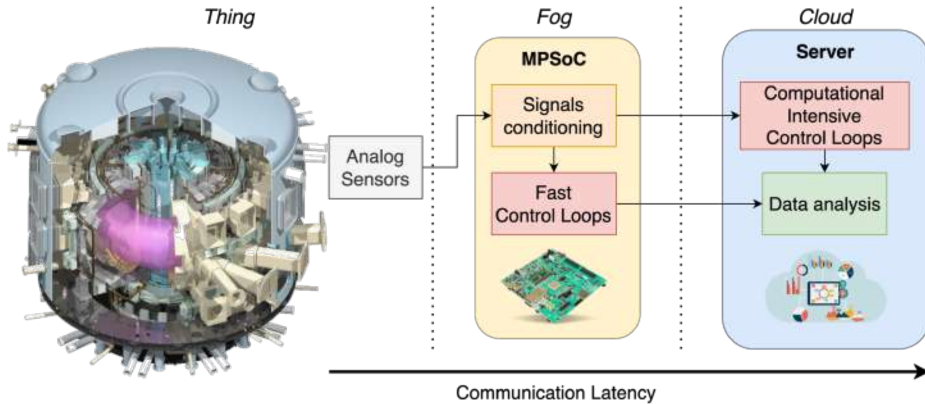


Figure 1.4: ITER System Overview

Not all control algorithms within the ITER control system can be deployed in the fog layer due to their dependence on high-end computing power which can be achieved only by leveraging powerful servers. Nonetheless, there has been significant progress in porting real-time nuclear fusion frameworks such as Multi-threaded Application Real-Time executor 2 (MARTe2), which has been used in Joint European Torus (JET) fusion reactor, to embedded systems such as the ARMv8-based MPSoCs used for ITER's magnetic diagnostics [21]. This is because high-frequency, lightweight control loops and signal conditioning algorithms can strongly benefit from implementation on MPSoCs situated close to the machine [9,

21]. First of all, the proximity between controllers and sensors guarantees low transmission latency and real-time communication between applications. Additionally, adopting a decentralized fog computing paradigm mitigates also network partitioning issues prevalent in cloud-based solutions [22] and can reduce the costs of the utilized hardware.

Challenges in Flexibility for Fusion Control Systems. While it is feasible to integrate applications onto a single board via custom implementations achieving good performance [9], this approach lacks flexibility; adding or replacing applications in such systems is costly since significantly complicates system reconfiguration. To enforce extensibility and flexibility, a more standardized approach is necessary. As a result, Real-time virtualization and partitioning emerge as promising solutions to achieve such requirements (see Section 2.3).

1.3 The Rise of Real-Time Virtualization

Among the various approaches proposed to deploy MCSs [23, 24, 25], real-time virtualization stands out as particularly promising for high-performance real-time systems [26, 27].

Virtualization technology has revolutionized cloud computing by offering the capability of abstracting the hardware resources of a physical machine into two or more VMs (see Figure 1.5). These VMs run isolated

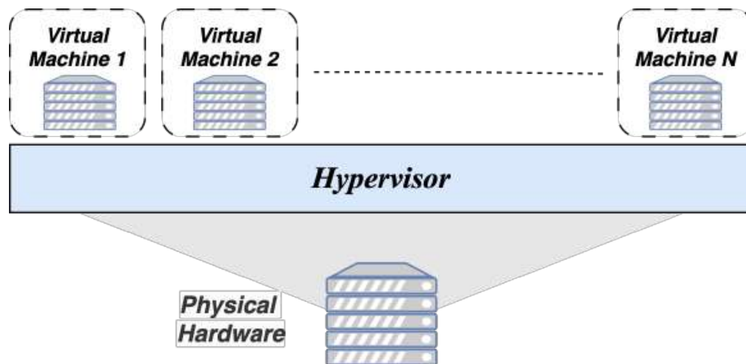


Figure 1.5: Example of Virtualization

software, each behaving as though it operates independently within the system, oblivious to the existence of the others. The most compelling feature of virtualization in the cloud context is its flexibility in resource allocation. Resources can be dynamically assigned to VMs based on their current needs, maximizing the utilization of physical machines and thereby reducing operational costs while supporting more applications.

However, for CPSs with stringent real-time performance requirements, throughput becomes secondary to predictability. The key advantage of virtualization for these systems lies in its ability to isolate resources, facilitating the consolidation of diverse applications, even those with different criticality levels, onto a single platform. Traditional virtualization, however, was not originally designed with predictability in mind, necessitating a redesign of virtualization software to host real-time applications effectively.

Over the past decade, several methodologies have emerged to enable real-time virtualization [27]. These range from integrating real-time schedulers into established cloud hypervisors such as KVM and Xen [28] to developing ad-hoc hypervisors with minimal footprints, like Jailhouse [29], Bao [30], and others. In Section 3, we systematically evaluate the temporal isolation provided by these kinds of solutions, specifically Xen, using a Domain of Experiment (DoE) approach. Our analysis reveals that while feature-rich and flexible, dynamic approaches may lead to unpredictable behaviors that complicate validation and certification processes. In contrast, partitioning-based approaches, despite being less feature-rich and flexible, offer superior isolation and are more conducive to meeting certification requirements. This makes partitioning hypervisors an ideal starting point for the Omnivisor model proposed in this dissertation (see Section 4) whose goal is to take control over heterogeneous MPSoCs with asymmetric processors to run isolated mixed-criticality applications.

Challenges and Opportunities in Virtualizing MPSoCs. Heterogeneous MPSoCs are increasingly recognized as a suitable architecture for complex CPSs due to their benefits in cost efficiency, area savings, power consumption, and performance. However, integrating MCSs into such architectures through virtualization presents significant challenges while also opening up new opportunities.

A primary challenge is adapting traditional virtualization models, which were designed for symmetric systems and do not account for the complexities of heterogeneous MPSoCs. In conventional setups, the hypervisor, which is the software that administers the VMs, manages primary processors but often ignores or inadequately handles co-processors, treating them as mere I/O devices assigned to VMs on the primary cores. This oversight allows VMs controlling secondary cores to load and execute code that can potentially access critical platform resources. Consequently, the co-processors, with their elevated privileges, pose a risk to the spatial and temporal isolation of other VMs, undermining system security and stability. This issue will be explored further in Section 4.

In contrast, an effective hypervisor for heterogeneous MPSoCs must comprehensively manage both primary and secondary processing elements, ensuring that each core, regardless of type, is integrated into the virtualization framework with appropriate controls and isolation measures. This approach mitigates the risks associated with uncontrolled access to platform resources and enhances the overall reliability and security of the system.

Additionally, shared resources in MPSoCs, such as interconnects, on-chip caches, and DRAM, introduce timing interference. Operations on one core can affect the timing behavior of others, significantly distorting WCET estimates—potentially increasing them by up to 300% [31]. Existing virtualization strategies primarily focus on symmetric multi-core architectures and fail to adequately address the timing interference issues unique to heterogeneous MPSoCs, leaving a critical gap in current methodologies.

Despite these challenges, heterogeneous MPSoCs present unique opportunities for optimizing MCS integration. Their architecture supports advanced levels of optimization, allowing applications to be deployed on cores specialized for specific tasks while strategically managing shared resources. For instance, resource allocation can be dynamically adjusted based on the needs of MCS, such as deciding whether the RPUs should share the On Chip Memory (OCM) with the CPU and/or should use a specific quantity of memory bandwidth depending on task requirements. This flexibility enables tailored solutions that strike a balance between isolation and performance optimization.

1.4 Objective and Scope of the Dissertation

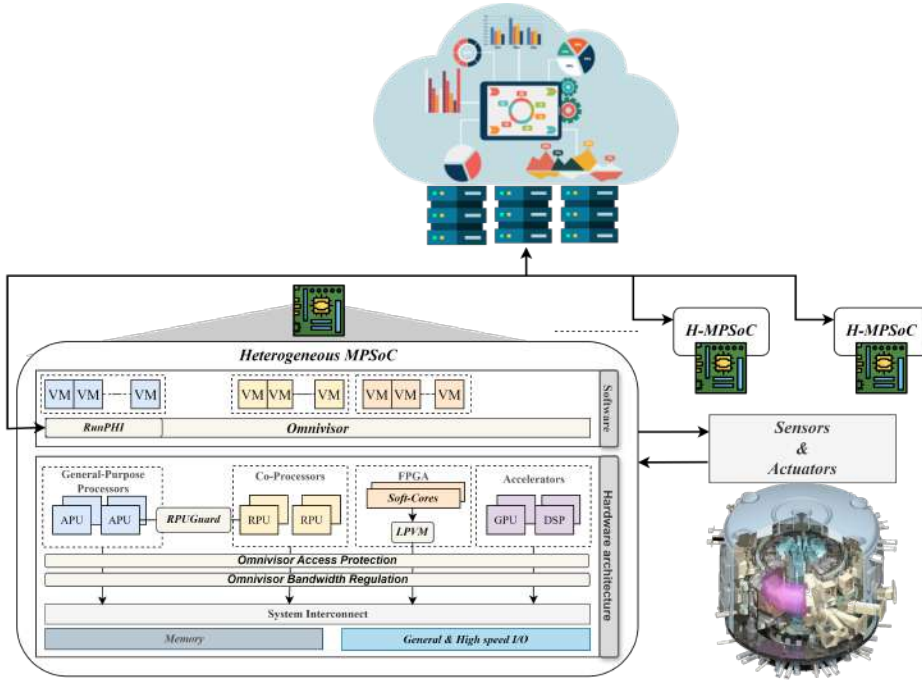


Figure 1.6: Overview of the Proposed Architecture based on the Omnivisor model

Given the increasing complexity of hardware and the stringent real-time requirements of modern CPSs, this dissertation advocates for a transition toward advanced, standardized strategies that consolidate MCS on modern platforms, as shown in Figure. 1.6. Central to this work is the introduction of the Omnivisor model, a groundbreaking approach leveraging real-time virtualization and partitioning to maximize the potential of heterogeneous computing platforms. By exploiting asymmetric cores and internal hardware protection mechanisms, the Omnivisor model ensures resource utilization, predictable performance, and high levels of safety and reliability. Starting from the Omnivisor model, the dissertation proposes RPUGuard as an innovative real-time communication framework

between asymmetric processors. Therefore, it explores LPVM, a new technology for predictable memory virtualization on microcontrollers. It also demonstrates the convenience of having Omnivisor-based virtualization integrated into orchestration frameworks such as Kubernetes by proposing the RunPHI runtime. Subsequently, this work investigates the usability of the proposed solutions for the deployment of complex, next-generation safety-critical systems for nuclear fusion. Thus, the following contributions are presented:

- A systematic Design of Experiments (DoE) approach for assessing the temporal isolation provided by hypervisors. This contribution outlines strategies to optimize hypervisor configurations and minimize inter-VM interference. The findings highlight partitioning as the most effective virtualization method for isolation, though challenges remain due to the complexity of modern architectures.

- A standardized deployment model for MCSs on distributed systems utilizing virtualized MPSoCs and servers within a Cloud-to-Thing continuum, designed to simplify integration and enhance scalability in complex infrastructures like the ITER magnetic control system.

- Omnivisor, a novel virtualization model that extends real-time static partitioning hypervisor's model to manage asymmetric cores (e.g., ARM64, ARM32, and RISC-V) and control the memory bandwidth of accelerators. This model enhances the dependability of mixed-criticality systems on modern MPSoC platforms while simplifying their utilization.

- RPUGuard, a communication framework designed for virtualized asymmetric processors. It provides fine-grained control over inter-processor communication in a virtualized configuration, reducing interference and ensuring real-time performance.

- LPVM, A lightweight and predictable hardware component design to virtualize microcontroller-level cores, such as RPUs, in future architectures, enabling higher resource utilization while maintaining real-time performance through virtualization.

- RunPHI, a runtime system that integrates the Omnivisor into an orchestration framework for automated configuration, management, and coordination of computer systems, apps, and services. This system improves flexibility and usability for applications running on heterogeneous platforms also with asymmetric cores.

- A comprehensive evaluation of the proposed framework and methodologies, applied within the context of the ITER project. Specifically, the Vertical Stabilization control algorithm was tested in a Hardware-In-the-Loop (HIL) setup with an emulated plasma model. This evaluation explored various system configurations on heterogeneous MPSoCs, using a range of stressors to demonstrate the robustness and adaptability of the proposed approach.

Despite real-time virtualization being an interesting solution to achieve the requirements of nuclear fusion reactors, it is still a challenge to achieve high utilization of hardware and effective protection and isolation between applications on the heterogeneous MPSoCs used in ITER. This dissertation addresses these challenges by demonstrating three key use cases enabled by this technology. First, (**UC1**) it allows real-time monitoring applications to be deployed alongside safety controllers without compromising predictability. Second, (**UC2**) it supports the deployment of multiple controller versions on the same board, facilitating both testing and redundancy. Finally, virtualization simplifies the deployment of applications across distributed systems (**UC3**).

To provide context and background, Section 2 explains the basic knowledge needed to fully understand the entire dissertation. It offers a concise discussion of related work, highlighting foundational studies and contemporary advancements on the consolidation of MCSs through virtualization. In Section 3, we address the critical issue of temporal isolation assessment and MCS modeling. We introduce a systematic DoE approach for configuring virtualized systems, focusing on minimizing interference and enhancing performance. The section uses the findings presented in the publications A.3 and A.5. Following a demonstration of the capabilities of modern partitioning hypervisors, Section 4 presents an extended model that adapts traditional hypervisor frameworks to accommodate the heterogeneity of Commercial Off-The-Shelf (COTS) architectures while en-

suring robust isolation for real-time applications. The model is based on the publications A.2, A.4, A.9, A.10, and A.11. In Section 5, we evaluate the proposed technologies and methodologies in Nuclear Fusion scenarios, demonstrating their applicability and flexibility with the aforementioned three use cases. The dissertation concludes with a discussion of findings and potential directions for future research in Section 6.

2
Background and Related Work

THIS CHAPTER introduces key concepts necessary to understand the approaches proposed in this thesis. As a starting point, we want to clarify what real-time systems are and why safety-critical applications need real-time guarantees. Therefore, we clarify how real-time applications for Nuclear Fusion Reactors are developed using real-time frameworks. Once the basic concepts of real-time systems are clear, we describe how the mixed-criticality system model was born on top of traditional real-time models to face the challenge of running tasks with different levels of criticality on the same physical platform. Despite the analytical models developed for MCS being well known and accepted by the community, going from an analytical description to a practical working environment is not straightforward. Therefore, we continue the chapter by looking at the practical development of MCSs over COTS MPSoCs, explaining the practical challenges of the consolidation of multiple applications and demonstrating the distance between the analytical world and the real world. Therefore, we cover virtualization technology as one of the possible practical solutions for the deployment of MCSs. Specifically,

we focus on how embedded virtualization works and why partitioning hypervisors are used today to keep systems isolated and secure. Finally, we explore the limitation of existing works that have been tried to address the MCSs deployment problems on heterogeneous MPSoCs, explaining how we can go beyond with the proposed Omnivisor model.

2.1 Real-Time Systems and Mixed-Criticality

Real-time systems are computing systems that must respond to events within a strict time frame. Unlike general-purpose systems where performance is measured by average response time, real-time systems are judged based on their ability to meet specific and bounded timing constraints. These systems must ensure that all tasks are completed within their deadlines under all specified conditions.

Real-time systems are usually modeled as a recurrent task model that allows a priori validation of temporal constraints of the tasks. For this reason, these systems are integral in applications where timely and predictable responses are critical, such as in automotive control, industrial automation, medical devices, telecommunications, and nuclear reactors.

We can categorize two types of real-time systems according to the consequences of missing a deadline:

- *Hard Real-Time Systems*: Where missing a deadline is considered a system failure and can lead to catastrophic consequences (e.g., flight control systems).

- *Soft Real-Time Systems*: Where occasional deadline misses are tolerable but can degrade system performance (e.g., video streaming).

2.1.1 Real-Time Task Model

In real-time systems, tasks are typically modeled with a set of parameters that define their execution behavior. This model is fundamental to understanding how real-time scheduling works and how tasks are managed. Under the well-known *sporadic task model* [32] each task T_i is repeatedly invoked by asynchronous, external events such as device interrupts or expiring timers. A real-time workload consists of a set of n sequential

tasks $\tau = \{T_1, ..., T_n\}$ and each task is characterized by the following key parameters:

- *Release Time (r_i)*: The time at which a task becomes ready for execution.
- *Start Time (s_i)*: This is the point in time when the task begins its execution.
- *Execution Time (C_i)*: The amount of time required by a task to complete its execution. In real-time literature, this is often represented as the WCET, which is the maximum time a task might take to execute under worst-case conditions.
- *Finish Time (f_i)*: This is the point in time when the task completes its execution.
- *Deadline (D_i)*: The time by which a task must complete its execution. The task is considered late if it is completed after its deadline.
- *Period (T_i)*: For periodic tasks, this is the time between consecutive releases of the task. It defines how frequently the task is executed.
- *Priority (P_i)*: The importance level assigned to a task, which can influence its scheduling order.

The following figure illustrates a typical periodic task with its parameters:

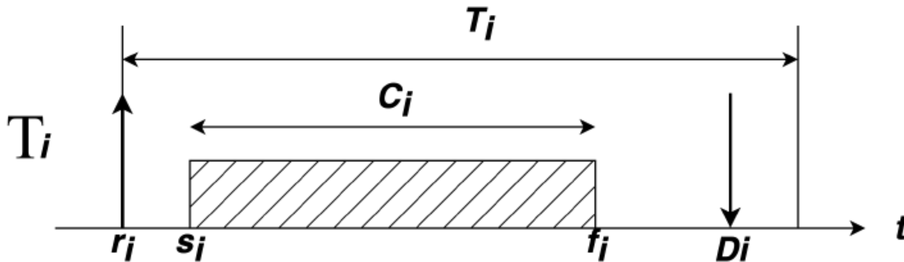


Figure 2.1: Example of Periodic Task

2.1.2 Real-Time Frameworks for Nuclear Fusion

A real system such as the one for controlling Nuclear Fusion reactions is composed of several distributed and high-frequency real-time tasks, and developing and deploying such tasks can be nontrivial: As an example, in real-time digital feedback systems for Nuclear Fusion [33], a large number of signals must be processed, particularly for plasma parameters like position and shape. Therefore the control cycle time varies based on the phenomenon and actuator response, ranging from 50 μs to a few milliseconds. Due to the complexity and number of signals, multiple computing units are often used, communicating via networks to manage and process the data.

Furthermore, the development of real-time applications is usually associated with non-portable code targeted at specific real-time operating systems. The boundary between hardware drivers, system services and user code is commonly not well-defined, making the development of the target host significantly difficult. As a solution to ease the programming of such tasks and make them portable, the Nuclear Fusion community started to use Real-Time frameworks.

One of the most used is MARTe2 framework [21, 34, 7]: MARTe2 is a versatile real-time C++ framework designed primarily for developing and deploying control systems, though it can be used for other applications as well. It is built on a modular, layer-based architecture that enhances code safety and simplifies debugging. Key features include an efficient logging system, built-in object introspection, automatic memory management, and flexible, data-driven configuration.

One of MARTe2's strengths is its generic application module, which clearly separates hardware interfaces, algorithms, and system configuration. This modularity allows for easy reuse of similar systems by swapping out modules, while keeping the hardware interface consistent.

MARTe2 has been successfully used in critical applications, such as the vertical stabilization of the JET tokamak, where it achieved precise control with minimal jitter. It also serves as the real-time control engine for the COMPASS tokamak's plasma control and ISTTOK's tomographic reconstruction, demonstrating its reliability and effectiveness in demanding environments.

For the ITER project instead, multiple frameworks will be used to im-

prove the diversity. Besides MARTe2 the Real-Time Framework (RTF) will be used [35]. RTF is a software suite developed by the ITER Control Data Access and Communication (CODAC) division to facilitate the implementation of real-time applications. Similarly to MARTe2, it is a middleware primarily designed for the Plasma Control System (PCS) but can also be used in other systems requiring real-time control or data processing due to its universal architecture. RTF simplifies the development of real-time systems by implementing and hiding many mechanisms typical for such systems, allowing developers to focus on the functional part of the application. It is highly modular, portable, and independent of hardware configuration.

2.1.3 Mixed-Criticality Task Model

As real-time systems evolve to handle more complex and diverse applications, the need to manage tasks of varying criticality levels (i.e., the designation of the level of assurance against failure [18]) on the same hardware platform has become evident. At the same time, these platforms are migrating from single cores to multi-cores and in the future heterogeneous many-core architectures. This led to the development of the mixed-criticality task model, significantly influenced by the work of Vestal [17]. This model extends the traditional real-time task model to accommodate tasks with different criticality levels, each with distinct timing and assurance requirements.

In traditional real-time systems, all tasks are treated equally in terms of their criticality, and the primary objective is to ensure that each task meets its deadline. In contrast, mixed-criticality systems recognize that not all tasks are equally critical. For example, in an automotive control system, tasks controlling the braking system are more critical than those managing the infotainment system. The Vestal model [17] introduces the notion of multiple criticality levels and allows tasks to have different execution requirements based on these levels. Here's how it modifies the traditional task model:

- *Criticality Level (L)*: Each task is assigned a criticality level L which indicates its importance. For instance, a task might be classified as Low (L) or High (H) criticality.

- *Multiple WCET Estimates*: Tasks have different WCET estimates depending on their criticality level. For example, a task τ_i might have $C_i(L)$ for Low and $C_i(H)$ for High criticality. These estimates reflect the increasing conservatism in the analysis as the criticality level rises.

The model also introduces the concept of mode switching based on the system's observed behavior:

- *Normal Mode*: The system operates under normal conditions, and tasks are scheduled based on their lower criticality WCET estimates.

- *Degraded Mode*: When a high-criticality task exceeds its lower criticality WCET, the system switches to a higher-criticality mode. In this mode, only the most critical tasks are guaranteed to meet their deadlines, while less critical tasks may be discarded or delayed.

This criticality-aware scheduling enables the system to achieve enhanced resource utilization under normal conditions and ensures that the essential functions of high-criticality tasks continue to operate effectively in the face of timing anomalies or system stress. The flexibility afforded by this model supports scalability, allowing new tasks with varying criticality levels to be integrated seamlessly without requiring fundamental changes to the scheduling framework.

However, the benefits of the Vestal model are accompanied by significant challenges. The complexity of scheduling increases substantially as it must account for the varying WCET estimates and the potential for mode switches, adding layers of decision-making that are absent in traditional real-time systems. Verification and validation processes become more demanding, as the system must be analyzed and tested across multiple modes of operation to ensure that all timing requirements are met consistently, particularly for high-criticality applications where safety and reliability need to be guaranteed. Additionally, managing the transitions between normal and high-criticality modes presents its own set of challenges. These transitions must be handled carefully to avoid system instability and ensure that critical tasks continue to meet their deadlines without introducing undue disruption to the less critical tasks.

2.1.4 The Evolution of Mixed-criticality Systems model

The Vestal model, which fundamentally shifted the approach to managing mixed-criticality tasks by incorporating multiple WCET estimates and criticality-aware scheduling, has continued to evolve through extensive research. Recent advancements have further refined the theoretical underpinnings and practical applications of this model, addressing some of its inherent challenges and expanding its capabilities. The core research question driving these advancements is how to systematically balance the conflicting needs of partitioning to ensure safety and sharing to optimize resource use. This challenge presents theoretical issues in modeling and verification, as well as practical concerns in designing and implementing the required hardware and software runtime controls. From the theoretical point of view, some important advancements have been introduced during the years:

- *Expansion to Multiple Criticality Levels*: Initially designed for dual-level criticality, the model now supports multiple levels, accommodating diverse operational requirements in cyber-physical systems [36, 37]. This extension improves scheduling flexibility and system scalability.
- *Advanced Scheduling Algorithms*: New algorithms have been developed to minimize overhead during mode transitions and optimize resource allocation dynamically [38, 39]. These algorithms balance efficiency and assurance, ensuring high-critical tasks receive necessary resources without excessive system disruption.
- *Probabilistic Analysis Integration*: Incorporating probabilistic WCET estimates into the framework allows for better handling of timing uncertainties [40, 41], leading to more robust scheduling. Statistical methods provide a nuanced understanding of task behavior under varying conditions.
- *Hierarchical Scheduling Frameworks*: The model introduces hierarchical scheduling mechanisms, where different levels of criticality are managed through nested schedulers [42, 43]. These frameworks, are perfectly mapped onto virtualized systems as detailed in Section 2.3.

However, as the mixed-criticality model evolves, practical issues arise regarding its application on real hardware platforms. Research has increasingly focused on implementing these systems on COTS heterogeneous multi-core platforms, emphasizing the critical role of hardware-software co-design to meet stringent timing and safety requirements [44]. The shared resources and inherent heterogeneity of these systems complicate the validation of the model's effectiveness. Since heterogeneous MPSoCs are a central topic in this work, we delve into how these architectures work, explaining the advantages and challenges in executing MCS on these platforms.

2.2 Multi-Processors Systems-on-Chip

The initial computing platforms were single-core systems, characterized by a single processing unit that performed all computing tasks. All the basic real-time systems theorems have been developed with these kinds of platforms in mind. However, these simple mono-core systems have met their limits in terms of processor frequency. Nowadays single-core processors are considered limited in performance and scalability. As applications grew more demanding, the industry shifted towards multi-core architectures to enhance performance without increasing clock speeds, which were constrained by power and thermal limitations. Multi-core SoCs (MPSoCs) integrate multiple processing cores on a single chip, allowing for parallel execution of tasks. This design improved overall performance and energy efficiency, making it possible to handle more complex applications. Each core in an MPSoC typically runs its own thread or process independently, enabling better utilization of the chip's resources.

One example is the AMD Zynq™ UltraScale+™ (ZCU) MPSoC board, which is used in the ITER project. Due to its relevance in this context, we selected it as the experimental platform for the evaluations in this dissertation. The board features a quad-core ARM® Cortex™-A53, a dual-core 32-bit ARM® Cortex™-R5, an ARM® Mali™-400 MP2 GPU, and a 16nm FinFET + Programmable Logic (FPGA). Additionally, the platform is equipped with protection mechanisms for both temporal isolation (QoS), address translation (MMU, SMMU), and access permissions (SMPUs, and SPPUs) that are discussed in Section 2.2.2. From now

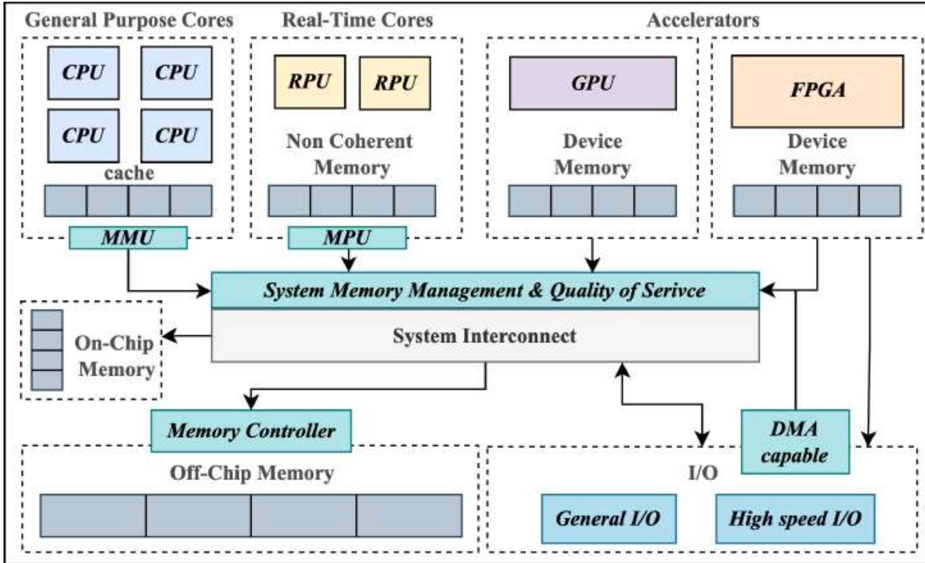


Figure 2.2: Simplified Schematic of Zynq Ultrascale+ MPSoC

on, we will refer to this platform with the *ZCU* notation. Moreover, to use the correct terminology, the SMPUs/ SPPUs on the board are named Xilinx Memory Protection Unit (XMPU) and Xilinx Peripheral Protection Unit (XPPU). Figure 2.2 shows a simplified architectural view of the described platform.

Despite their advantages, MPSoCs introduce significant complexity, particularly in managing shared resources. These shared components include caches, memory controllers, and interrupt systems, which not only play crucial roles in the overall functionality but also pose challenges to the predictability of the system.

2.2.1 Resources Contention in Modern Architectures

Shared Caches Interference. Modern multi-core architectures are usually equipped with per-core private cache, also known as level-1 cache, and shared cache which is usually referred to as level-2 (L2) cache, or level-3 cache (L3) when an intermediate cache level exists. In general, we can refer to the shared cache as last-level cache (LLC). These are used

to store data that multiple cores might need. While they improve efficiency by reducing memory access times, they also introduce variability in execution times due to cache contention. When multiple cores access the shared cache simultaneously, it can lead to unpredictable delays, complicating the estimation of the worst-case execution time (WCET). This is because the memory accesses of an application running on a core can cause the eviction of data belonging to a different application from the cache. This will eventually cause an unexpected cache miss to the second application which will suffer an increase in its execution time. In general, we can categorize cache interference into four primary types [45]:

1. *Intra-Task Interference (Self-Eviction)*: This type occurs when a task displaces its own data in the cache, leading to self-evictions.

2. *Inter-Task Interference*: This happens when one task causes the eviction of cache data belonging to another task running on the same CPU.

3. *Cache Pollution from Kernel Activities*: Asynchronous kernel activities, such as Interrupt Service Routine (ISR) and deferrable functions, can pollute the cache by displacing useful data.

4. *Inter-Core Interference*: This occurs when tasks running on different cores displace each other's data in a shared cache.

The first three types of interference can occur in both single-core and multi-core systems. However, the fourth type, inter-core interference, introduces a dependency between cores that complicates traditional schedulability analysis. Because of these various interference sources in multi-core systems, achieving deterministic behavior with traditional caches requires restrictive operational approaches. The literature proposes various cache management techniques implemented in software and hardware to address these scenarios, from cache locking [46, 47] and cache partitioning [45, 48] to more recent run-time cache analysis [49].

Memory Controller Contention. A SoC typically includes several managers (cores and accelerators) that access the system's main memory through a memory controller. To align with ARM's terminology, we will

use the term "*managers*" to refer to all hardware capable of initiating memory transactions. In heterogeneous SoCs, different managers often exhibit diverse memory access patterns, leading to contention at the memory controller. This contention can cause delays and reduce memory bandwidth available to individual managers, adversely affecting the predictability of task execution times.

In COTS platforms, the memory controller is often a closed-source component, limiting the ability to directly manage or modify its operation. This limitation necessitates the implementation of memory bandwidth reservation systems to regulate memory access. Over the years, various solutions have been proposed to address this issue, both in software and hardware.

Software-based approaches, such as [50, 51], focus on assigning specific memory bandwidth quotas to each manager to prevent over-utilization and minimize interference between managers. These solutions help maintain predictable memory access patterns, reducing unexpected latency increases in applications.

Hardware-based approaches, such as [52, 53], offer alternative strategies to manage memory access. These solutions aim to provide deterministic memory access by controlling the allocation and scheduling of memory bandwidth among different managers.

The common goal of these approaches is to allocate memory bandwidth efficiently and ensure that managers do not interfere with each other, thereby maintaining the predictability and performance of real-time applications. By implementing such memory bandwidth reservation systems, it becomes possible to mitigate the impact of contention at the memory controller and achieve more reliable task execution times in heterogeneous MPSoCs.

Interrupt Handling Impact. Handling interrupts in multi-core systems poses significant challenges for real-time performance. Interrupts are signals that demand immediate attention from the CPU, signaling incoming events and helping utilize CPU resources more efficiently by avoiding continuous polling. However, in a multi-core environment, deciding which core should handle an interrupt can impact system performance and predictability.

Interrupts serve to notify the CPU of events and to optimize resource use, but they also introduce latency and synchronization issues that can disrupt the timely execution of real-time tasks. When an interrupt occurs, the CPU deviates from its main execution path to run the corresponding interrupt handler. This deviation often involves accessing a different code locality, which can generate traffic on shared micro-architectural resources such as LLC and main memory. Consequently, interrupts can introduce significant interference within the system, affecting the predictability of real-time operations.

This interference is problematic because it leads to unexpected delays and contention for shared resources, complicating the assurance of timely task execution. Even though recent research has tackled this issue, it remains a challenge. Studies such as [54] and [55]. have shown that the unpredictability of interrupts continues to be a significant obstacle in real-time systems. These works emphasize the need for strategies to manage interrupt handling more effectively to mitigate its impact on system performance and predictability.

2.2.2 Emerging and Established Architectural Trends

While homogeneous multi-core systems use identical cores, heterogeneous multi-core SoCs combine different types of cores, each optimized for specific tasks. This approach provides greater flexibility and efficiency since it enables the execution of diverse workloads by assigning them to the most suitable core type. For example, high-performance cores handle demanding computations, while energy-efficient cores manage less intensive tasks.

Considering the high heterogeneity of processing elements deployed on MPSoCs that act as managers—i.e., heterogeneous CPUs, GPUs, Direct Memory Access (DMA)s, and FPGAs sharing system resources like the memory controller, memory storage, I/O devices—hardware manufacturers provide a robust suite of hardware protection mechanisms to improve both spatial and temporal isolation guarantees. **Spatial isolation** ensures that a processing element accessing a shared resource prevents other processing elements from accessing its private data. **Temporal isolation** guarantees that the time behavior of a processing element is not affected by (or has a bounded effect on) the behavior of other processing elements,

even if those (partially) access the same shared resources. Additionally, to facilitate the integration of independent and unmodified software on the same platform, typically in the form of VM, hardware manufacturers have integrated into the SoCs sophisticated mechanisms for **multi-level address translation**, thereby decoupling the logical view of application-level (or guest-OS level) address spaces from the physical memory mapped to them.

Before going into the details of virtualization as a technique for the development of MCS we aim to provide a comprehensive summary and categorization of the various processor types and protection mechanisms employed on state-of-the-art MPSoCs, shedding light on their roles and scope within the considered class of platforms.

MPSoCs processors classes. Embedded MPSoCs are nowadays characterized by heterogeneous clusters of CPUs that can be categorized into three classes that feature different protection mechanisms:

- ***microprocessor-level CPUs***: Fully featured general-purpose multi-core CPUs characterized by all the modern hardware optimization techniques such as prefetching, branch prediction, cache coherence, as well as memory virtualization (Memory Management Unit (MMU)-based, see Section 2.2.3). These processors present at least three privilege levels to differentiate permissions and registers belonging to the hypervisor, the operating system, and the user-level applications. These are often referred to as APUs; an example is the cores belonging to the ARM Cortex-A family.

- ***microcontroller-level CPUs***: Specific-purpose CPUs that do not have any mechanism for memory virtualization (Memory Protection Unit (MPU)-based). They exhibit reduced hardware optimization techniques to improve simplicity and predictability. Furthermore, these microcontrollers usually support less than three privileged levels. This is because the software deployed on these CPUs is simpler and typically consists of a bare-metal application or, at most, a Real-Time Operating System (RTOS). An example includes the ARM Cortex-M and the ARM Cortex-R family, and often referred to as RPUs.

- ***programmable logic CPUs***: Highly specialized soft-cores deployed on re-programmable hardware to run code with specific requirements. Although these processors are extremely heterogeneous, their deployment on FPGA platforms enables communication with the rest of the system, mediated by system-level protections (see Section 2.2.3). This category includes soft-cores such as the AMD MicroBlaze [56], or the RISC-V Pico32 [57].

2.2.3 Spatial and Temporal Protection Mechanisms

The MPSoCs protection mechanisms can be systematically categorized into two distinct domains according to their roles: spatial isolation and temporal isolation. Furthermore, within the spatial isolation domain, a finer granularity can be achieved by distinguishing between two specific sub-types, namely spatial isolation with address translation and spatial isolation without address translation.

Spatial Isolation

Address Translation (MMU/SMMU): The MMU is the most known and used memory isolation mechanism for address translation. It is a component integrated into most microprocessor-level CPUs, serving a fundamental role in virtual memory management. The MMU maps virtual addresses to physical addresses, enabling applications (or guest OSes) to access memory locations in a manner that is transparent and independent of the physical memory layout. This mapping, maintained through page tables or hierarchical data structures, ensures that each process believes that it has its dedicated memory space, while efficiently sharing the physical memory resources with others. In the context of heterogeneous MPSoCs, the System Memory Management Unit (SMMU) is an extension of the MMU, tailored to manage memory and address translation for DMA-capable devices and accelerators. The SMMU plays a pivotal role in enhancing spatial isolation by extending address translation capabilities beyond the CPU cores. It enables virtualization and spatial isolation for a wider range of processing elements, including GPUs, Tensor Processing Unit (TPU)s, and programmable hardware such as FPGAs. Thus, in a heterogeneous MPSoC, both MMU and SMMU collectively

contribute to spatial isolation and virtual memory management, ensuring efficient resource utilization and system reliability. However, not all processing elements that can potentially assume the role of a manager on these boards are equipped with an MMU/ SMMU. Consequently, if not properly configured, certain managers can potentially access other managers' data in a manner that poses inherent security risks and/or results in poor fault containment, as evidenced in our evaluation.

Accesses Protection (MPU/SMPU/SPPU): Address translation mechanisms are not the only means of achieving spatial isolation in modern architectures. Microcontroller-level CPUs typically employed to run bare-metal software or RTOS do not necessitate address translation mechanisms. This is due to both the inherent cost of such mechanisms in terms of space occupation and energy consumption and the temporal unpredictability that MMU-based mechanisms introduce [58]. In these scenarios, CPUs are equipped with more straightforward mechanisms known as MPUs. These are implemented as hardware tables deployed between the manager (CPU) and the subordinate (Memory). Using the tables, an MPU divides the platform address spaces into fixed regions to enforce permissions to processes that access them. In heterogeneous MPSoCs, given that not all processing elements within these platforms possess address translation mechanisms, a comprehensive spatial isolation strategy is implemented by deploying system MPU-based protection mechanisms at the access port of important system resources. We term these system-level protection mechanisms System Memory Protection Units (SMPU) when used to protect memory; we use the term System Peripheral Protection Unit (SPPU) when they are used to protect memory-mapped I/O.

Temporal Isolation

Hardware Bandwidth Allocation: To manage memory traffic at the level of bus managers, modern ARM-based platforms support Quality of Service (QoS) mechanisms. Communication between a manager and a subordinate within an ARM-based platform is facilitated through the AXI protocol. The latest iteration of the AXI protocol, the AXI4 standard, incorporates a set of signals, specifically ARQOS and AWQOS, which convey traffic prioritization details essential to enforce bandwidth regula-

tion in QoS-aware on-chip memory. However, it is important to note that these additional details prove beneficial only when the subordinate components involved in processing transactions are QoS-aware. The QoS technology was initially introduced into MPSoCs with the primary objective of achieving load balancing. However, numerous studies have subsequently demonstrated its versatility and effectiveness in ensuring temporal isolation [59, 52]. However, there is a common trend in existing QoS-enabled platforms [60]: multi-core CPUs are typically treated as a unified manager. As a result, QoS support is primarily employed to regulate the aggregate traffic generated by all CPUs collectively. While this observation holds for main cores, it differs in the case of remote cores. These remote processors are usually equipped with distinct QoS ports for each CPU, a crucial distinction leveraged in the Omnivisor model to achieve temporal isolation between heterogeneous cores.

Software Bandwidth Allocation: Despite the QoS limitation in managing individual CPUs in a multi-core cluster, software solutions exist to regulate the bandwidth of the multi-core processors, offering per-CPU granularity that an Omnivisor shall leverage [50] [51].

2.2.4 Real-Time Processing Units and Virtualization

In modern MPSoCs, there is a growing trend to incorporate co-processors specifically designed for real-time computations, known as RPUs. These processors are typically simpler than contemporary cores, lacking certain optimizations like branch prediction and featuring very short pipelines to maximize predictability. Deploying real-time computations on separate RPUs can be highly beneficial for mixed-criticality systems, as it helps isolate time-critical tasks from less critical ones.

However, despite the advantages of using RPUs for real-time tasks, MPSoCs still face inherent unpredictability due to shared hardware resources such as memory controllers and interrupt systems. Even with RPUs, these shared resources can introduce latency and interference that undermine the predictability required for real-time computations. This necessitates careful consideration and management of these shared resources to ensure the effectiveness of RPUs in maintaining real-time performance.

Moreover, RPUs typically possess complete control over the platform,

which can be problematic in environments where maintaining strict control and isolation is critical. This control can potentially lead to security and safety concerns, as RPUs might access critical platform resources without adequate oversight.

Currently, there is no comprehensive solution addressing these issues. However, with the Omnivisor model we propose leveraging virtualization to give hypervisors better control and visibility over RPUs. By integrating RPUs into the virtualization framework, we can restore a hierarchical structure, ensuring that the hypervisor can manage and oversee the operations of these co-processors. This approach offers a new level of control over heterogeneous platforms, enhancing the predictability, security, and overall management of real-time tasks within mixed-criticality systems.

2.3 Towards Embedded Systems Virtualization

Although virtualization is currently a prominent technology utilized in various domains ranging from cloud computing to embedded systems, it is actually an older concept introduced in the 1970s by Popek and Goldberg [61]. The fundamental idea is straightforward, but over the years, virtualization has evolved significantly, adapting its form and expanding its role in computing environments.

Traditional Hypervisors. In the traditional hypervisor model, a virtualization layer is set between multiple software environments, namely virtual machines (VMs), and the underlying hardware. The responsibility of this layer is to abstract the physical hardware resources to the VMs to give them the illusion of running alone on the platform. To realize such abstractions, modern hypervisors take advantage of a combination of software mechanisms, including *hypercalls* and the *trap-and-emulate* technique. In addition, they leverage hardware mechanisms such as advanced MMU systems with dual stages of translation and support for multiple privilege levels within processor cores. This approach is designed to ensure spatial isolation between VMs, preventing one VM from accessing the data belonging to another VM while striving to maintain high performance and resource utilization levels. On top of this layer, hypervisors provide an interface for managing the VMs, allowing a high-privilege user

to create, stop, and control the resources assigned to VMs at run-time. Well-known open-source hypervisors that follow this model are KVM [62], Xen [63], and many others. These are widely used, and researchers have extended their capabilities to accommodate various use cases, including real-time scenarios [28, 64]. However, certifying these systems to be used in safety-critical scenarios can be challenging due to their inherent complexity.

Static Partitioning hypervisors. Real-time static partitioning hypervisors (SPHs), such as Jailhouse [29], Bao [30], Xtratum [65], and Quest-V [66], moves from traditional hypervisor model by adding resource separation constraints bearing the cost of less efficient use of resources to meet the requirements of real-time applications. In the SPH model, temporal isolation is as important as spatial isolation; therefore, they statically partition hardware resources between VMs to minimize shared components and mitigate temporal interference. According to this model, each VM gets a subset of the platform's resources; therefore, the CPUs are statically assigned to the VMs, and so are the memory, I/O devices, and accelerators. The separation is realized from the spatial point of view using hardware resources for address translation such as MMU and SMMU. Meanwhile, from the temporal point of view, the most recent approach proposed in [60] consists of combining CPU-centric bandwidth regulation techniques such as cache-coloring [67] and Memguard/Mempol [51][50] with hardware support for regulation such as QoS to isolate accelerators and CPUs.This model is easily applicable on top of traditional SoCs characterized by a single Multi-Core cluster, but how does the model apply on top of more complex and fully feature MPSoCs with heterogeneous core clusters?

2.3.1 Static Partitioning Hypervisors Shortcoming

Static Partitioning Hypervisor (SPH)s are currently designed to operate exclusively on microprocessor-level CPUs, with little or no consideration given to remote cores within the system, such as microcontrollers or soft-cores on FPGAs. In this scenario, deploying code on remote cores requires the system programmer to manually load the code and start the core. This is currently possible using two approaches: (I) using the bootloader and

thus at boot time or (II) using the Linux `remoteproc` driver on a VM at runtime. However, the former approach sacrifices the flexibility of dynamically halting and reloading code on the remote cores as needed, and the latter gives a VM full access to remote cores that can easily introduce time delays, interferences, or even system failures. Specifically, the remote cores are not isolated by default from the other virtual machines, and the code running on them can cause temporal and/or spatial isolation issues for the othe VMs by accessing the shared resources. To address this, a system programmer can manually configure and enable platform-specific hardware protection mechanisms, such as SMPU/ SPPU and QoS, to isolate the cores from the other VMs. Although effective, this approach diminishes the flexibility of the hypervisor and requires significant effort and specialized expertise. To actually maintain the isolation, every time a new VM is created, and every time a new code is loaded in the remote cores, the system developer must promptly reconfigure these mechanisms to isolate resources, otherwise risking data corruption or possible interference between cores.

An SPH on heterogeneous MPSoCs should ensure holistic protection across the entire board, transparently to the user. It should handle isolation seamlessly, avoiding the need for manual programming of specialized hardware protection mechanisms and providing a more user-friendly and robust solution for running code on asymmetric multi-core systems.

2.4 Related Work on Hypervisors for MPSoCs

2.4.1 Partitioning Systems

Numerous real-time hypervisors and microkernels proposed in the literature are engineered with partitioning techniques aiming to explicitly meet certifications such as ARINC-653 and AUTOSAR [68, 69]. Instead, the Omnivisor distinguishes itself by offering partitioning with spatio-temporal isolation for a diverse range of processor categories. Unlike works such as [70], which propose a partitioning microkernel-based design targeting microcontrollers-level cores, and [68], which propose an ARINC-653 scheduling on Xen focusing microprocessor-level cores, Omnivisor addresses the challenge of applying partitioning to asymmetric core platforms

by leveraging different isolation mechanisms for each category in a coordinated manner. Although this work's focal point is not about certification, the Omnivisor aims at establishing the blueprint of a partitioning hypervisor for heterogeneous systems which is the first step for future certification endeavors.

2.4.2 Asymmetric Multi-Core Architectures

The management of asymmetric multi-core architectures is a well-explored field within the systems software community, which has proposed OS designs [71, 72, 73, 74] and hypervisors [75, 76] capable of fully leveraging heterogeneous platforms. However, these existing works are not directly comparable to the Omnivisor, since they often overlook the isolation challenges that heterogeneous cores can introduce, making them unsuitable for mixed-criticality scenarios. In [77] the authors discuss the challenges and opportunities of asymmetric architectures, proposing the OpenAMP framework as a solution for remote core communication and power management. Despite the framework is not meant for mixed-criticality, the works in [8, 78] and [79] explore the possibility of using such a framework in critical scenarios. Both approaches focus on real-time communication with remote cores, overlooking the interference between cores. In contrast, the Omnivisor aims to provide spatio-temporal isolation between asymmetric cores, offering a complementary solution that will incorporate real-time communication in the future.

2.4.3 MPSoCs Hypervisors

Some recent works have been proposing techniques to virtualize heterogeneous platforms featuring programmable logic (FPGA) as well as heterogeneous processors, to realize reliable mixed-criticality systems.

Moratelli *et al.* propose a real-time full-virtualization technique for MPSoCS [80]. While this work provides a solution to run unmodified software on a traditional hypervisor with real-time requirements, the Omnivisor is an extension for partitioning systems where the resources are statically allocated to virtual machines and there is no need for schedulers.

Gracioli et al. [44] explore the capability to run mixed-criticality systems in MPSoCs where an SPH is deployed on APUs to isolate resources. The paper outlines how the rich hardware features provided by modern heterogeneous SoCs can reduce the contentions between partitioned applications. However, while this work analyzes the optimal utilization of heterogeneous resources such as diverse scratchpad memories, aspects not considered in our work, it overlooks the threat posed by unrestrained microcontroller-level CPUs. In contrast, Omnivisor (Chapter. 4) focuses on addressing temporal and spatial isolation issues between asymmetric cores. It also offers flexible and seamless control over remote cores through the hypervisor.

CHIPS-AHOy is a predictable holistic hypervisor [81] that aims to satisfy temporal predictability and high-performance requirements of software running over MPSoCs while simultaneously handling energy efficiency, thermal bound, and system lifetime. The authors' goal is to address the most relevant source of unpredictability in MPSoCs, such as the memory hierarchy, the I/O subsystem, and the hardware variability, by using techniques such as cache coloring and I/O throttling. However, the authors do not provide a common interface to manage heterogeneou VMs and neither consider using bandwidth regulation mechanisms to improve temporal isolation.

Biondi *et al.* present the SPHERE project [82], an integrated framework to abstract the hardware complexity of MPSoCs and simplify the management of heterogeneous hardware. The work explores the interesting possibility of using the dynamic partial reconfiguration of the FPGA to provide efficient implementations for cryptography modules, as well as hardware acceleration for deep neural networks in a hypervisor-based system. However, the authors do not explore asymmetric Instruction Set Architecture (ISA) cores as the Omnivisor (Chapter. 4, and instead focus solely on accelerators. While there is a strong effort in the literature to develop virtualization systems that utilize FPGA, existing works primarily focus on sharing the FPGA among Virtual Machines running on the main cores [83, 84]. In contrast, our model (Chapter. 4 acknowledges the presence of cores in FPGA, which run entire and isolated VMs.

Although the Omnivisor model we present in this dissertation has similar objectives to those described in related work, that is, to realize a

mixed-criticality system with strong real-time guarantees for critical VMs and to streamline the use of heterogeneous systems, it may be distinguished primarily by three points. First, it is the first hypervisor model that considers running isolated VMs on cores with heterogeneous ISAs as equal from the point of view of the hypervisor interface. This simplifies the adoption of such complex platforms and improves the overall system reliability. Secondly, unlike other solutions, it dynamically coordinates a combination of modern heterogeneous hardware protection mechanisms at runtime (including MMU, SMMU, SMPU/SPPU, and QoS) to provide spatial-temporal isolation to heterogeneous cores, transparently to the user. Finally, it is the first approach that considers using the soft-cores deployed on FPGA as isolated domains where to run VMs.

Before diving into the Omnivisor model in the next chapter we want to motivate, using real tests cases, the reason to choose partitioning hypervisor in safety critical environments. To do it we also propose a standardized DoE-based approach to assess the temporal isolation of virtualized environment and a standardized way to tune the parameters offered by modern hypervisor in order to improve the temporal predictability of our systems.

3

Temporal Assessment and Modeling of Mixed-Criticality Virtualized Systems

THIS CHAPTER explores the practical challenges of configuring hypervisors to deploy MCS using virtualization, as well as modeling it within an IoT-fog-cloud continuum [85].

Hypervisors provide various configuration options, such as scheduler parameters and CPU affinity, which can significantly affect the performance of VMs. Given the vast number of possible configurations, manually testing each setup to find the best for a specific application is impractical. To address this, we introduce a Design of Experiments (DoE)-based methodology to systematically evaluate temporal isolation across different configurations, reducing the number of experiments needed. The proposed DoE method allows us to rigorously estimate the importance and significance of the numerous existing factors in virtualized scenarios (e.g.,

different real-time schedulers at hypervisor level), against various operational scenarios (e.g., virtual CPU/physical CPU sharing ratios, external disturbance, etc.).

Additionally, to systematically integrate the system in a more complex IoT-fog-cloud scenario, such as the one described for nuclear fusion in (Section 1.2), we propose a MCS deployment model in Section 3.4.

To test the proposed design we examine the capabilities offered by the real-time flavor of the Xen hypervisor running on top of an ARM-based Xilinx Zynq Ultrascale+ board family (see Section 2.2), which are popular choices as a basis for safety-critical virtualized systems [86, 87]. We focus on Xen since, over the years, the community has provided several features for enabling industrial settings. Indeed, current versions of Xen include support for real-time applications along with built-in cloud computing support (e.g., orchestration), minimal size for ARM architectures (less than 50KSLOC), paravirtual and GPU mediation for rich I/O, Trusted Execution Environment (TEE) virtualization support, and static partitioning (i.e., Dom0less architecture), as well as undergoing work for safety certification [88, 89, 90].

We implement a virtualized 2 out of 2 (2oo2) schema which is one of the most common fault-tolerant approaches and a summary of the main findings of the experiments are in the following:

- Adjusting configurable knobs, such as schedulers and their parameters, can significantly influence the temporal isolation of the system. Our approach allows us to tailor the configuration according to specific requirements, such as prioritizing high performance for certain VMs over others with soft real-time requirements.

- When we statically assign each virtual CPU to a physical CPU, we observe no significant difference in predictability by varying the Xen schedulers, even when external CPU-related disturbances are enabled.

- Despite the static allocation of virtual CPUs to physical ones being the preferred option for real-time scenarios, it still leads to a severe increase in the mean and standard deviation of maximum execution time when the system experiences various types of stress.

- In a *privileged VM*-based hypervisor (like Xen), the privileged VM (namely *Dom0* in Xen), heavily influences the predictability of the software running on it compared to unprivileged VMs (namely *DomU* in Xen).

The objective is to provide guidelines that allow the development and fine-tuning of virtualized MCSs, to document temporal isolation evidence required for certification purposes. These results highlight the challenges that need to be addressed in real scenarios such as nuclear fusion reactors when deploying mixed-criticality software using virtualization. Therefore, based on this analysis, we identify a set of directions toward the real use of virtualization in the context of ARM-based systems. Specifically, this observation serves as the key motivation for developing the Omnivisor model (Chapter. 4), which aims to solve the problem of asymmetric cores virtualization, starting from static partitioning hypervisors to maximize the isolation between VMs.

3.1 Systematic Approach and Workflow

The proposed temporal isolation assessment approach follows the steps highlighted in 3.1. The proposal includes a systematic approach to unveil potential interference in a virtualized MCS under corner case conditions (e.g., stressful privileged/unprivileged VMs, faults in the hypervisor, bad configurations, etc.). The ultimate objective is twofold: i) quantifying how strong the selected hypervisor assures the temporal isolation, and ii) providing guidelines for industry practitioners to fine-tune the system parameters in case of weak isolation. The main idea is to leverage the well-known *DoE* to rigorously assess the *impact* and *significance* of virtualized system parameters in real operational scenarios.

3.1.1 Analysis

In the **first step** (① in Figure 3.1), industry practitioners need to analyze the target safety-related standard to extrapolate *temporal isolation properties*, then *recommended mechanisms* (if specified) to assess such properties.

Table 3.1 shows how four different safety-related standards, i.e., ISO 61508 [91], DO-178C [92], ISO 26262 [16], EN 50128[14], treat or men-

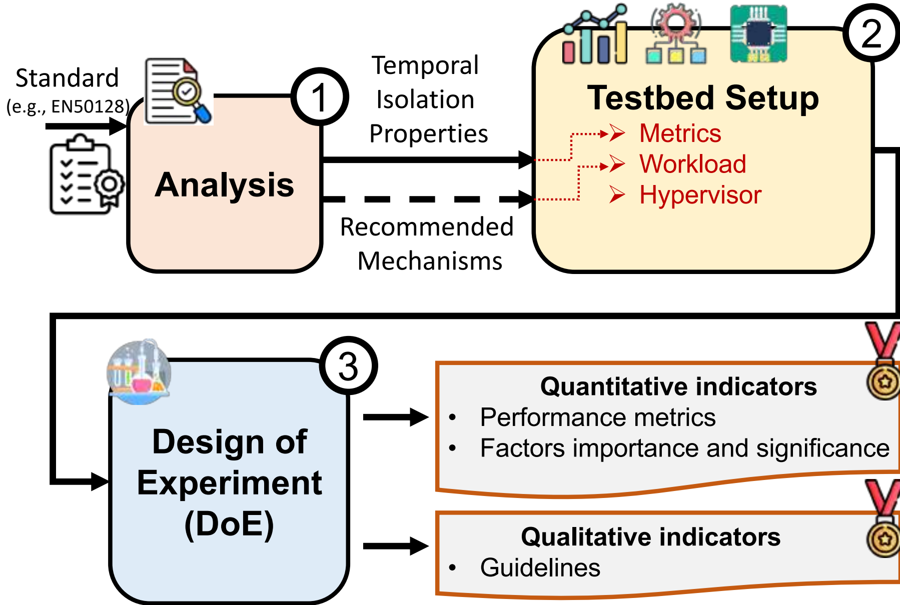


Figure 3.1: Proposed Temporal Isolation Assessment Workflow.

tion temporal isolation properties to be met according to specific level of safety. Depending on the standards, the safety level is indicated as Safety Integrity Level (SIL), Automotive Safety Integrity Level (ASIL), Software Safety Integrity Level (SSIL), or Design Assurance Level (DAL). Furthermore, some standards also mention specific techniques to ensure these properties. For instance, DO-178C and IEC 61508 recommend achieving temporal predictability through fixed cyclical scheduling, time-triggered scheduling, fixed priority-based scheduling, CPU execution time monitoring, or WCET analysis for temporal isolation purposes.

In the context of virtualized MCSs, temporal isolation includes the ability to isolate or limit the impact of resource consumption (e.g., CPU, network, disk) of a VM on the performance degradation of other VMs and even against the host. This means that a task running in a VM must not cause delays to other critical and non-critical tasks running in different VMs, avoiding phenomena such as starvation, reduced throughput, and increased latency. Indeed, by adding this new level of software in-

Table 3.1: Temporal isolation properties across safety-related standards.

Domain	Standard	Details	Reference
Generic	IEC 61508, Part 3 [91]	*"Temporal: one element shall not cause another element to function incorrectly by taking too high a share of the available processor execution time or by blocking the execution of the other element by locking a shared resource of some kind."*	F.2, F.5 (Annex F)
Avionic	DO-178C [92]	*"A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution."*	Section 2.4.1.b
Automotive	ISO 26262, Part 6 [16]	*"With respect to timing constraints, the effects of faults such as those listed below can be considered for the software elements executed in each software partition: blocking of execution; deadlocks; livelocks; incorrect allocation of execution time; incorrect synchronization between software elements."*	D.2.2 (Annex D)
Railway	EN 50128 [14]	*"An analysis is performed which will identify the distribution demands under average and worst-case conditions. This analysis requires estimates of the resource usage and elapsed time of each system function. These estimates can be obtained in several ways, for example, comparison with an existing system or the prototyping and benchmarking of time critical systems."*	D.45 (Annex D)

direction (i.e., the hypervisor), we need to take into account additional knobs, e.g., the guest OSes schedulers, vCPUs/pCPUs mapping, the hypervisor scheduler, and so on: if not properly managed they could easily break temporal isolation requirements.

3.1.2 Testbed Setup

Once specified what isolation properties should be monitored and verified, the **second step** (② in Figure 3.1) includes *testbed setup*. This step consists of i) selecting metrics that reflect the temporal isolation properties identified in the previous step; ii) defining a workload that involves average and worst-case conditions; the workload can reflect recommended mechanisms proposed by the target standard to verify temporal isolation properties; iii) selecting the target hypervisor to deploy the virtualized MCS. In turn, the hypervisor selection includes several knobs to be manipulated, e.g., how many guests need to be used for running the target workload, guest OSes schedulers and real-time patches to be applied (e.g., PRE-

EMPT_RT), vCPUs/pCPUs affinity, hypervisor scheduler, hardware/-software isolation mechanisms (e.g., cache coloring [67, 93, 94], memory bandwidth reservations [93]). Clearly, if these features are not properly tuned, they could easily cause temporal isolation violations.

3.1.3 Design of Experiment (DoE)

The **third step** is the *Design of Experiment (DoE)* [95] (③ in 3.1). DoE enables a systematic way to unveil the relationship between the key factors in a virtualized deployment (e.g., vCPU/pCPU affinity) and the variation of information according to the variation of these factors (e.g., the standard deviation of application latencies). In particular, the DoE allows planning experiments so that appropriate data can be analyzed by statistical methods that result in valid, unbiased, and meaningful conclusions. Realizing a DoE involves a *design phase*. This latter consists of choosing a *response variable* of interest and a set of controllable *factors* that we expect to affect it. Each factor is characterized by different values called *levels*, and each experiment has to be replicated to reduce as much as possible the statistical error. By performing enough experiments, it is possible to define which of the factors is more *important* and *significant* than the others and which level of each factor allows for improvement in the response variable. Subsequently, DoE exploits statistical methods such as the ANOVA analysis [96], which includes a set of guidelines to extract statistically meaningful information from data to provide numerical evidence. Eventually, the DoE step produces both *quantitative* and *qualitative indicators*. The former is related to performance metrics, such as throughput and latency, and the statistical importance and significance of chosen factors during *step 2*. The latter, instead, includes guidelines about providing the best combination of testbed configurations to achieve the desired temporal isolation.

3.2 Temporal Isolation Assessment

The proposed temporal isolation approach is applicable in several domains, including Automotive, where adherence to the ISO-26262 standard underscores the importance of isolation guarantees. Similarly, in Avionics, com-

pliance with DO-178C standards emphasizes the need for robust temporal isolation mechanisms. Even in cutting-edge fields such as nuclear fusion, virtualization finds utility, as demonstrated in [78].

The target of our analysis is a 2oo2-based mixed-criticality system where a simple control application is replicated in two different VMs and a voter mechanism is deployed on a different VM to assess the correctness of the replica's output. When an error is notified, the system should be brought to a fail-safe state. This is a typical scenario for safety-critical applications such as the controllers in nuclear fusion reactors. The objective is to determine in which conditions it is possible to employ virtualization to migrate existing applications and reduce SWaP-C.

3.2.1 Step 1: the Standard Analysis

Given the context, the adopted standard will guide the developers for the `Software Verification` and `Architecture and Design` activities. Specifically, developers should provide the `Software Verification Plan` and `Software Integration Test Specification` documents, both of them refer to `Performance Testing` requirement.

This analysis provides the temporal isolation properties to be verified (*resource usage* and *elapsed time*) of target critical system function by *comparing* different configurations, and leveraging *benchmarks* to unveil potential issues.

3.2.2 Step 2: Xen on ARM-based MPSoC Testbed Setup

Following the *step 2* in the workflow depicted in the previous section, we deployed a 2oo2-based MCS testbed (see Figure 3.2). The 2oo2-based MCS is consolidated via Xen v4.16 hypervisor, all on top of the ZCU board (described in Section 2.2).

The implemented 2oo2-based MCS application (see Figure 3.2) includes two replicas (i.e., *VM A* and *VM C* running as *DomUs*) that periodically produce measurements to be assessed by a voter (i.e., *VM B* running within *Dom0*) that compares them once received. The period of the replicas and voter is $20ms$; the period was chosen to have enough slack time to verify the temporal behavior in various stress conditions and at the same time to speed up the DoE step, which includes several factors

and test repetitions. After sending the measurement, each replica sleeps until the next period. As soon as a value from each replica is received, the voter compares them, returning an *ACK* to replicas within the next period. All the VMs leverage UNIX sockets for inter-VM communication. This choice was made since our main objective is to reuse (unmodified) legacy software in a virtualized environment. This setting is also useful in unveiling whether the utilization of the network software stack can lead to unpredictability even if it is used inside the same platform.

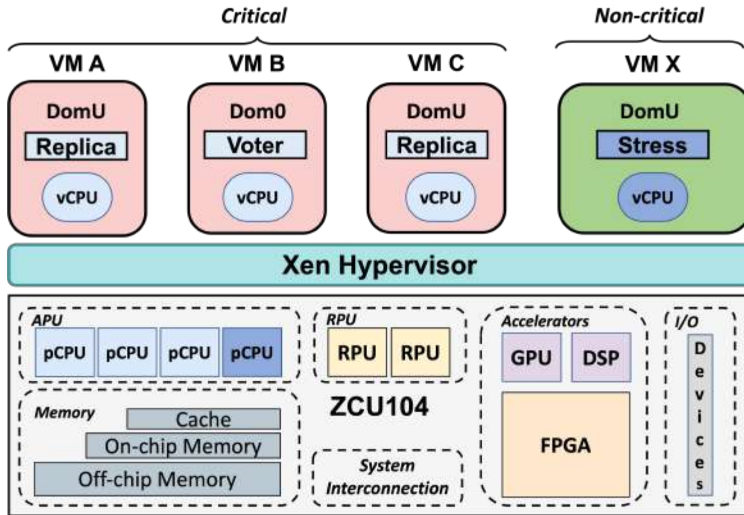


Figure 3.2: The Xen-based virtualized 2oo2-based testbed used for experimentation.

The testbed described before is realistic in terms of the replication schema commonly used in the industry domain. However, what is crucial for us is to understand how to fine-tune the critical knobs mentioned before to build a virtualized MCS properly. To this end, during experimentation, we launch a stress VM, i.e., *Stress DomU* (*VM X* in 3.2), which runs side by side with replicas and voter domains to emulate a co-located non-critical workload. The purpose is to cause a high computational load that might result in interference between the existing critical domains. To introduce the desired stress (e.g., CPU, network, I/O), we used *stress-ng* [97], a versatile stress testing tool suite, also used in indus-

Table 3.2: Experimental factors and levels

Factors	Levels								
Host-level scheduler	Credit2				RTDS				null
Scheduler parameters	Rate Limit [μs]				Budget/Period [μs]				N/A
	100	500	1000	5000	1k/5k	3k/9k	5k/10k	10k/10k	
CPU affinity	One-to-One					Many-to-Many			
Stress Tests	Yes							No	
	CPU	Cache	Device	I/O	Intr	FS	Net		

trial domains [98, 99], designed to assess system stability and performance by inducing various types of stress on hardware components. It offers a wide range of stressor classes, including CPU, memory, disk, network, and other subsystems, allowing for comprehensive testing of system resilience under diverse workloads. Clearly, in an in-production scenario, the testbed in 3.2 can be implemented by adopting multiple MPSoCs running spare versions of both replicas and the voter. However, we aim to unveil temporal isolation breaks induced by the hypervisor.

As mentioned before, we isolate each replica in different VMs (*DomUs* in Xen jargon) and deploy the voter application on the most critical domain in Xen, namely *Dom0*. It is important to specify that in a realistic scenario, the voter is a very high-critical component that needs to be implemented on dedicated hardware with hard-wired logic for real-time communication. However, we still decided to deploy the voter on *Dom0*, both to facilitate the testing phase and to understand how much this domain affects the execution time of replicas within *DomUs*. This analysis is crucial since by default Xen comes with *Dom0* anyway, and avoiding running applications on top of it would be a waste of resources, which are already scarce in real-world embedded scenarios, especially in the case of strict partitioning, when physical hardware resources are statically assigned to VM and not shared. We compare different configurations of Xen to verify which is the best one according to the requested real-time requirements.

Regarding the metric selection task in the proposed workflow (see *step 2* in 3.1), we choose the replicas *maximum execution time* as the worst-case metric for comparing all the target hypervisor configurations, where each replica runs a periodic task that produces data and sends it to the voter

whose check whether the data produced by the two replicas is equivalent. The testbed knobs we decided to manipulate are the host-level schedulers and the related configuration parameters (e.g., budget/period ratio for *RTDS* scheduler), and the affinity between virtual CPU of VMs and physical CPUs, i.e., *CPU affinity*). Therefore, we set a *PREEMPT_RT*-patched Linux, enabling the *SCHED_FIFO* policy for each guest OS. *PREEMPT_RT* patch is known to provide the best real-time guarantees (lowest kernel-induced latencies) for Linux environments [100, 101]; further, we leveraged *SCHED_FIFO* policy since each application (the voter and the replicas) is single-threaded, and *SCHED_FIFO* is the simplest Linux scheduler with real-time capabilities. We discarded any of the new experimental techniques proposed in the literature that improve resource isolation (e.g., *Memguard* [50], *cache coloring* [67], *I/OGuard* [102]), due to either no support or still ongoing work for Xen. The ultimate objective is to have a simple and easily reproducible scenario, as well as a precise worst-case scenario.

Given this scenario, we can understand if some configurations exhibit more interference than others by precisely estimating the significance of factors via the DoE approach (see *step 3* in 3.1), which is described in the following.

3.2.3 Step 3: Design of Experiments

The last step within the proposed workflow is defining the factors and levels for DoE use. 3.2 shows their description.

The first factor is the **host-level scheduler** that, as mentioned above, includes all the possible Xen schedulers except for the old version of *Credit* and for ARINC-653, which only supports one CPU core per guest and wastes compute resources for virtual guests that do not fully utilize compute time. Thus, we consider *RTDS*, *Credit2*, and *null* schedulers as levels for that factor.

We assume the host-level **scheduler parameters** as a second factor. Regarding *RTDS*, the scheduler parameters include the *budget* and the *period* of the scheduler; specifically, we give to the VMs one-fifth, one-third, one-half, and finally the entire budget in each period as levels. For the *Credit2* scheduler, we decided to vary the *rate limit*, i.e., the minimum execution time interval of a vCPU before a context switch. The valid

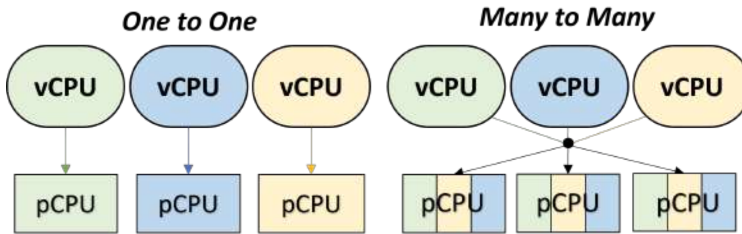


Figure 3.3: CPU affinity factor and its levels.

range in Xen is 100μs to 500000μs (500ms), but since the replicas and voter period were set at 20ms in our experiments, and we have four VMs, we decided to reduce the range from 100μs to 5000μs (5ms) so that at least once in a period all VMs can run. Regarding *Credit2*, Xen also allows modifying the *weight* associated with each VM, where the weight indicates how much physical CPU to reserve for one VM compared to the others. However, this parameter is too application-dependent, so we set the same weight for each VM even though it could not be the optimal solution. Lastly, for the *null* scheduler, there are no parameters modifiable.

Therefore, we assume the affinity level of vCPUs over available pCPUs as the third factor, namely, **CPU affinity**. Specifically, we focus on two scenarios depicted in Figure 3.3: the first one is where each vCPU (thus, each VM) is pinned on a dedicated pCPU (i.e., the *One-to-One* level); the second case is where more vCPUs (thus, more VMs) are not pinned to a specific pCPUs but run on multiple shared pCPUs (i.e., the *Many-to-Many* level). In our experimental scenario, each critical VM (replicas and voter) has a single vCPU, and we can use 3 out of 4 pCPUs available on the target board (i.e., Zynq Ultrascale+ ZCU104) for these VMs since the fourth pCPU is dedicated to the non-critical stress VM. Finally, the *One-to-One* level implies that each VM has one pCPU at his own disposal, meanwhile the *Many-to-Many* level implies that the three critical VMs share three pCPUs. However, in case the host-lost scheduler is *null*, the only *CPU affinity* level available is *One-to-One*; this is because *null* scheduler by default assigns each vCPU to a single pCPU not allowing any kind of scheduling of vCPUs over several pCPUs.

Finally, the **stress** factor indicates the presence/absence of the stress VM,

where 8 possible levels of stress were considered if the stress VM is enabled. In particular, we focused on *CPU* (CPU intensive), *Cache* (stress CPU instruction and/or data caches), *Device* (raw device driver stressors), *I/O* (generic Input/Output operations), *Intr* (high interrupt load generators), *FS* (file system activity), and *Net* (TCP/IP, UDP, and UNIX domain socket stressors).

To assess the temporal isolation provided by Xen, we need to record the time needed by the replicas for sending measurements. In particular, we measure the execution time, of both replicas, $n = 2000$ times for each configuration test, then we take the *maximum execution time* among the n samples. To provide statistical significance in our experimentation, and to remove the noise, we performed 30 repetitions for each experimental configuration, by also resetting the target board, for each repetition, to assure test independence. Considering the DoE described in table 3.2 we have 4 factors with respectively 3, 4, 2 and 8 levels. Using a *Full Factorial Design* we would have a total of 192 potential experiments to conduct, repeated 30 times, for a total of 5.760 experiments. However, some of these configurations are not implementable; for example, *null* scheduler has just one possible level for both *scheduler parameters* and *CPU affinity* factors. Furthermore, we decided to not consider all the stress classes (CPU, Cache, Device, etc.) for each possible combination of the other three factors but we decided to test first the CPU stress class fixing the best scheduler parameters factor and varying the Scheduler and the CPU affinity factors and then to test all the eight stress classes fixing the best level for each of the other factors. In this way, the number of final experiments is reduced to only 29 such as to allow us a more targeted analysis of our data.

3.3 Temporal Isolation Results and Analysis

This section presents the experimental results from the tests conducted on the established testbed setup and experimental design outlined earlier.

To ensure clarity, we initially present the results by isolating each scheduler and analyzing its performance under various scheduler parameters configurations and CPU affinity. This approach allows us to discern the optimal configuration for each scheduler and gain insights into their respective behaviors.

Subsequently, we fix the best configuration found for each scheduler and compare the schedulers with each other. To perform a fair comparison, we first compare the schedulers when the physical CPUs are fully shared (*Many-to-Many* scenario), and then we compare the schedulers when the physical CPUs are not shared (*One-to-One* scenario), thus including *null* scheduler in the comparison. Having identified the most optimal scheduler and configuration based on the mean and standard deviation of the response variable (i.e., maximum execution time), we proceed to evaluate its performance under various stress conditions introduced by executing stress tests on one isolated stress VM.

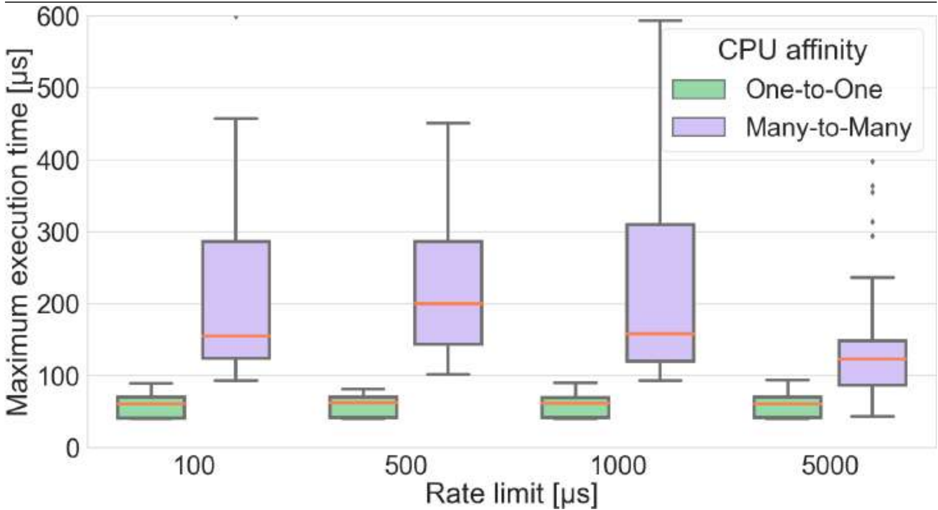
3.3.1 *Credit2* Analysis

The *Credit2* scheduler is the Xen default non–real-time scheduler. As mentioned, we compute the maximum execution time, performing 30 repetitions w/ and w/o the *stress DomU*, configured with a CPU stressor, to reveal potential isolation breaks. We could use a two-way ANOVA with interactions and repetitions to verify if both factors (*CPU affinity* and *rate limit*) induce a significant difference in the response variable (i.e., the *maximum execution time*). To apply it we need independence of observations, the homoscedasticity, and the normality of the residuals [96]. To ensure as much as possible independence between observations, we reset the target hardware for each experiment. Hence, we have to check for homoscedasticity and normality of the residuals. The residuals are normal according to the performed Shapiro-Wilk test (i.e., $W = 0.83$ and p-value < 0.0001); however, the homoscedasticity assumption is rejected by the Levene test with high confidence from each type of test. In such conditions, we must proceed with a *heteroscedastic ANOVA* test [103], and we decided to use Welch's ANOVA test. The results indicate statistical significance of both factors with a very low p-value (< 0.0001), i.e., variation in the levels of both *CPU affinity* and *Rate Limit* factors implies significant variation in the response variable.

3.4a and 3.4b show the variability of the *maximum execution time* in one of the replicas used in the 2oo2-based application, according to the variation of *CPU affinity* and *rate limit* factors. The sharing of physical CPUs implies a large increase in average maximum execution time as well as greater standard deviation. Furthermore, in the case of non-

(a) Means (*M*) and standard deviations (*SD*) of maximum execution time for replica VMs. *One-to-One* is highlighted with green (■), while *Many-to-Many* with magenta □).

Rate Limit [μs]	*100*	*500*	*1000*	*5000*
M [μs]	56.62	56.81	56.19	57.33
	211.50	227.07	217.87	134.72
SD [μs]	15.42	14.07	14.58	15.96
	116.22	107.00	131.48	76.04



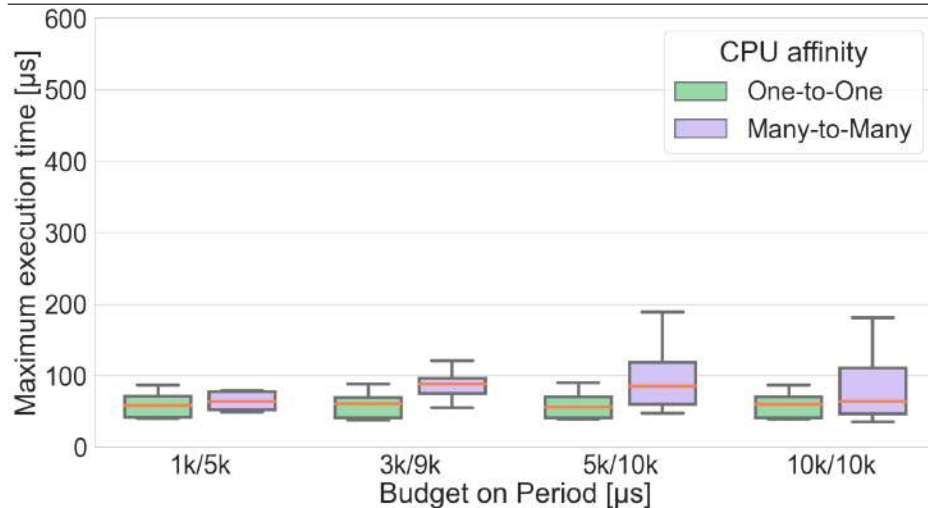
(b) Boxplots depicting the maximum execution times of replica VMs.

Figure 3.4: *Credit2* host-level scheduler analysis, by varying *rate limit* i.e., the minimum execution time interval of a vCPU before a context switch.

sharing of pCPU (i.e., the *One-to-One* scenario), the *rate limit* factor, for each level, does not lead to any noticeable change, neither in the average nor in the standard deviation of the maximum execution time. On the other hand, in the *Many-to-Many* case, the configuration with the highest *rate limit* (i.e., 5000us) has both a lower average of maximum execution time and lower standard deviation, thus is the best configuration for this type of scheduler in this scenario. This is because this configuration causes as few context switches as possible.

(a) Means (*M*) and standard deviations (*SD*) of maximum execution time for replica VMs. *One-to-One* is highlighted with green (■), while *Many-to-Many* is highlighted with magenta □).

Budget/Period [μs]	*1k/5k*	*3k/9k*	*5k/10k*	*10k/10k*
M [μs]	57.78	56.54	55.82	56.87
	64.28	85.68	92.32	80.88
SD [μs]	15.06	14.38	15.41	15.76
	12.60	15.59	37.34	45.01



(b) Boxplots depicting the maximum execution times of replica VMs

Figure 3.5: Analysis of *"RTDS"* host-level scheduler factor, while varying the *budget/period* in microseconds.

3.3.2 *RTDS* Analysis

Same as with *Credit2*, we leveraged two-way ANOVA with interactions and repetition for *RTDS* level for host-level scheduler factor. Accordingly, the *scheduler parameters* factor is the ratio between *budget* and *period*. Again, the Shapiro-Wilk test demonstrates the normality of the residuals ($W = 0.9$ and p-value < 0.0001) and the Levene test proves the heteroscedasticity. Therefore, in this case, as well, we leveraged Welch's ANOVA test to prove the significance of our factors with high confidence (p-value < 0.0001 for both factors).

3.5a and 3.5b show the difference between maximum execution time by varying the CPU affinity factor. In contrast with *Credit2*, *RTDS* is much more robust to variations, as we expected. Anyway, as for *Credit2*, in the case of the *One-to-One* scenario, the variation of factor levels does not affect significantly the maximum execution time; however, in the *Many-to-Many* case, it is clear that the configuration set with one-fifth of the bandwidth, i.e. the *1k/5k* level, guarantees very low variation both in the mean and standard deviation of the response variable. Thus, we can state that the *1k/5k* configuration seems to be the best configuration for *RTDS* in our scenario.

Credit2 and RTDS Analysis

★ ***Credit2* and *RTDS* scheduler parameters** (rate limit and budget/period, respectively) **heavily impact temporal isolation**. Fixing the best scenario, in which no pCPUs are shared between VMs, comparing to vCPU/pCPU sharing enabled:

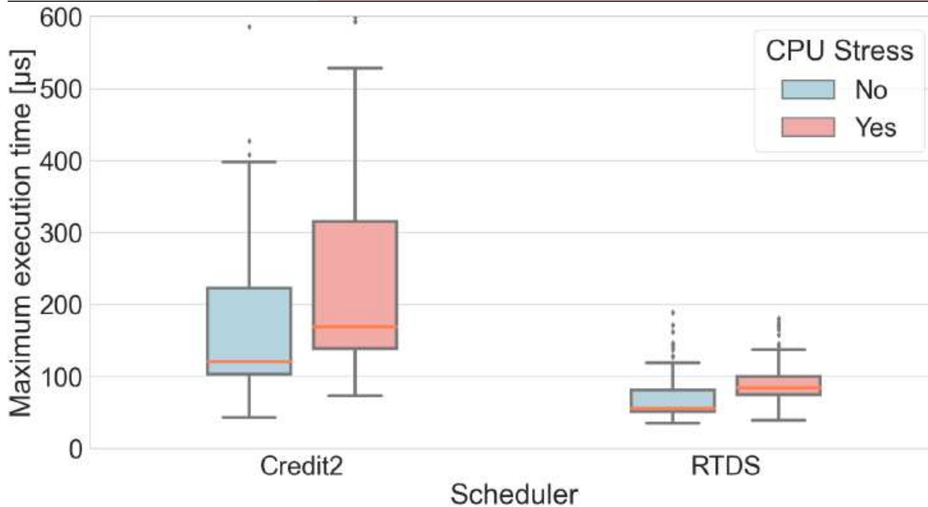
- *Credit2* leads to a noticeable deterioration in the standard deviation of maximum execution time, i.e., $+801\%$ and $+376\%$, in the worst (rate limit equal to 1000μs) and the best (rate limit equal to 5000μs) cases, respectively;
- *RTDS* leads to a smaller increase in the standard deviation of maximum execution time, i.e., $+185\%$ in the worst case (budget/period equal to $5k/10k$). The best case (budget/period equal to $1k/5k$) includes no statistically significant difference by enabling or disabling pCPUs sharing between VMs.

3.3.3 *Credit2* vs *RTDS*: Many-to-Many Analysis

In this section, we take into account scenarios in which there is a strict requirement for using more virtual CPUs than the available physical CPUs. This is the common case for high consolidation degrees in virtualization environments. However, these kinds of scenarios, especially in the presence of sources of disturbance, can heavily affect real-time guarantees. To evaluate the robustness of Xen in the worst case, against these scenarios, we performed a comparative analysis of *Credit2* and *RTDS* in the case of *Many-to-Many* level for *CPU affinity* factor. Precisely, we fixed the best-case configurations of both*Credit2* (rate limit equal to *5000*)

(a) Means (M) and standard deviations (SD) of maximum execution time replica VMs. *Stress* factor disabled is highlighted with blue (□), while *stress* factor enabled (CPU level) is highlighted with red (■).

Scheduler	*Credit2*	*RTDS*
M [μs]	163.13	70.01
	232.46	91.62
SD [μs]	94.84	30.74
	123.01	30.63



(b) Boxplots depicting the maximum execution times of replica VMs

Figure 3.6: Analysis of *"Credit2"* and *"RTDS"* host-level scheduler factors, fixing the *CPU affinity* factor to the *Many-to-Many* level, and by enabling/disabling *stress* factor, CPU level. Parameters for both schedulers are configured based on the optimal results attained in prior experiments.

and *RTDS* (budget/period equal to *1k/5k*) in the *Many-to-Many* case (see 3.3.2). However, in this case, instead of joining the repetitions w/ and w/o the stress DomU, we provide two different isolated data samples for each scheduler; one with the stress DomU (configured with a CPU stressor), and one without.

In this scenario, the stress VM runs on a dedicated physical CPU, and can be configurable according to the desired stress workload: we leveraged *CPU intensive stress* class workloads provided by *stress-ng* (e.g. Fi-

bonacci, factorial, etc.). This is to understand how robust are *Credit2* and *RTDS* against such kind of disturbance. From a DoE point of view, we consider two factors with two levels, respectively, i.e., the *stress* factor (specifically the *CPU* stress level) with $\{yes, no\}$ levels, and *host-level scheduler* factor with $RTDS, Credit2$ levels. We still used the ANOVA test in this scenario, so we need to verify the normality and homoscedasticity of the residuals. The results show that residuals are normally distributed according to the Shapiro-Wilk test ($W = 0.88$ and p-value < 0.0001), and heteroscedastic according to the Levene test. Also, in this case, Welch's ANOVA test suggests that both factors are significant with a high level of confidence (p-value < 0.0001).

As we expected, 3.6a and 3.6b show that *RTDS* is the most suitable scheduler for real-time setting. Indeed, the standard deviation of the maximum execution time is much lower than *Credit2*, and in addition, the mean variation w/ and w/o stress enabled is also lower.

Credit2 vs *RTDS*: Many-to-Many Analysis

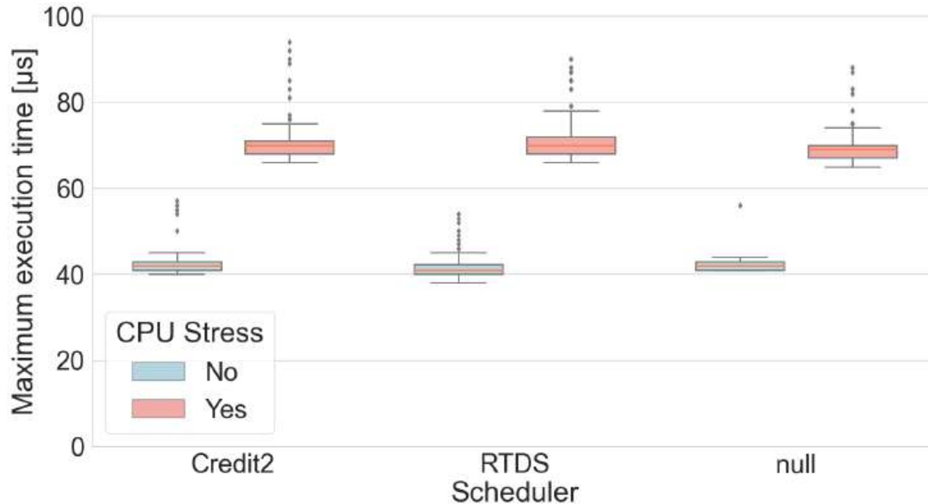
★ ***RTDS* is the most suitable scheduler for real-time setting when vCPU/pCPU sharing is enabled**, even enabling external CPU-related disturbances. Indeed, the standard deviation (+30% vs +0%) and means (+42% vs +31%) of the maximum execution time are much lower comparing *RTDS* against *Credit2*.

3.3.4 *Credit2* vs *RTDS* vs *null*: One-to-One Analysis

So far, we left alone *null* scheduler since it does not imply any scheduling actions but simply runs each vCPU on an isolated pCPU. Notably, factors considered for Credit2 and RTDS schedulers, such as *CPU affinity* and *scheduler parameters*, can not be applied for *null* scheduler. In this section, we take into account the *One-to-One* level of *CPU affinity* factor to make a fair comparison between *null*, *Credit2*, and *RTDS* schedulers. As in the previous section, we also considered the *stress* factor configured with the *CPU* level. In this case, we can not rely on the ANOVA Welch test since the analysis of the residuals, via the Shapiro-Wilk test, shows that they are not normally distributed ($W = 0.69$ and p-value < 0.0001). According to the Levene test, the residuals are heteroscedastic for the *stress* factor and homoscedastic for the *host-level scheduler* factor. Thus,

(a) Means (M) and standard deviations (SD) of maximum execution time replica VMs. *Stress* factor disabled is highlighted with blue (□), while *stress* factor enabled (CPU level) is highlighted with red (■).

Scheduler	*Credit2*	*RTDS*	*null*
M [μs]	42.45	42.29	42.60
	71.03	71.22	69.03
SD [μs]	3.31	3.33	2.77
	5.04	4.89	3.85



(b) Boxplots depicting the maximum execution times of replica VMs.

Figure 3.7: Analysis of *"Credit2"* and *"RTDS"* host-level scheduler factors, fixing the *CPU affinity* factor to the *One-to-One* level, and by enabling/disabling *stress* factor, CPU level.

to verify whether the distribution of data, by varying schedulers, belongs to the same population or not, we leverage the Kruskal-Wallis test. The test proves, with a high level of confidence (p-value < 0.0001), that the data distributions, when the disturbance factor varies, belong to different populations, meanwhile, when the scheduler factor varies, they belong to the same population.

The data reported in Figure 3.7a and 3.7b confirm that all the schedulers behave similarly when we avoid any sharing of physical CPUs among virtual CPUs (i.e., the *One-to-One* scenario). However, what stands out is that even for *One-to-One* scenario, the *stress DomU*, configured with

a CPU stressor, leads to statistically significant delays in maximum execution times for replicas.

Credit2 vs *RTDS* vs *null*: One-to-One Analysis

★ Considering the ***null* scheduler**, which is designed to statically assign every vCPU to a pCPU, and **no pCPUs sharing** between VMs, **compared to *Credit2* and *RTDS*, we obtain no significant difference** on both means and standard deviations of maximum execution time, even enabling external CPU-related disturbances.

3.3.5 *null* Scheduler Stress Analysis: DomU vs Dom0

To further analyze the stress impact on our testbed, we decided to fix the *host-level scheduler* to *null* scheduler, as the best scheduler for real-time by design, and perform several stress tests by varying levels (i.e., *stress-ng* stressors) of the *stress* factor, as depicted in Table 3.2. For each stress level, we use all the stressors provided by *stress-ng* application belonging to it. For example, to stress the CPU, *stress-ng* computes the Fibonacci sequence, finds factorials from 1 to 150 using Stirling's and Ramanujan's approximations, solves a 21 disc Towers of Hanoi stack using the recursive solution and so on. In the following, we compare both replicas and voter behaviors.

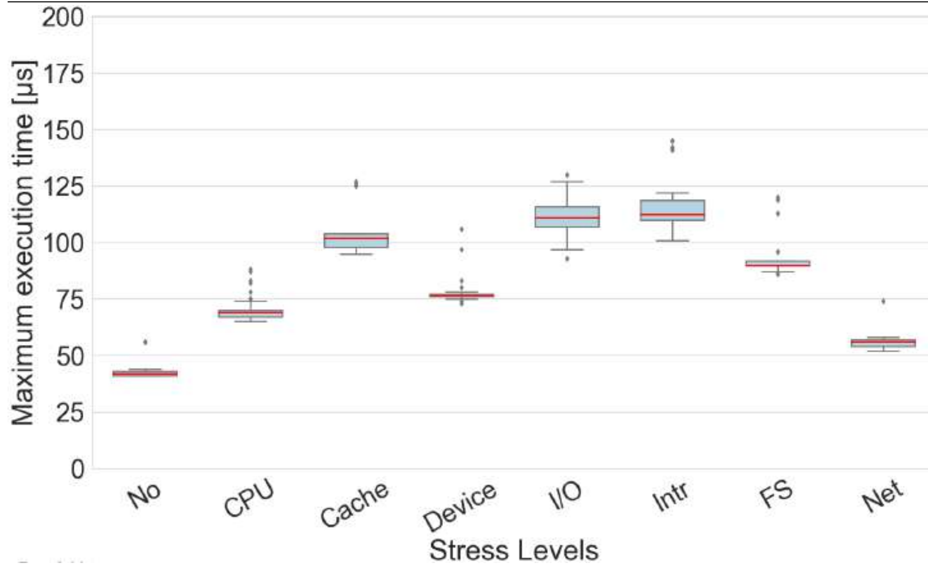
Replicas (DomU)

Also in these experiments, the residuals are not normally distributed according to the Shapiro-Wilk test ($W = 0.60$ and p-value < 0.0001), and they are heteroscedastic according to the Levene test. Therefore, we use the Kruskal-Wallis test, which confirms with a high level of confidence that all stress levels lead to distributions of data that are statistically different, i.e., the type of stress is a significant factor for our response variable.

Data reported in Figure 3.8a and 3.8b show that the presence of a stress VM, running on a separated pCPU, cause non-negligible delays on the execution time of high-critical real-time VMs. However, we must also consider that in the case of not-shared physical CPUs, we can observe that all execution times even in the worst case are always under $150\mu s$, a time that is within the requirements of most critical applications.

(a) Means (M) and standard deviations (SD) of maximum execution time for replica VMs (DomUs).

Stress	*No*	*CPU*	*Cache*	*Device*	*I/O*	*Intr*	*FS*	*Net*
M [μs]	42.60	69.03	104.56	78.13	111.66	115.57	92.80	56.07
SD [μs]	2.77	3.86	9.88	6.71	8.37	10.28	8.60	3.66



(b) Boxplots depicting the maximum execution times of replica VMs.

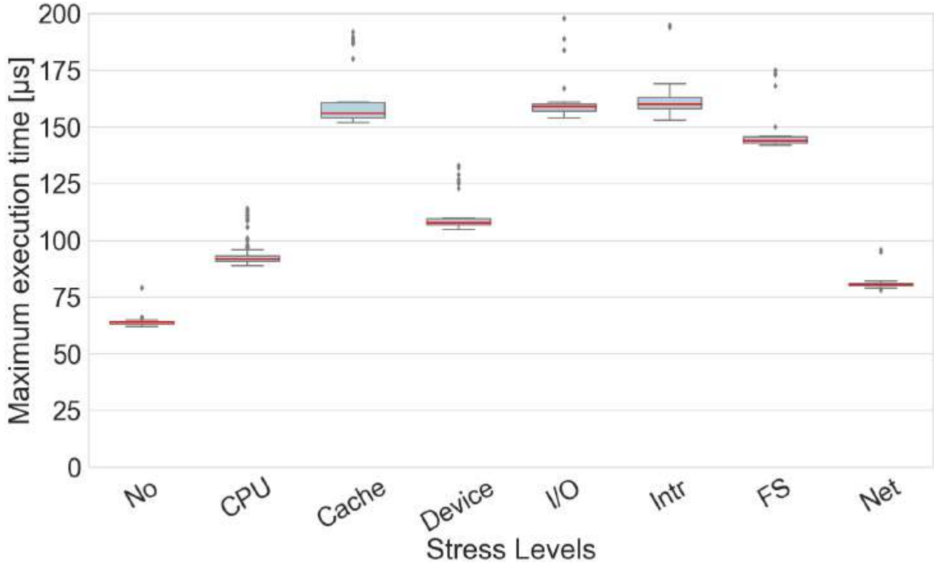
Figure 3.8: Analysis of *"null"* host-level scheduler factor on DomU, fixing the *CPU affinity* factor to the *One-to-One* level, and varying all levels within *stress* factor (CPU, Cache, Device, I/O, Intr, FS, Net).

null Scheduler Stress Analysis: DomU

★ Despite the ***null* scheduler** was designed to be the best choice for real-time scenarios, it still **leads to severe increase against a different kind of stress** in the means and standard deviation of maximum execution time, i.e., +31.62% and +32.13% in the best case (socket stressors), and +162% and +271% in the worst case (high interrupt load).

(a) Means (M) and standard deviations (SD) of maximum execution time for voter VM (Dom0).

Stress	*No*	*CPU*	*Cache*	*Device*	*I/O*	*Intr*	*FS*	*Net*
M [μs]	64.32	94.19	162.60	112.20	161.42	162.61	147.90	81.46
SD [μs]	2.96	5.93	14.17	9.08	10.38	9.64	9.87	3.89



(b) Boxplots depicting the maximum execution times of voter VM (Dom0).

Figure 3.9: Analysis of *"null"* host-level scheduler factor on Dom0, fixing the *CPU affinity* factor to the *One-to-One* level, and varying all levels within *stress* factor (CPU, Cache, Device, I/O, Intr, FS, Net).

Voter (Dom0)

The results provided previously focused on replicas and their maximum execution time distributions. As specified in Section 3.2.2, the voter runs on *Dom0* for easy testing and inducing a realistic disturbance on the target testbed. However, we also analyzed the maximum execution time distribution for voter VM (Dom0) by fixing the host-level scheduler to *null* and by varying the stress factor across all the target levels. As for experiments in 3.3.5, we can state that applying all stress levels leads to distributions of data that are statistically different with a high level of confidence. Further,

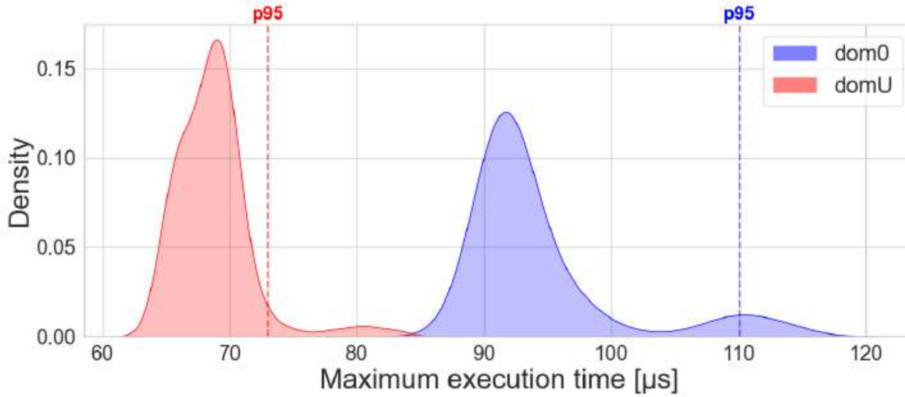


Figure 3.10: Distribution of maximum execution times for voter(dom0) and replica(domU) by fixing *host-level scheduler* factor to *null* level, the *CPU affinity* factor to the *One-to-One* level, and the *stress* factor to *CPU*.

Table 3.9a and Table 3.9b show the results obtained. Compared to what was obtained for replicas, it is clear that running applications on *Dom0* heavily influences predictability, with higher standard deviation variation compared to no stress. Furthermore, looking at 3.10, the tail distribution of the maximum execution times in *domU* is significantly smaller than in *dom0*, making it more suitable for real-time applications. This is due to critical operations that *Dom0* performs in tandem with the Xen hypervisor. As mentioned in previous sections, the voter should be implemented in an ad-hoc component, leaving *Dom0* with no applications running on top. Actually, *Dom0* is well-known to be one of the weakest points in terms of predictability of the Xen hypervisor, as it includes all the features for handling paravirtualized devices (i.e., PV drivers) and VMs lifecycle. For this reason, current works by *Xen FuSa Special Interest Group* [88, 89] are focusing on making Xen a fully static partitioned hypervisor for resource-constrained embedded systems and mixed-criticality systems. involves the complete removal of *Dom0* to speed up VMs' boot, simplify the certification process, and reduce the overall complexity of Xen-based systems.

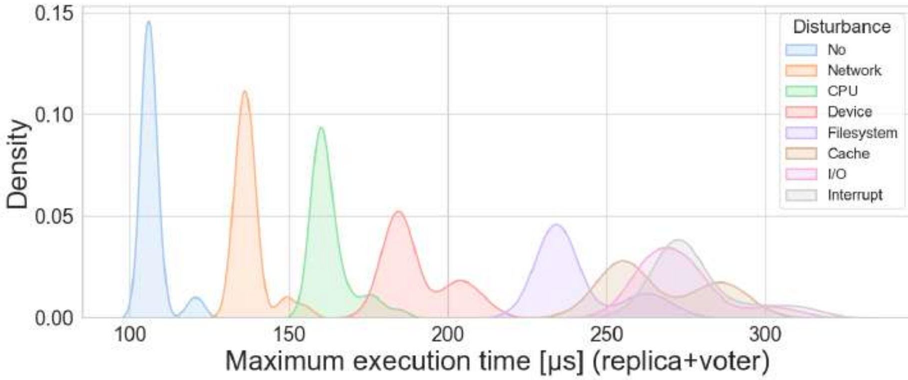


Figure 3.11: Distribution of maximum execution times for the entire execution (replica + voter) by fixing *host-level scheduler* factor to *null* level, the *CPU affinity* factor to the *One-to-One* level, and varying all levels within *stress* factor.

null Scheduler Stress Analysis: Dom0

★ ***Dom0* heavily influences the predictability of the software running on it compared to *DomUs***. This is due to the critical operations *Dom0* performs in tandem with Xen hypervisor (i.e., PV drivers). *CPU instruction and/or data caches* stress the most impactful, with an increase in the standard deviation of the maximum execution time of +379%. Additionally, *Dom0* exhibits a larger tail distribution compared to *DomU*.

Aggregated Response Time Analysis

Up to this point, our analysis has focused on examining the temporal isolation of individual replicas and the voter in isolation. However, the most crucial metric for the application under analysis is the aggregated response time. This encompasses the duration from when the replicas initiate computation to when the voter renders its final evaluation. To conclude our analysis, we present in 3.11 the distributions of WCET for the aggregated response times. These are computed as the sum of the voter's response time and the maximum execution time among the two replicas. Each distribution corresponds to a specific type of disturbance applied using stress-ng in the other DomU. Our results reveal that de-

Table 3.3: Means (M) and standard deviations (SD) percentage increase, comparing *One-to-One* level to *Many-to-Many* level. SD best case and worst case are highlighted in green and red, respectively.

Rate limit [μs] **(Credit2)**	*100*	*500*	*1000*	*5000*
M [μs]	+274%	+300%	+288%	+134%
SD [μs]	+654%	+660%	**+801%**	**+376%**
Budget/Period [μs] **(RTDS)**	*1k/5k*	*3k/9k*	*5k/10k*	*10k/10k*
M [μs]	+11%	+51%	+65%	+42%
SD [μs]	**-16%**	+8%	+142%	**+185%**

Table 3.4: Means (M) and standard deviations (SD) percentage increase, comparing *No Stress* level to *CPU Stress* level. SD best case and worst case are highlighted in green and red, respectively.

Scheduler [Many-to-Many]	*Credit2*	*RTDS*	
M [μs]	+42%	+31%	
SD [μs]	**+30%**	**+0%**	
Scheduler [One-to-One]	*Credit2*	*RTDS*	*null*
M [μs]	+67%	+68%	+62%
SD [μs]	**+52%**	+47%	**+39%**

spite the utilization of a real-time scheduler, the inherent complexities of a hypervisor such as Xen introduce a certain level of interference when the system is under stress. However, this interference can be thoroughly considered and analyzed to provide a realistic estimation of WCET for mixed-criticality applications.

3.3.6 Results and Lessons Learned

Tables 3.3, 3.4, and 3.5 summarize results obtained, and highlight the percentage increases caused by factor variations in the analysis shown so far. The experimental analysis shows us a clear and statistically significant spectrum of Xen's behaviors across several configurations for building mixed-criticality systems on ARM-based architectures. However, the virtualized software operations heavily depend on whether the hypervisor allows or not the sharing of physical CPUs. Further, the fewer physical CPUs shared, the better the time isolation (i.e., lower means and

Table 3.5: Means (*M*) and standard deviations (*SD*) percentage increase, comparing *No Stress* level by fixing the *host-level scheduler* factor to *null* scheduler. SD best case and worst case are highlighted in green and red, respectively.

Stress Levels: [DomU]	*CPU*	*Cache*	*Device*	*I/O*	*Intr*	*FS*	*Net*
M [μs]	+62%	+145%	+83%	+162%	+171%	+118%	+32%
SD [μs]	+39%	+257%	+142%	+202%	**+271%**	+210%	**+32%**
Stress Levels: [Dom0]	*CPU*	*Cache*	*Device*	*I/O*	*Intr*	*FS*	*Net*
M [μs]	+46%	+152%	+74%	+151%	+152%	+130%	+27%
SD [μs]	+100%	**+379%**	+207%	+251%	+226%	+233%	**+31%**

standard deviations). However, this does not mean that temporal isolation is assured if a partitioning approach is used, i.e., running virtual CPUs on dedicated physical CPUs. This is because sharing resources, such as I/O and memory (LLC), still causes uncontrolled latency under the hood.

A partitioning approach (see One-to-One and *null* scheduler experiments in 3.3) might be a good solution for high-criticality VMs. However, this approach obliges knowing in advance vCPU/pCPU affinity for all VMs, which is not the case if many less critical VMs need to be handled. An optimal solution would still use less stringent scheduling algorithms if possible (e.g., *Credit2*, *RTDS*) to optimize resource utilization across high and low-critical VMs, especially in embedded and resource-constrained environments.

The experimental results highlight that industry practitioners can identify optimal combinations of scheduler parameters for both *Credit2* and *RTDS* schedulers through the systematic assessment approach proposed in this study. This allows for effective management of Xen overhead, making it manageable across various industrial applications.

Additional mechanisms are currently developed for the next versions of ARM-Xen, which promise to greatly reduce the hypervisor latency and overhead revealed in this analysis, at the cost of increasing the complexity of configuring Xen and reducing resource sharing. In particular:

- **Cache Coloring:** a static partitioning of LLC for VMs to avoid cache misses induced by running many VMs.
- **Dom0less:** Xen can be configured statically without the need for a

privileged VM (i.e., Dom0), which has full control over the hardware. This completely removes dependence on Linux to run Dom0, and speeds up the boot of guests (DomUs);

- **Static memory communication:** it allows i) static allocation of shared memory between VMs, ii) static event channels without the need for complex communication drivers (e.g., xenbus [104]);

- **CPU-pools:** static group division of pCPUs that enables the usage of different host-level schedulers (e.g., *null*, *RTDS*) simultaneously for different groups of VMs.

Other studies in the literature proposed novel mechanisms to increase the temporal isolation in hypervisor-based environments (e.g., Mempol [51], Memguard [50], I/OGuard [102], RPUGuard [8]), but unfortunately they are not currently integrated into the mainline Xen.

The experimental analysis outlined in this section demonstrates how temporal isolation can be evaluated and optimized in virtualized environments. The lessons learned from these experiments highlight the need for precise control over resource sharing to ensure predictable real-time performance, especially in complex safety-critical applications like those found in nuclear fusion systems.

While these results provide valuable insights into the behavior of virtualized systems, they also underline the necessity for a more systematic approach to deploying MCSs in distributed environments. This is particularly relevant in industrial settings such as nuclear fusion, where multiple criticality levels must be managed simultaneously across both edge and cloud environments (see Section 1.2).

To address these challenges, the following section introduces a model-based approach for the deployment of MCSs within the context of edge/fog computing. This model is designed to efficiently allocate and manage tasks on virtualized MPSoCs and remote servers, ensuring that critical and non-critical applications meet their performance requirements while communicating seamlessly across the edge-cloud continuum.

3.4 Mixed-Criticality Deployment Model

This section proposes a standardized deployment model for MCSs on distributed systems utilizing virtualized MPSoCs and servers within a Cloud-to-Thing continuum environment.

Our model describes the deployment of MCSs as architectural scenarios, which are quadruplets (ENV_L, ENV_R, SS, CS) where ENV_L is the local environment, ENV_R is the remote environment, SS is the scheduling scheme, i.e., the scheduling algorithms used to schedule VMs and tasks within VMs, and CS is the inter-VM communication scheme, i.e., the communication channels VMs use to communicate with each other.

Please note that from now on, VMs are considered as sets of tasks, where each VM isolates tasks from other VMs.

The aforementioned quadruplet is obtained by a mapping function AS_MAP that combines tasks, computational resources, scheduling algorithms, and inter-VM communication channels together, configuring architectural scenarios. Figure 3.12 collects all the elements here discussed; the way they are arranged together depends on the AS_MAP function, which configures the virtualization layer and the interconnection among tasks, I/O devices, and data repositories.

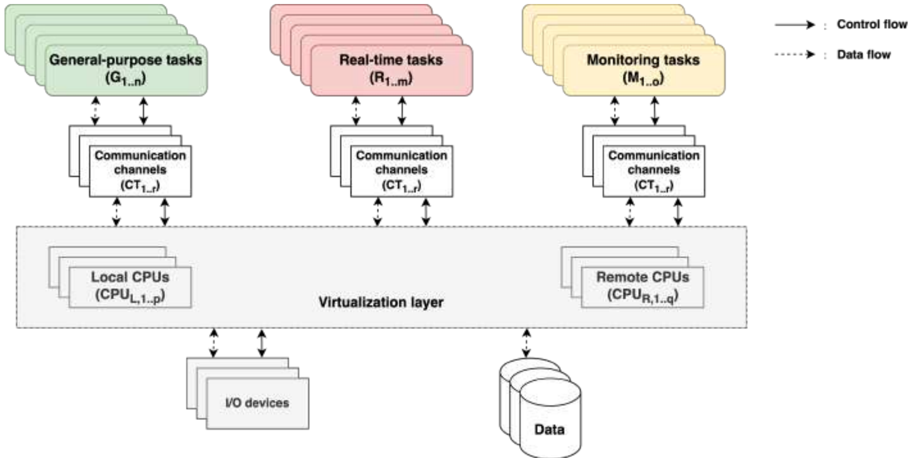


Figure 3.12: The elements that the proposed MCS deployment model handles.

Global Task Set. The global task set (TS) is the set of all tasks of an MCS:

$$TS = RT \cup M \cup G\,,$$

where RT is the set of real-time tasks, G is the set of all other general-purpose tasks, and M is the set of monitoring tasks. Monitoring tasks are considered part of the set because they play a crucial role in distributed critical scenarios, such as nuclear fusion. These tasks are more important than general-purpose ones but not as critical as the real-time controllers.

Local and Remote Environments. Our model splits MCSs into two environments: the local (ENV_L) and remote (ENV_R) environments, where the ENV_L environment is a fog node with substantial hardware resources and virtualization support (e.g., an MPSoC), and the ENV_R environment is the cloud.

The set of computational resources (CR) groups Processing Elements (PEs) from both environments:

$$CR = \mathrm{PE}_L \cup \mathrm{PE}_R\,,$$

where a PE is a generic processing unit that MPSoCs ship with, such as CPUs, RPU,GPUs, FPGAs, and NPUs. PE_L and PE_R are the sets of local and remote PEs, respectively.

$$\mathrm{VM}_L \subseteq \mathrm{VM}_G, \mathrm{VM}_R \subseteq \mathrm{VM}_G\,,$$

where $\mathrm{VM}_L \cup$ $\mathrm{VM}_R =$ VM_G, i.e. the union of local and remote VMs is the global set of VMs

In order to fully define environments, we consider PE pools. A PE pool (P) is an element of power sets of either PE_L or PE_R ($P \in P(\mathrm{PE}_L) \cup P(\mathrm{PE}_R)$), and represents a group of physical PEs isolated from other PEs ($P_i \cap P_j = \emptyset, i \neq j$).

Given $\mathrm{PE}_L \subset P(\ \mathrm{PE}_L) and\ \mathrm{PE}_R \subset P(\ \mathrm{PE}_R)$) the two sets of local and remote PE pools (i.e., the two local and remote pooling schemes), ENV_L and ENV_R are defined as follows:

$$ENV_L = \{DM_{i,L} = (\mathrm{VM}, P) \in \mathrm{VM}_L \times PS_L\}\,,$$

$$ENV_R = \{DM_{i,R} = (\mathrm{VM}, P) \in \mathrm{VM}_R \times PS_R\},$$

where $DM_{i,R}$ is the i-th deployment module of the x environment. If $\|ENV_x\| = 1$, the environment x is made of only one VM and one PE pool, i.e., environment x is non-virtualized.

We here note that during evaluation (Section 5.1) we consider CPUs as the only PEs. Therefore, in the following we consider the set of computational resources as the collection of local and remote CPUs, i.e. $CR = \mathrm{CPU}_L \cup \mathrm{CPU}_R$.

Scheduling. Concerning both tasks within VMs and VMs themselves, we split scheduling algorithms according to the hierarchical scheduling taxonomy [105, 106].

Thus, we split scheduling algorithms into two categories: local scheduling algorithms (S_L) for tasks within VMs, and, global scheduling algorithms (S_G) for VMs assigned to a given CPU pool.

The scheduling scheme SS_{DM} for a given deployment module ($DM \in ENV_L$) is a pair $(S_{L_{DM}}, S_{G_{DM}})$, where $S_{L_{DM}}$ and $S_{G_{DM}}$ are the local and global schedulers linked to DM, respectively. The set of all scheduling schemes of all deployment modules is labeled as SS.

Defining the scheduler at deploying time is important for the designer since the choice affects the number of deployable VMs. If a partitioning hypervisor is used, the global scheduling is null and the number of VMs is limited by the number of CPUs. On the other hand, the number of VMs can be higher than the number of CPUs with a traditional scheduler.

Starting from the generated deployment model, the designer can later employ an analytical timing analysis method that aligns with the scheduling taxonomy that has been chosen during the deployment phase (e.g., hierarchical scheduling [105]).

Inter-VM Communication. The inter-VM communication scheme CS is a set of triples defined as follows:

$$CS = \{(CC, \mathrm{VM}_1, \mathrm{VM}_2) \in CC \times (\mathrm{VM}_L \cup \mathrm{VM}_R) \times (\mathrm{VM}_L \cup \mathrm{VM}_R)\},$$

where CC is the set of communication channels, such as hypervisor-managed shared memory or network sockets.

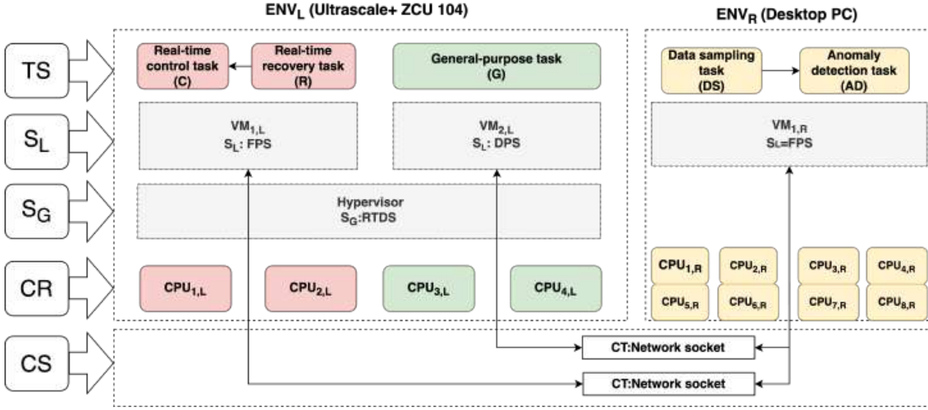


Figure 3.13: A sample architectural scenario.

Mapping Function. The mapping function AS_MAP is defined as follows:

$$AS = AS_MAP(TS, CR, S_L, S_G, CC)\,,$$

where $AS = (ENV_L, ENV_R, SS, CS)$, i.e., the architectural scenario, which collects the: local and remote environments (ENV_L and ENV_R); scheduling schemes (SS); and inter-VM communication scheme (CS).

In order to clarify the application of AS_MAP, we depict in Figure 3.13 a sample architectural scenario commonly found in distributed applications and described through the MCS deployment model. In this case, the elements are:

- TS: one general purpose task (G), two real-time tasks (R and C), and two monitoring tasks (DS and AD).
- CR: Local CPUs of the ZCU 104 MPSoC and remote CPUs of a Desktop PC.
- S_L and S_G: the local schedulers of VMs deployed on top of the hypervisor and its global schedulers to orchestrate such VMs to handle their access to shared hardware resources, and the local schedulers of an Ubuntu-based OS.
- CC: Network sockets.

Table 3.6: Local and remote VMs, CPU pools, DMs, and scheduling schemes of the sample architectural scenario.

Local environment			
$\boldsymbol{VM_L}$	$\boldsymbol{P_L}$	$\boldsymbol{DM_L}$	$\boldsymbol{SS_{DM_L}}$
$1, L : \{R, C\}$ $2, L : \{G\}$	$1, L : \bigcup_{i=1...p} CPU_{i,L}$	$1, L : (VM_{1,L}, P_{1,L})$ $2, L : (VM_{2,L}, P_{1,L})$	$DM_{1,L} : (RTDS, DPS)$ $DM_{2,L} : (RTDS, FPS)$

Remote environment			
$\boldsymbol{VM_R}$	$\boldsymbol{P_R}$	$\boldsymbol{DM_R}$	$\boldsymbol{SS_{DM_R}}$
$1, R : \{DS, AD\}$	$1, R : \bigcup_{i=1...p} CPU_{i,R}$	$1, R : (VM_{1,R}, P_{1,R})$	$DM_{1,R} : (-, FPS)$

Table 3.7: The deployment quadruplet of the sample architectural scenario.

Deployment quadruplet			
$\boldsymbol{ENV_L}$	$\boldsymbol{ENV_R}$	$\boldsymbol{SS}$	$\boldsymbol{CS}$
$\{DM_{1,L}, DM_{2,L}\}$	$DM_{1,R}$	$\{SS_{DM_{1,L}}, SS_{DM_{2,L}}, SS_{DM_{1,R}}\}$	$\{(Socket, VM_{1,L}, VM_{1,R})\}$

The way these elements are laid out by AS_MAP is described in Tables 3.6 and 3.7, which collect local and remote VMs (VM$_L$ and VM$_R$), local and remote CPU pools (P_L and P_R), local and remote deployment modules (DM_L and DM_R), the scheduling scheme of each local and remote deployment module (SS_{DM}), and the corresponding quadrupled.

Given our objective to deploy MCSs in safety-critical environments, and the takeaways of this chapter, for the next chapter we chose to focus on partitioning hypervisors that already incorporate advanced techniques for improving isolation. In the next chapter, we will introduce the Omnivisor model, which addresses the limitations of current hypervisors in managing these challenges in heterogeneous multi-core systems. For our reference implementation, we decided to start with the Jailhouse hypervisor. Unlike Xen, Jailhouse already implements advanced techniques such as cache coloring and memory bandwidth management using both software (Memguard) and hardware (QoS) mechanisms. This foundation will allow us to build and demonstrate the efficacy of the Omnivisor model in ensuring robust isolation for mixed-criticality systems.

4 The Omnivisor: A Unified Approach to Virtualizing Heterogeneous MPSoCs

In this chapter, we introduce the Omnivisor, a novel hypervisor model that generalizes static partitioning hypervisors to enable the transparent execution of VMs on heterogeneous cores over COTS MPSoCs. The model aims to streamline the deployment process and simplify the programming model of such complex architectures while providing strong spatial and temporal isolation as required by mixed-criticality systems.

4.1 Unified Model for Heterogeneous Virtualization

We introduce a cohesive virtualization solution that integrates multiple technologies designed for managing heterogeneous MPSoCs. Central to this solution is the Omnivisor model, which incorporates key technologies to tackle virtualization challenges in mixed-criticality systems.

The Omnivisor model is built around a core technology that enables the hypervisor to manage remote cores as resources as explained in Section 4.2. This allows the hypervisor to start VMs on these remote cores while ensuring isolation over heterogeneous cores. This core technology is essential for providing a unified management interface and ensuring that all cores, regardless of their type, can be efficiently utilized without compromising system integrity or performance.

A critical component of the Omnivisor is the RPUGuard technology described in Section 4.3. RPUGuard is designed to ensure secure and reliable communication between multiple VMs running on APUs and those running on RPUs. This technology is crucial for maintaining isolation and preventing interference, which is particularly challenging in heterogeneous environments where different types of processing units must interact seamlessly.

Additionally, looking beyond current COTS boards, in Section 4.4 we introduce a method to implement lightweight and predictable virtual memory on microcontroller-based processors within MPSoCs. This advancement is aimed at enhancing the future capabilities of the Omnivisor, allowing it to provide robust virtualization support even on microcontroller-based processors, which traditionally lack sophisticated memory management features.

Finally, to further improve the usability of this technology we have integrated the Omnivisor in an orchestration system. Specifically, we introduce RunPHI as a solution to deploy high-criticality containers on heterogeneous platforms.

Through these integrated technologies, the Omnivisor model offers a comprehensive solution for managing heterogeneous MPSoCs, ensuring both performance and isolation in mixed-criticality systems.

4.2 The Omnivisor Architectural Design

As depicted in Figure 4.1, while conventional hypervisors are designed to manage microprocessor-level CPUs, the Omnivisor model extends its control to include microcontroller-level CPUs and soft-cores on programmable logic (FPGA). To achieve this, the Omnivisor assumes control over different hardware mechanisms to ensure isolation, both temporally and spa-

tially, of the VMs. Three primary objectives underpin the Omnivisor model:

- **1.** To offer users a consistent, transparent, and easy-to-use interface for managing virtual machines on both primary and remote cores.
- **2.** To reorganize the privilege levels of the software running on heterogeneous cores in order to build a holistic privilege hierarchy across the platform.
- **3.** To seamlessly administer spatial and temporal isolation between virtual machines, regardless of the specific core on which they are deployed.

According to this novel model, remote cores are no longer mere I/O devices; instead, they are elevated to primary CPUs capable of running self-contained, strongly isolated VMs.

Clarification of Terminology. Before delving into the specifics of the Omnivisor, it is important to clarify why we chose to use the term *"Virtual Machine"* to denote the code executed by the Omnivisor on all the types of cores. We acknowledge that the code running on remote cores does not execute atop an actual hypervisor, meaning that there is no scheduler, and the code has complete control over the core itself. However, we have opted to label them VM for two main reasons. First, they are encapsulated by the Omnivisor, which is capable of isolating the accessible resources in the system, similar to how SPHs handle traditional VMs. Second, we provide users with a unified and transparent method for utilizing remote cores, mirroring the process of launching a VM on application cores.

4.2.1 Requirements Responsibilities and Features

Requirements

The Omnivisor model is based on the assumption of having at its disposal a fully featured MPSoC with the following characteristics:

- *Multiple Core Clusters*: Two or more heterogeneous clusters of cores, and at least one of the clusters is a multiprocessor-level CPU cluster.

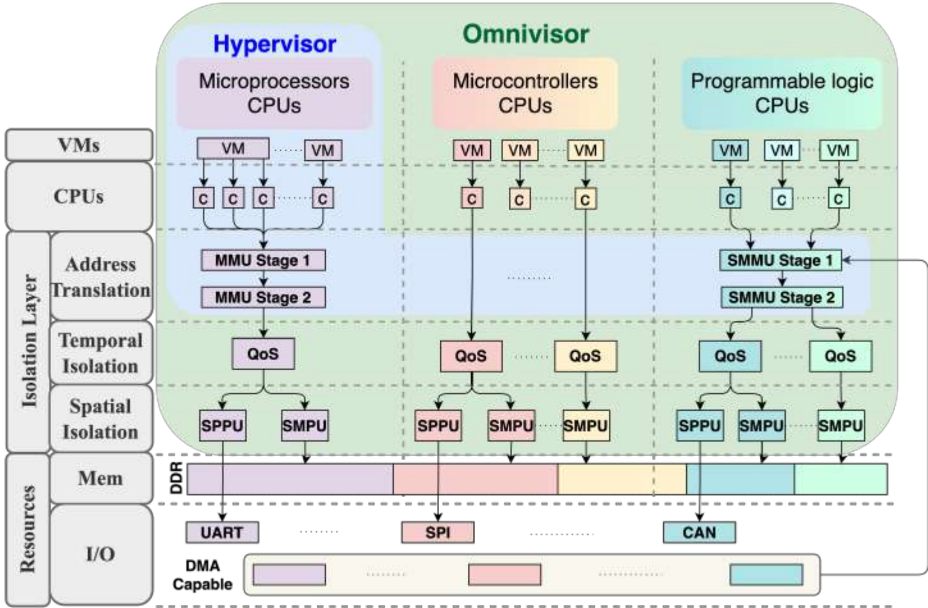


Figure 4.1: A block diagram illustrates the Omnivisor model, showcasing varied temporal and spatial isolation mechanisms across CPU clusters, emphasizing their heterogeneity. The arrows indicate the flow of a request from an initiator to the accessed resource (memory or I/O).

- *Address Translation*: An MMU featuring two levels of translation in front of each multiprocessor-level CPU cluster and an SMMU placed between DMA-capable peripherals/accelerators and shared resources.
- *Accesses Protection*: SMPU/SPPU hardware protection mechanisms to shield shared resources (memory, system registers, and peripherals).
- *Bandwidth Regulation*: A hardware QoS-like bandwidth allocation mechanism for each core cluster and DMA-capable peripherals that access shared resources.
- *Power Management Firmware*: A board-specific firmware that exposes an interface to the hypervisor for power management of cores.

These specified characteristics represent the foundational prerequisites for a platform to be deemed *Omnivisor-ready*. Although these requirements may initially appear as limiting factors, they effectively align with the design standards of modern embedded system platforms [107, 108, 109], tailored to meet industrial demands.

Responsibilities

The Omnivisor operates as holistic software running at the highest privilege level on the board. It delivers services to software running at lower privilege levels. Therefore, its primary responsibility is to prevent the escalation of VM privileges, regardless of the used cores.

AMP privilege enforcement. The coexistence of multiple cores with varying architectures in an MPSoC precludes the application of a Symmetric Multi-Processing (SMP) approach. In SMP, all cores are orchestrated by a single software instance sharing a common address space. Instead, MPSoCs imply the use of an Asymmetric Multi-Processing (AMP) approach, where different core clusters operate independently, each with a unique address space. Given that constraint, similar to traditional hypervisors in SMP configuration, the Omnivisor must ensure that VMs running in AMP configuration do not access resources outside of the boundaries of the partitions. However, while traditional hypervisors can leverage multi-core hardware extensions to manage the privilege levels of VMs, the Omnivisor must employ a combination of distinct hardware mechanisms tailored to the specific core cluster it is managing. For instance, while soft-cores deployed on FPGA can be protected using the SMMU, the Omnivisor must leverage SMPUs to shield the resources from the VMs running on microcontroller-level CPUs. Consequently, as shown in Figure 4.2 (right), every time a new VM is launched on a remote core, before starting the core, the Omnivisor must configure an isolation layer that restrains the capabilities of the newly-created VM restricting access to both higher privileged resources (e.g., system registers) as well as resources belonging to different VMs (e.g., I/O peripherals, memory regions). The arrows in Figure 4.2 (right) are color-coded based on the operation and are enumerated in temporal order. Specifically, when the user requests to run a new VM on a remote core, the Ominvisor internally parses the configuration file

describing the VM, loads the code in the memory, programs the hardware protection mechanisms (isolation layer), and starts the remote core.

DMA-capable I/O: The cores are not the only platform managers within the system. Indeed, DMA engines could have access to all system resources, potentially jeopardizing inter-partition spatio-temporal isolation. To address this risk, the Omnivisor must prevent:

- **1.** DMA engines from having unrestricted and unregulated access to memory resources.
- **2.** A core from programming the DMA to access memory regions it does not own.

As depicted in Figure 4.1, the Omnivisor addresses the first issue by employing SMMU mechanisms to enforce address translation and access protection for DMA, much like traditional SPHs. Additionally, the QoS is employed to provide temporal isolation.

Typically, when an SPH allocates the DMA to a VM, it configures the SMMU to allocate the same memory regions to both. Therefore, a VM cannot exploit the DMA to access inaccessible regions. However, if a second virtual machine is running on a remote core without MMU/SMMU protection (microcontroller-level CPUs), it can freely access the address region of the DMA registers. Therefore, it could potentially program the DMA to gain unauthorized access to memory areas belonging to the first VM. To avoid that, addressing the second issue, the Omnivisor employs a strategy wherein SMPUs are configured to restrict access to the DMA registers exclusively to the Omnivisor itself and to the VM that is supposed to use it.

In a broader context, the Omnivisor applies a similar strategy to restrict permissions of remote cores to protect other critical address regions, including those for configuring the SMMU, SMPUs, and QoS.

Features

The Omnivisor provides a set of features that includes that of the traditional SPHs while expanding them to encompass heterogeneous processing elements (see Figure 4.2). Given the diversity among existing hypervisors,

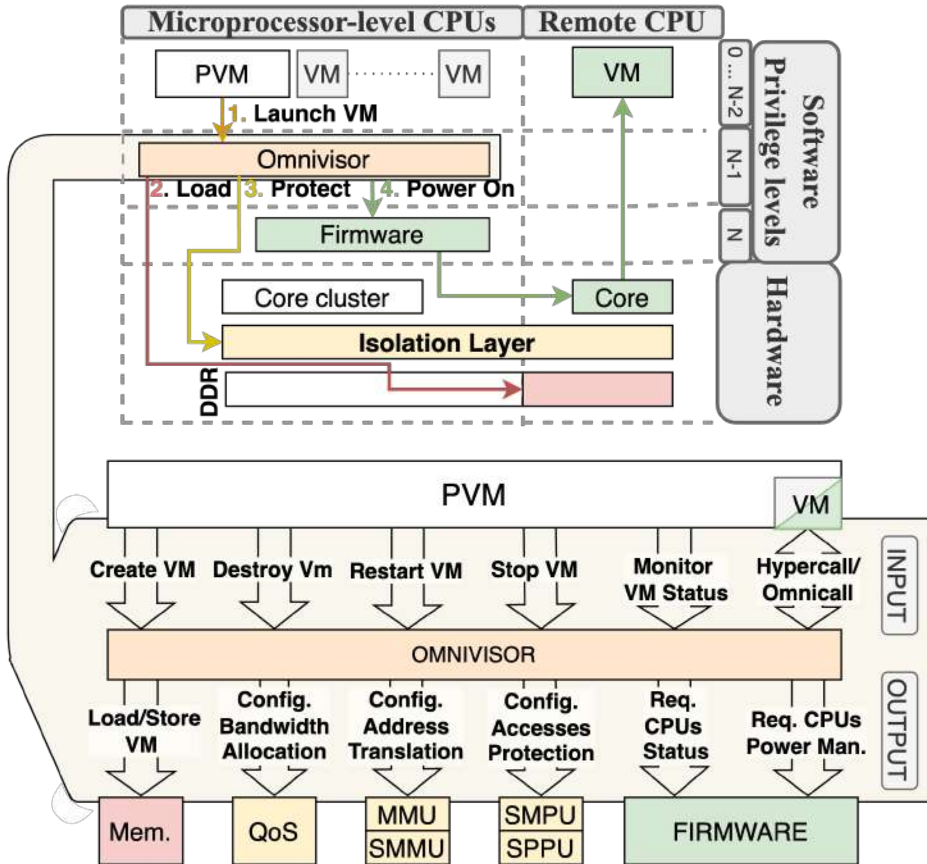


Figure 4.2: Omnivisor feature set (left) and remote core VM startup process (right).

defining the minimum feature set and how they are extended for effective operation on asymmetric architectures is crucial.

The Privileged Virtual Machine (PVM) interface. First, we introduce the Privileged Virtual Machine (PVM), which is a known concept in hypervisor's literature [110], and is the only VM with the ability to manage other VMs. A few examples are the *root-cell* in Jailhouse [29], and the *Dom0* in Xen [111]. The Omnivisor provides the PVM with the

same interface for managing VMs for both the main and remote cores. For instance, as shown in Figure 4.2, the PVM only needs to request the VM launch, and then the Omnivisor takes charge of programming the underlying resources to serve the request for the specified processor. Other than launching a VM, the Omnivisor provides methods for stopping and restarting a VM and an interface for monitoring the current status of the VMs.

Omnicall. Most state-of-the-art hypervisors implement *hypercalls* to expose functionalities to virtual machines, akin to how operating systems implement system calls for processes. Despite the current implementation of Omnivisor restricting this mechanism to virtual machines running on the APUs, we aim to propose a design for extending this service to VMs running on remote cores, which we will refer to as *"Omnicalls"*. To implement this mechanism, the Omnivisor needs to provide three additional features:

- **1.** Event signaling from the Omnivisor to VMs on remote cores.
- **2.** Event signaling from VMs on remote cores to the Omnivisor.
- **3.** A real-time protocol for inter-VM communication.

For the first functionality, we need to differentiate between processing elements that support interrupt delivery, like APUs, and those that do not support them, such as hardly restricted soft-cores. To signal an event to the former category, the Omnivisor can leverage Software Generated Interrupt (SGI). Meanwhile, signaling events to the latter requires the remote VM to periodically check for Omnivisor-originated pending events (polling).

Regarding the second functionality, the Omnivisor can grant the VMs on remote cores access to a subset of the interrupt controller's configuration space, enabling the generation of SGIs toward the cores where the Omnivisor operates. Currently, the Omnivisor supports restricted access to the interrupt controller configuration space for these VMs.

Lastly, using shared memory for data exchange is already implemented in most legacy SPHs. We extended this feature to remote cores in the Omnivisor, but enhancing the real-time performance of the communications

requires a tailored mechanism. To provide real-time guarantees, one existing solution consists of using an external processing element as a broker to orchestrate the communications between VMs. This has been theoretically proved and tested on a heterogeneous MPSoC by Schwäricke et al. [112], and the Omnivisor can easily integrate the broker as a VM running on a remote core while using its features to isolate it both temporally and spatially from the other VMs.

Dynamic Address Translation. In traditional hypervisors, when a new VM is created on the APU, address translation is typically implemented using the MMU. The Omnivisor extends this functionality to soft-cores by utilizing the SMMU. It's worth noting that the SMMU is already employed by SPHs to perform address translation for I/O devices associated with VMs. However, the Omnivisor changes the perspective and utilizes the same mechanism to implement self-contained translation specifically for soft-cores, which are treated as self-contained VMs in this context. This approach unfortunately cannot be employed on microcontroller-level CPUs which usually do not present any mechanism for address translation. However, there is a trend for the deployment of virtualization support over microcontroller-level CPUs which can be exploited in future MPSoCs to extend the translation even to these cores as will be discussed in Section 4.4 [113] [114] [115].

Dynamic Accesses Protection. Protection mechanisms on MPSoCs, such as SMPU/SPPU, are commonly configured statically at boot time by high-privilege and secure software (e.g., first-stage bootloader). These configurations typically remain unchanged throughout the system's lifetime. However, to enable the seamless execution of isolated VMs on remote cores, the Omnivisor dynamically determines how to configure all access protection mechanisms. This approach ensures dynamic system-level protection that adapts during runtime based on the specific VMs currently active.

Dynamic Bandwidth Allocation. Traditional SPHs ensure that resource assignments remain static between PVM management calls. This implies that everything can be dynamically reassigned by these calls, re-

maining static until the next call. The Omnivisor maintains consistency by applying the same approach to bandwidth allocation. Hence, every time a new VM is launched, it is possible to dynamically allocate the bandwidth to that VM. Moreover, to enable mission-critical reconfiguration scenarios and ease parameter tuning, the Ominivisor implements bandwidth allocation as a settling call that the user can leverage to modify the temporal behavior of the VMs to a new static configuration. Once more, the Omnivisor shifts the paradigm regarding resource utilization. Unlike SPHs, which primarily focus on protecting VMs solely on the APU, the Omnivisor extends its scope to encompass VMs on other remote processors. Consequently, bandwidth regulation mechanisms like QoS are not only employed on accelerators to maintain service quality for APUs but also for remote cores, even if they are soft-core deployed on FPGA.

4.2.2 Implementation Strategy

The Omnivisor model is designed to apply to a wide range of existing partitioning hypervisors; nonetheless, our reference implementation is built on top of the Jailhouse hypervisor [29] because it has low overhead [55] while maintaining an easy-to-use interface to manage VMs at runtime. Furthermore, the Jailhouse-RT branch, overseen by Minerva Systems [116], already implements MemGuard-like regulators for the APUs, page coloring, and basic SMMU drivers. It also provides a rudimental interface to control ARM QoS regulators.

The implementation was carried out with testing focused on the ARM-based Zynq Ultrascale+ board from Xilinx. This MPSoC aligns with all the requirements outlined in Section 4.2.1: it features a quad-core ARM Cortex-A53 (APUs), a dual-core ARM Cortex-R5F (RPUs), and a 16nm FinFET + Programmable Logic (FPGA). Additionally, the platform is equipped with protection mechanisms for both temporal isolation (QoS), address translation (MMU, SMMU), and access permissions (SMPUs, and SPPUs). From now on, we will refer to this platform with the *ZCU* notation. Moreover, to use the correct terminology, the SMPUs/SPPUs on the board are named XMPU and XPPU.

This section aims to illustrate key Omnivisor technical details, providing a comprehensive discussion of strengths and limitations. To achieve this, we first briefly describe the Jailhouse hypervisor, and the additional

functionalities introduced by the Omnivisor extension. Then, we walk through the compiling and start processes of a VM from the user's perspective while explaining how the Omnivisor manages the system under the hood.

Jailhouse in a Nutshell. A pivotal design choice in Jailhouse is to initiate the hypervisor from a running Linux instance. Specifically, by utilizing a Linux kernel module, users can load the hypervisor into memory and initiate a series of procedures to prepare the system. Upon initialization on each core, the hypervisor takes control of the underlying hardware, transforming the running Linux into the first virtual machine within the system, referred to as the *root-cell*. For its bootstrap, the hypervisor requires only a configuration file that lists the resources allocated to the *root-cell*. Next, to create reservations (*cells* in Jailhouse jargon) for the creation of additional VMs (*inmates*), the hypervisor reallocates hardware resources (e.g., CPU(s), memory, PCI or MMIO devices) from Linux to the new cells as detailed in other cell-specific configuration files. From now on, we will use the term *"VM"* to refer to the cell plus inmate pair and *"PVM"* to refer to the root-cell.

Omnivisor Extension Overview. Starting from a vanilla Jailhouse, besides the small modifications integrated all over the code to transparently unify the interface of Jailhouse with the new services, the Omnivisor extends the hypervisor with new low-level functionalities. First, the power management of remote cores has been implemented, encompassing shutdown, stop, and start functionalities for both microcontroller-level and soft-cores. Second, spatial isolation management has been enhanced to include dynamic control of XMPUs/XPPUs. Moreover, temporal isolation management has been refined through the integration of QoS regulator control. Finally, the compiling procedure for remote cores VMs has been integrated into the hypervisor offline workflow. The usage of these functionalities is detailed below.

4.2.3 Omnivisor Usage Workflow

One of the key objectives of the Omnivisor is to simplify the utilization of complex heterogeneous architectures for users. Therefore, the Om-

nivisor provides a unified approach for managing VMs on both main and remote cores. In our implementation, based on the ZCU, alongside the legacy APUs we have integrated all the necessary code to run VMs on two types of remote cores: RPUs (ARM32-CortexR5F) and RISC-V soft-cores (Pico32 [57]). To streamline our discussion, we will utilize the term *"rCPUs"* to refer to any remote core, while we will delve into the implementation for RPU and RISC-V cores only when required.

VM Compiling Process (Offline)

The initial step involves the user compiling a specific VM application to run on a remote core. The offline compiling procedure, along with its input and output, is depicted in Figure 4.3. Given the nature of the remote cores, the applications we run are either bare-metal or built on top of simple RTOSes. In both cases, linking some libraries may be a requirement for the code to work correctly on a specific core. For instance, the traditional compiling approach for RPUs on ZCU entails using Xilinx-provided libraries. To streamline the utilization of *rCPUs* and align with the Jailhouse methodology, we have integrated the libraries for compiling VMs targeting RPU and RISC-V cores into the Omnivisor code. Consequently, the user only needs to integrate the application-specific code into the Omnivisor code, as all the necessary libraries are already provided, similar to how Jailhouse includes libraries for compiling APU-based VMs. Additionally, the user must provide a configuration file for the VM, specifying the required resources. This configuration should include details on the core(s) used by the VM, whether they are main cores or remote cores, as well as information about memory regions and peripherals the VM will access. Furthermore, the configuration must list the IDs with which the VM's managers (e.g., CPUs/rCPUs and DMA-capable devices) are recognized in the system. Once the user has prepared the application code and the configuration file for the VM, they can be compiled together with the Omnivisor code. To do it, the user must provide a list of cross-compilers, with one compiler designated for each core with a different ISA in the system. For instance, in the case of the ZCU, this would entail using the AArch64 compiler for main cores, the AArch32 compiler for RPUs, and the RISC-V 32-bit compiler for the soft-cores. The output artifacts are the Omnivisor binary and the binary images for the VMs.

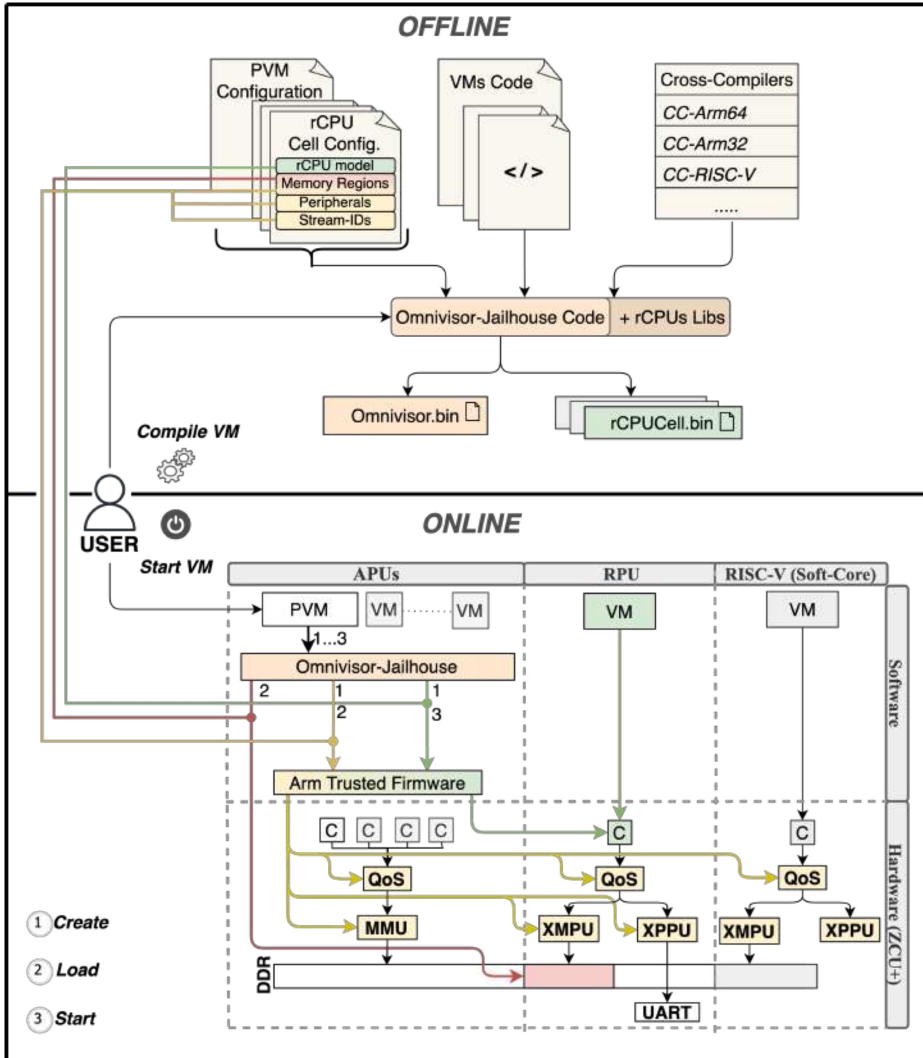


Figure 4.3: Architectural view of VM compiling and start procedures using the Omnivisor implementation on top of Jailhouse and the Zynq Ultra-scale+ board.

VM Start-Up Process (Online)

Omnivisor Enable Before starting an inmate, since the Omnivisor generalizes Jailhouse, we need to enable it from a Linux instance as explained in Section 4.2.2. Different from the vanilla Jailhouse, the configuration in our Omnivisor may also include a field for the *rCPUs*. If this is the case, the Omnivisor verifies whether the remote cores are already active, and if they are, it proceeds to shut them down. After that, it statically assigns their ownership to the PVM so that it can later assign the cores to other VMs. Additionally, the Omnivisor disables all the access permissions to the resources protected by the XMPUs such as memory and system registers. Then it configures the first entry of each XMPU's table to allow only the PVM to access those regions specified in its configuration file. This means that every manager outside the Omnivisor control can not access any resource in the system.

While the Omnivisor is up and running, launching a VM for the user involves using three simple commands, as depicted in Figure 4.3: (1) create, (2) load, and (3) start, reviewed below.

Create. The create command takes as input the configuration file of the VM, which is parsed to generate per-VM data structures. Resources are then *carved out* from the PVM and mapped to the new VM. For example, the requested remote and main cores are hot-plugged and detached from the PVM to be assigned to the new VM. After that, the isolation layer is configured. First, the MMU is programmed to manage the APU's memory region accesses. However, the *rCPUs* lack protection from the MMU and, if left unprotected, have direct access to all memory-mapped regions. Therefore, XMPUs are dynamically re-programmed to allow access permissions only to the resources requested in the configuration, avoiding unexpected accesses to sensible memory-mapped registers (e.g. DMA registers) and memory regions belonging to other VMs. Finally, in the case of soft-cores over FPGA, the Omnivisor configures the address translation by leveraging the SMMU.

Load. The load command requires as input the VM image. Initially, it verifies the image size against the carved-out memory reservation. Then, if the available memory is sufficient, it loads the VM image into memory.

Moreover, before starting the VM, the user can optionally regulate the memory bandwidth assigned to the managers to provide specific temporal guarantees to the VMs. The Omnivisor provides the knobs to do it, leveraging the aforementioned QoS and MemGuard interfaces in Jailhouse-RT [116]. This step is integrated into the load command during the start-up of a VM to avoid adding another PVM call between load and start. However, the Omnivisor also implements a PVM call for bandwidth allocation separately from the load to enable mission-critical reconfiguration scenarios. When selecting parameters for bandwidth allocation, it is the system integrator's responsibility to determine the suitable bandwidth for each VM, as this choice heavily relies on the application's requirements. However, using the tools offered by the Omnivisor, it is possible to empirically evaluate the parameters needed to enforce a specific maximum slowdown for a given VM. An example of a simple offline policy to automatize the choice of bandwidth parameters is provided in the experimental Section

Start. Finally, using the start command, the user initiates the VM start-up. Different MPSoC's architectures have different standards for power management of cores, such as the *ARM* Power State Coordination Interface (PSCI) [117] or the *Intel* Advanced Configuration and Power Interface (ACPI) [118]. However, the functionalities provided by these standards are similar. Therefore, the Omnivisor implements a series of generic power management procedures that are subsequently customized to the specific platform and core. We have implemented the procedures for the RPUs (ARM32-CortexR5F) and for a RISC-V soft-core (Pico32) deployed on the FPGA. In the ZCU the RPUs are overseen by the Platform Management Unit (PMU) core, which exercises control over their execution and power state. The only software with enough permission to call PMU services is the PSCI layer within the *ARM trusted firmware*. Consequently, we implement a specific ZCU module to communicate with the PSCI to request the wake-up and power-off of the RPUs. Regarding the soft-core(s), instead, we have implemented a memory-mapped configuration port in FPGA, and we expose this port to the Omnivisor to control the reset state of each soft-core.

The experimental evaluation in Section 5.2 demonstrates that deploy-

ing this model on a real system enables the seamless deployment of virtual machines on cores with heterogeneous ISAs (ARM and RISC-V) within a single platform, even if some or all are implemented as soft-cores in FPGA. Furthermore, the solution ensures robust and controlled spatial and temporal isolation of VMs, achieved through a combination of software/hardware mechanisms, which is the key to deploying safety-critical nuclear fusion control systems in future reactor infrastructure.

4.3 Real-Time Asymmetric Communication: RPUGuard

While the Omnivisor simplifies the use of VMs on remote cores, as previously demonstrated, the challenge of maintaining real-time requirements when multiple VMs need to communicate with these remote CPUs remains unresolved. Therefore, in this section, we address the following questions:

- *[Q1] Multi-Tenancy Abstraction*: how can a hypervisor abstract the RPUs to more than one VM, making the latter believe to be the only one using it?
- *[Q2] Communication Technologies*: are the current AMP communication technologies (adopted for inter-processor communication in MPSoCs) appropriate to realize deterministic and reliable applications with strict temporal constraints?
- *[Q3] Isolation Methodologies*: how can a hypervisor manage parallel communication channels ensuring isolation and a certain amount of communication bandwidth to each channel (useful for timing analysis and incremental development and certification)?

Aiming to answer to above questions, we propose a novel architectural solution for RPU sharing in mixed-criticality applications, through virtualization. In Section 5.2.4 We also provide experiments on real hardware showing the shortcomings of currently available AMP technologies and the benefits of the proposed solution in the context of a compelling case study related to the Vertical Stabilization (VS) system [119, 120] of the ITER experimental fusion reactor [121] (more details are given in Chapter 5).

4.3.1 RPUGuard Design

The envisioned system model foresees several VMs running on APUs with different levels of criticality that require services wich are running on RPU. The requested services can be very heterogeneous since the RPU, unlike GPU and FPGA, is a fully functional processor unit as shown in the previous tests. The RPU does not simply accelerate calculations, but it can implement more or less complex logic to manage incoming messages. The advantage is that the code executed on this kind of prequestedrocessor is expected to be more easily predictable and more suitable for performing safety-critical tasks as the hardware is certified (e.g., the ARM Cortex-R5 RPU can be used in safety-critical applications up to SIL 3 according to IEC 61508 and up to ASIL D according to ISO 26262). Hence, before describing the architectural design of the solution, we must define some assumptions.

The first assumption (*A1*) states that each VM is aware of the services offered by the VM running on RPU. This assumption is reasonable since typically, in embedded systems, applications that will be performed are well known before the execution, especially in a safety-critical environment with strict temporal requirements.

The second assumption (*A2*) states that the software stack executed on RPU is realized taking into account that more than one VM may require its services. Therefore, the RPU's software stack must manage its load guaranteeing the worst-case execution time of all exposed services.

Figure 4.4 shows the high-level design architecture for RPUGuard. Starting from the APU software stack, we imagine a number of VMs running different OSes or Bare-Metal applications characterized by different levels of criticality. However, each VM has at least one function that requires RPU usage, using an inter-processor communication mechanism. Even if assumptions A1 and A2 are respected, concurrent requests by VMs can still introduce non-predictable effects and interferences on critical tasks, as will be confirmed by our experiments (Section 5.2.4) in response to *[Q2]*. Therefore, we need to introduce a component in the hypervisor that manages requests: The *RPUGuard*. It uses a shared memory to exchange messages with the RPU software stack, as typically done by AMP communication mechanisms. To answer the questions *[Q1]* and *[Q3]*, the RPUGuard mechanism must capture all messages from and

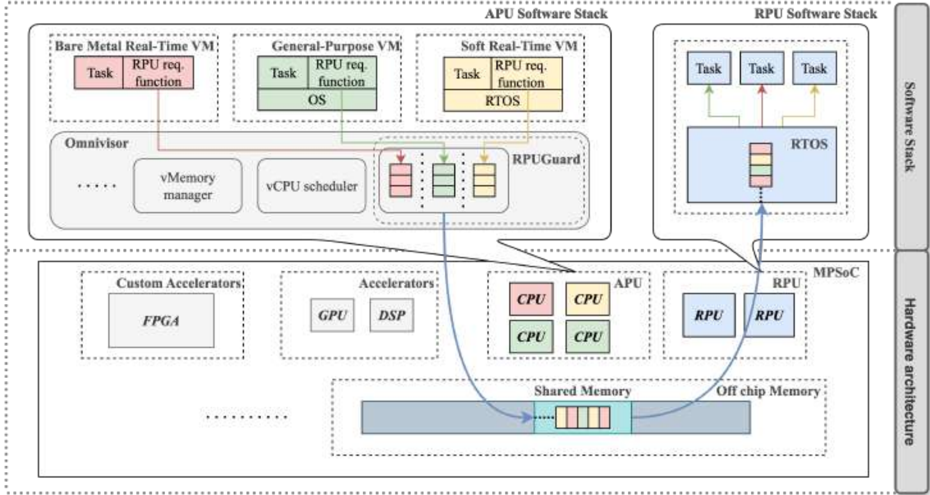


Figure 4.4: RPUGuard Architectural Design

to the RPU, establishing an isolated communication channel for each VM. The channels must be reliable and must have deterministic latency to meet the requirements in *[Q2]*, hence the bandwidth limits must be accurately defined. Finally, to fulfill the third research question *[Q3]* the mechanism must assure that a sudden increase in message requests to and from a VM does not induce an unexpected increase of latency to other VMs, and to the software that runs on RPU.

4.3.2 RPUGuard Algorithm and Implementation

Given the maximum communication bandwidth BW_{tot} of the channel, RPUGuard is in charge of partitioning it, ensuring a certain amount to each VM. When a VM starts, it can specify the maximum bandwidth BW_{vm} needed to communicate with the RPU during its execution, and RPUGuard can accept or refuse the request; granting or denying access to the RPU from the beginning of the execution. All VMs that start without any bandwidth request are managed with a best-effort policy by

Algorithm 1 RPUGuard: feasibility_check

1: $BW_{tot} \leftarrow$ Maximum bandwidth
2: $BW_{new} \leftarrow$ New VM bandwidth request
3: **if** $BW_{tot} \geq (\sum_{i=0}^{n-1} BW_{vm}(i) + BW_{new})$ **then**
4: Create new communication channel
5: $n \leftarrow n + 1$
6: $BW_{extra} \leftarrow (BW_{tot} - \sum_{i=0}^{n-1} BW_{vm}(i))$
7: The new VM can access the RPU
8: **else**
9: The new VM cannot access the RPU
10: **end if**

RPUGuard; to these VMs is reserved a bandwidth equal to

$$BW_{extra} = (BW_{tot} - \sum_{i=0}^{n-1} BW_{vm}(i)),$$

where n is the number of VMs which did require bandwidth guarantees at the start time. Then, only if a new VM starts specifying a bandwidth, RPUGuard must perform a feasibility check (see Algorithm 1) to verify that the new VM can be added without reducing the communication bandwidth of running VMs below their designed limits. If the feasibility check is completed successfully, RPUGuard manages the bandwidth assigned to each VM to include the new communication channel. When a VM mistakenly uses more bandwidth than it was suppose to (BW_{VM}), the RPUGuard must handle the requests. Hence, every time a new send message request to RPU arrives, the RPUGuard checks the elapsed time (T_{el}) since the last request from the same VM. If the elapsed time is below the allowed time interval (T_{min}) assigned to the VM, the request is delayed for a time interval equal to the difference $T_{min} - T_{el}$ plus another time interval T_{min} for each request not yet served. Doing so, RPUGuard prevents the VMs from sending messages with a frequency higher than those set at start time (see Algorithm 2).

Algorithm 2 RPUGuard: send_message

1: $R_n \leftarrow$Number of requests in queue
2: $T_{el} \leftarrow$ elapsed time since the last send message
3: $T_{min} \leftarrow$ minimum interval time between requests
4: $T_d \leftarrow (max(0, T_{min} - T_{el}) + (R_n - 1) * T_{min})$
5: delay the request for a time T_d
6: Send the message to RPU

Implementation Details

The proposed architecture design requires a message-based communication technique that makes use of shared memory. Recently, practitioners of AMP systems rely on an open-source framework named *OpenAMP* [122] to realize communication between asymmetric processors. OpenAMP is built on top of two other frameworks: *RemoteProc* and *RPMsg*. RemoteProc is used to manage the lifecycle of a remote processor; it allows loading the firmware into remote cores, starts and stops the remote cores, and configures resources that the remote cores might need during their execution. We don't need this mechanism since the same role is already implemented in the Omnivisor. On the other hand, RPMsg is a messaging mechanism implemented on top of the *VirtIO* framework [123] to realize inter-processor communication. OpenAMP is currently the most used technology to exchange messages with the RPU because it is highly portable and open source, but this framework alone is still not enough to ensure isolation and transparency to VMs running on APU cores. Our implementation makes use of the OpenAMP framework, but we implement the RPUGuard mechanism to guarantee isolation and transparency to VMs. Since our solution uses OpenAMP and more specifically RPMsg, we provide some further details on RPMsg in the following.

The RPMsg framework implements AMP transmission using three independent layers: *Transport*, *Media Access Control*, and *Physical Layer*. The *Physical Layer* is the most straightforward and comprises only two hardware components: shared memory and Inter-Processor Interrupt (IPI). The shared memory is accessible by both the cores and each time a new message is sent, an IPI is called to warn the remote core. The central layer, or *Media Access Control* layer, uses the *VirtIO* queue mechanism:

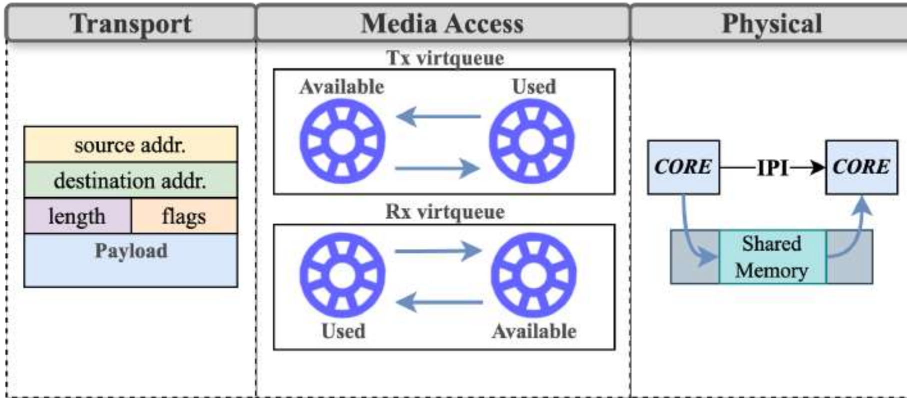


Figure 4.5: RPMsg layers

each virtqueue contains two ring buffers named “Available" and “Used". The Master core allocates the “used" ring buffer, writes inside them, and adds them to the “available" ring buffer. The remote core reads the data from the “available" ring buffer, processes them, and finally returns them to the "used" ring buffer. To implement the double-side communication two virtqueues must coexist in each channel, one to send messages (Tx virtqueue) and one to receive them (Rx virtqueue). Finally, the *Transport* layer simply defines a message format as shown in Figure 4.5. The format's header includes two fields for the destination and source addresses, one containing the message length, and one last field for some flags, while the payload contains the message data.

Usually, each remote core in the RPMsg component is represented by an RPMsg *device*, also known as RPMsg *channel*. To implement the envisioned multichannel communication, we could use an entire RPMsg virtual device, with its two virtqueues, for each VM. However, this solution has two intrinsic limitations: The RPU firmware, before the system starts, has to maintain the memory area needed for each RPMsg device. So, (I) the number of channels that VMs will require must be known in advance, limiting the flexibility of virtualization, and (II) if a VM only uses a certain number of buffers in one channel, leaving the others unused, it causes a waste of memory that is not desirable in an MPSoC with limited hardware resources.

A more interesting solution consists of using RPMsg endpoints that are logical connections on top of the RPMsg channel. Endpoints allow the user to bind multiple Rx callbacks on the same channel distinguished by the destination address of the messages. When an application creates an endpoint with the local address, all subsequent inbound messages with the destination address equal to the local address of the endpoint are routed to that callback function. Our solution uses endpoints but adjusts the order and frequency of messages sent between VMs via the RPUGuard component implemented in the hypervisor.

The experimental evaluation in Section 5.2.4 demonstrates that it is possible to realize isolated communication channels in software using existing AMP communication mechanisms and that real-time processors such as the RPUs are good candidates to deploy safety-critical control algorithms for plasma stability in nuclear fusion reactors.

4.4 Virtualizing Co-Processors: Microcontroller Virtualization

As shown in the previous sections consolidation in embedded systems can be obtained by employing ad-hoc tailored hypervisors running on top of processors powered by hardware extensions for virtualization. However, by its very nature, virtualization support integrated into application-class processors (e.g., Arm Cortex-A family) leads to downsides concerning real-time and power-saving requirements. The complexity of mechanisms such as page-based memory virtualization and translation caches, improves the average performance while decreasing predictability and hardening the certification. Moreover, in certain scenarios where one needs to combine several simple applications, these processors incur higher costs and introduce a significant unexploited energy consumption. Even in complex MPSoCs where partitioning hypervisors reach a good level of isolation, there is a strong limitation on the number of virtual machines and so the number of applications that are possible to deploy in the system. In these scenarios, despite performing worse, the use of microcontrollers is preferred, as they are characterized by low power consumption and high predictability [124].

Modern microcontrollers have enough computing power to run several applications at once [125, 126], but concerning memory management,

they only feature memory protection, while lacking memory virtualization. This places a limit on consolidating independent applications on such architectures while ensuring the required protection both from a temporal and security point of view. Integrating traditional page-based memory virtualization technologies into microcontrollers would increase nondeterminism as well as cost and power consumption, which would make it more appropriate to directly leverage application-class processors. This clearly points out the need for ad-hoc memory virtualization solutions taking these constraints into account.

In light of this, in this section, (1) We examine two hardware-based models to support lightweight and predictable memory virtualization for medium-end microcontrollers. We specifically categorize *medium-size* microcontrollers as those having sufficient computing power for multiple applications, but only featuring memory protection mechanisms with no memory virtualization support (such as the RPU). Both models enable the consolidation of multiple applications and RTOSes with overlapping memory addresses on a single microcontroller but with different impacts on energy consumption and predictability. Then, (2) we provide an open-source implementation of the two models on a RISC-V processor together with (3) an extensive and reproducible analysis of the area along with performance overhead investigation on numerous benchmarks.

4.4.1 Lightweight and Predictable Virtual Memory

The problem of virtual memory predictability is widely recognized in the literature and numerous solutions have been proposed to address it. Most of these solutions optimize the paging mechanism rather than proposing a different MMU design such as [127, 128, 129]. However, we agree with Puffitsch et al. [130] in pointing out that the requirements of hard real-time systems are different from general-purpose systems, and that these different requirements call for a different virtual memory design. Modern microcontrollers have reached enough computing power to support multiple execution environments [131, 132]. However, different from general-purpose virtual memory, specific requirements related to flexibility, security, efficiency, and predictability are to be enforced in order to satisfy the industry's demands. We call a virtual memory scheme supporting the following requirements a *LPVM*.

Predictability. The introduction of LPVM must avoid nondeterminism, such as the one caused by Translation Lookaside Buffer (TLB) and memory swapping mechanisms. If extra latency is to be added for the sake of performance, the upperbound must be known in order to guarantee the WCET of each workload.

Lightness. An LPVM scheme must require few resources, both in terms of memory usage and power consumption. Cost-related features are to be preferred to execution time.

Isolation. Multiple environments must be isolated, providing protection from both malicious software and programming errors, and minimizing the chances of critical system failure.

Memory efficiency. Real-time systems assume their memory to be a precious and scarce resource. Under the assumption that an execution environment features sparse memory and the host system has more physical resources than the tenants, memory regions can be rearranged in order to optimize memory usage. Unlike general-purpose virtual memory, execution environments are unlikely to dynamically require more memory or implement a disk swapping mechanism. Hence, segmentation shall be preferred to paging because it can provide finer memory organization and no internal fragmentation, also avoiding the burden of defragmentation.

Memory Transparency. Modern critical systems require to dynamically update running environments and less frequently deploy new ones independently of the others. This implies that a translation mechanism should exist to provide independent linking and execution. However, real-time environments will only be assigned a specific translation when deployed, making a translation mechanism lighter compared to classic virtual memory.

There are two main *use cases* where LPVM can be applied: predictable virtual machines management and isolation-enhanced operating systems.

For the former, a single level of LPVM can be used by real-time hypervisors to support the deployment and updating of multiple physical-memory RTOSes provided by distinct parties. This applies to automotive

where different Electronic Control Unit (ECU)s can host multiple subsystems driving actuators and sensors, to achieve ECU consolidation [133]. Furthermore, LPVM can be used to enforce simple process abstraction, providing isolation among different workloads, similarly to TockOS [134]. Such an approach is ideal for medium-end microcontrollers, granting them flexibility and robust isolation. While LPVM might be purely implemented in software, performance degradation must be minimized with dedicated hardware support, as described in the next Section

4.4.2 Hardware-Based LPVM Methods

While microcontrollers implement software protection by means of table-based structures, e.g. ARM MPU, RISC-V Physical Memory Protection (PMP), application processors support sophisticated MMUs which use hardware page-walkers to explore a memory-based *radix trie* structure through implicit accesses in order to find the physical address, using TLBs as caches for those translations.

The simplest approach to implement LPVM consists of reusing a hardware table configured by means of a private memory-mapped address space. As depicted in Figure 4.6, each entry in the table describes a memory region in the form of a $\{base, range, relocation, permissions\}$ quadruple. Each transaction (i.e. read, write, execute) must be checked against all entries in parallel; if the access falls into the interval $[base, base{+}range]$, access permissions are checked and finally, an *address relocation function* is applied. A plain relocation would map a virtual address onto a physical address. This would require a 32-bit field per entry, but also simple logic and maximum translation space coverage. On the other hand, more complex relocation functions can reduce the number of used bits, increasing scalability, but hurting remapping coverage and also requiring more logic for each entry.

A tabular approach is quite simple to design and brings the benefits of small hardware overhead and no extra latency. However, the number of hardware entries does not scale well in terms of area occupancy and power consumption because of the associative table along with the extra control and translation logic. Moreover, MPU-like structures lead to other downsides, as depicted in [135]. Specifically, positioning a new component into the memory path is likely to reduce the maximum working frequency.

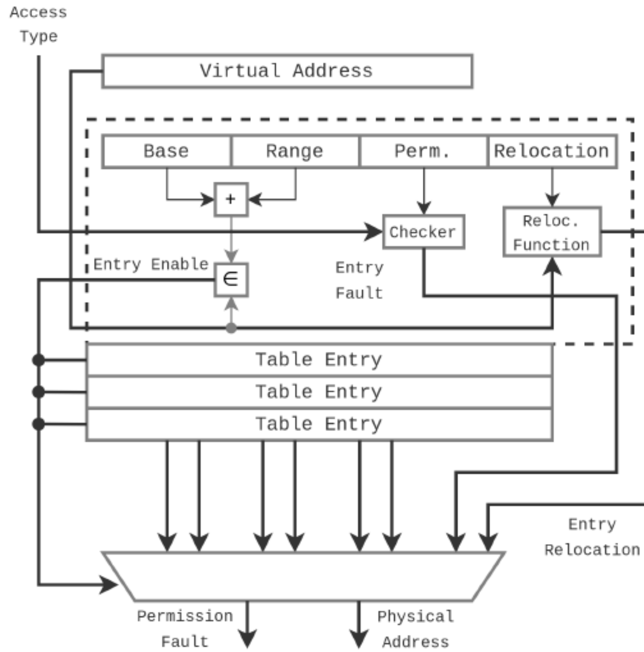


Figure 4.6: Table-based LPVM. The virtual address is checked against all entries, and permissions are computed thanks to a checker module. The relocation function is then applied, and the correct entry is selected. If no region is found, or a permission mismatch occurs, a fault is raised.

Finally, table entries are to be considered part of an environment context (especially in the case of VMs), making context switches slower.

These issues can be addressed by adopting a trie-like structure. Pan et Al. [113] describes tries as a possible generalization of ARM MPUs; differently from MMUs, LPVM can leverage the sparsity of microcontroller memory to implement a hardware walker for *GRT* [136], minimizing memory usage and latency. In fact, a GRT is based on the key idea that subparts of a virtual address prefix can be skipped during trie exploration because only one entry in that trie exists for it. As a consequence, multiple levels can be avoided, reducing overall performance degradation. LPVM-GRT memory layout can be structured in two parts: the *ST* and the *LT*. We call a level of the ST a *STT*; each STT will have 2^{prefix_size} entries,

but not all of them need to be defined. Because dynamic environment deployment can be quite uncommon or not required at all, the GRT will be programmed at boot-time and unlikely to be modified again. Therefore, it is possible to have a compressed representation, where a STT only contains valid entries.

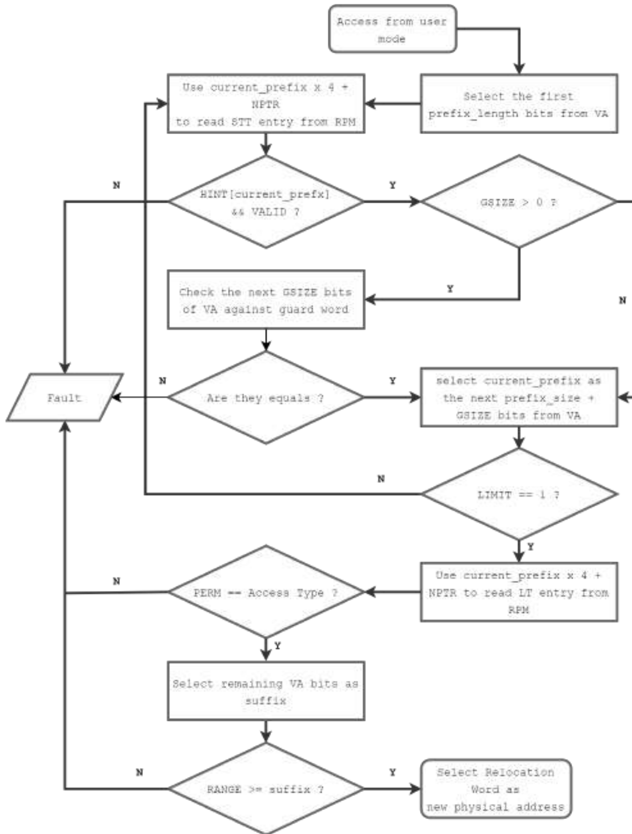


Figure 4.7: GRT-based LPVM flow diagram describing the translation process starting from a user mode access to the physical address generation.

A single STT entry is a 64-bit structure made of two 32-bit words, the `control word`, structured in five fields {`NPTR[31:19]`, `VALID[18]`, `LIMIT[17]`, `GSIZE[16:12]`, `HINT[11:0]`}, and the `guard word`. `HINT`

is a 12-bit field that provides information regarding the next level. If the `HINT[next_access_prefix]` bit is high, then the next entry exists, otherwise a fault is raised. If the `LIMIT` bit is high, the next level is the LT, so `HINT[envid]` will be used instead, where `envid` (environmental ID) refers to a task ID or a Guest ID and serves the purpose of distinguishing overlapping virtual addresses among different environments. The `VALID` bit specifies if the current entry is valid or not. If an invalid entry is implicitly accessed, a fault occurs. `GSIZE` specifies the guard size of the `guard word`. If it is not zero, the next `GSIZE` bits of the current address are compared to the `guard word` for the current entry. If no match is found, a fault occurs, otherwise, the next prefix will be selected as the first prefix after the matched guard. Finally, `NPTR` is used as a pointer to the base of the next STT, while the entry will be addressed by using the next prefix.

If an entry with a high `LIMIT` bit is reached without any exception, a valid segment base address is found for the provided virtual address. `NPTR` is then used to access the LT, which contains the permissions and range in the first word, the `configuration word`, and the plain translation in the second word, called `relocation word`. Permissions are encoded in the three least significant bits of the `configuration word`, while the remaining bits are used to encode the range, meaning a segment can be as small as eight bytes. If the current access is permitted and the virtual segment offset falls into the physical segment size, the access is allowed and translated by means of the `relocation word`. For LPVM-GRT the best relocation function is a direct mapping. Because the number of clock cycles is $N+1$, where N is the number of walked ST levels, the LPVM walker hardware requires a static and fast on-chip memory only accessible by the privileged code, called *Region Protection Memory (RPM)*. User code will be accessed from main memory and translated by the walker reading from the RPM, while privileged code will be protected by using a small MPU-like structure with no translation. The flow is depicted in Figure 4.7, while the architecture design is shown in Figure 4.8. As long as the RPM is faster than main memory, latencies can be partially masked, providing benefits similar to TLBs but with no stochastic behaviors. LPVM-GRT model brings great flexibility and scalability compared to table-based approaches, with a fixed hardware overhead. Also, because RPM can contain translations

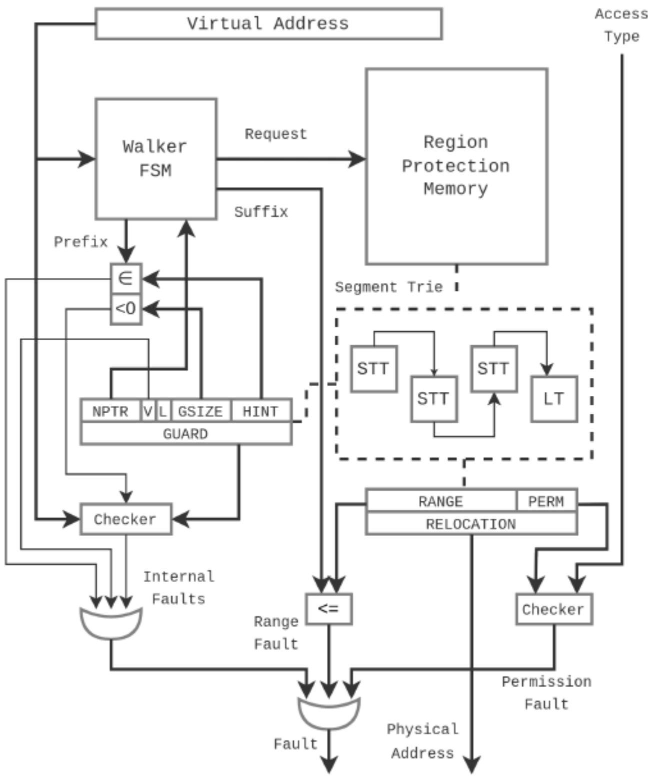


Figure 4.8: GRT-based LPVM. The virtual address is used by an FSM to extract the current prefix and suffix. According to the current STT entry, checks are performed on the control word and guard word and a new request to the RPM is submitted. If no fault occurs during ST exploration, the physical address is retrieved from the LT.

for each guest segment, it does not enlarge environment contexts, enabling fast context switches. The main downside consists of adding extra latency to each access. In terms of predictability, however, this extra latency is strictly bounded by the GRT depth. Furthermore, the sequential nature of the LPVM walker potentially allows limited or even null impact on the critical path delay, and hence the circuit working frequency, as confirmed by the experimental results in Section 4.4.3.

4.4.3 LVPM Experimental Results

We implemented both the aforementioned schemes on a 32-bit open-source RISC-V processor, cv32e41s [137]. It supports a new nonstandard extension called `extended PMP`, which allows for the processor to host a trie-based PMP and the translation mechanism itself. Also, all the following results have been collected from our clock-cycle accurate simulator, freely available at [138].

Table 4.1: Area, Power, and Frequency results for cv32e41s-based soc baseline, with different PMP and LPVM-TAB configurations and LPVM-GRT as well. Data refer to the Nexys A7-50T board.

	Look Up Tables	**Flip Flops**	**Power Usage**	**Working Frequency**
Baseline (no PMP)	7223	5853	201 mW	$\sim 48Mhz$
8 PMP entries	9176	6232	214 mW	$\sim 40Mhz$
32 PMP entries	15323	7136	228 mW	$\sim 40Mhz$
64 PMP entries	27651	8343	273 mW	$\sim 40Mhz$
8 LPVM-TAB entries	10071	6488	220 mW	$\sim 35Mhz$
32 LPVM-TAB entries	19517	8127	251 mW	$\sim 35Mhz$
64 LPVM-TAB entries	32072	10285	298 mW	$\sim 35Mhz$
LPVM-GRT	10193	6592	211 mW	$\sim 40Mhz$

All experiments performed are easily reproducible by means of a set of scripts that can be found at [139]. Results from Table 4.1 show various synthesis results for multiple configurations of cv32e41s on a Nexys A7-50T board. A key observation is that the maximum working frequency decreases by 16% compared to the baseline because of the PMP, regardless of the number of entries, which instead only increases the number of resources used and, consequently, the power consumption. While eight PMP entries are quite acceptable, 32 or more entries result in a Look Up Tables (LUT) usage comparable to the entire core itself. Because LPVM-TAB is built on top of the PMP, it introduces further overhead compared to the PMP configurations. Eight LPVM entries only cause a 9% increase in LUTs and a 4% increase in Flip Flop (FF)s, because of a new register for each entry and a very simple logic. However, despite using simple offsetting as a relocation function, the impact on frequency is considerably

higher since it downgrades it by about 12.5%, for a total degradation of approximately 27% with respect to the baseline. This is due to the fact that no pipelining mechanism is provided inside the PMP in order to avoid extra latency. Also, while eight entries do not add a significant overhead in terms of area and power, they scale more than linearly, making a 64 entries configuration quite heavy and inefficient for medium-end microcontrollers with a worsening factor of 16% for area and 9% for power. On the other hand, an LPVM-GRT implementation does not depend on the number of entries and does not cause a significant overhead on the frequency compared to the baseline. While its overhead is slightly larger compared to an eight LPVM entry configuration in terms of area, energy requirements are even better with an improvement of about 4%, making it more resource-effective than table-based LPVM (LPVM-TAB).

Although the results in Table 4.1 seem to highlight the benefits of LPVM-GRT compared to the table-based approach, it still requires a new on-chip memory to host the trie. Moreover, this analysis alone cannot provide a full understanding of how the LPVM-GRT memory layout affects the processor in terms of time execution overhead and predictability. Therefore, we decided to test our solutions on a set of realistic benchmarks. Specifically, our choice has gone towards TACLEbench [140], due to the lack of dependencies, a distinguishing feature of these benchmarks. However, due to the limitations related to the nature of the used simulation platform (Verilator) and the absence of a floating-point extension of the RISC-V processor we used, we ended up using a subset of them.

We chose to run the benchmarks on four distinct simulated platforms, namely *"Small"* (S), *"Medium Static"* (MS), *"Medium Dynamic"* (MD), and *"Large"* (L) in order not only to show the overhead that the trie-based solution causes to the benchmark execution times, but also to establish when its use is advised as opposed to the table-based approach by varying the complexity of the microcontroller where the LPVM is implemented. We distinguish the four platforms by two main factors, which are the maximum depth of the tree, and the presence or absence of an off-chip dynamic memory. The maximum depth of the tree is an index of the number of memory regions that the processor expects to manage; for each new memory region added, we need a node in the last level of the tree. Therefore, a higher tree enables a higher number of possible memory

regions. We assume that smaller platforms are used to run a smaller set of isolated workloads compared to larger ones. Therefore, we decided to implement a tree with a maximum depth equal to 2, 4, 6, and 8 in the "S", "MS", "MD" and "L" platforms, respectively. Moreover, while the "S" and "MS" platforms are designed with an on-chip static memory, the "MD" and "L" platforms implement also an off-chip dynamic memory whose access time is an order of magnitude greater compared to the static one. This is because microcontrollers designed to run a larger number of applications have higher memory requirements, which implies off-chip memory to be used. In these larger platforms, all the user code is stored on the off-chip memory, while the privileged code (e.g., hypervisor, firmware, and the entire tree) is hosted on a small on-chip static memory.

The results in Figure 4.9 show the worsening factor for execution time when the trie-based relocation mechanism is enabled compared to execution times without relocation (yellow line) and with table-based relocation mechanisms (red line). The yellow line represents the baseline where neither of the LPVM mechanisms is used while the red line represents the table-based relocation overhead, which is always equal to 14% since, as previously explained, it only depends on the lowering of the processor frequency. Due to the limited space, we decided to show only the worsening factor, while the absolute temporal results are available in the associated repository [139]. Furthermore, for each run, in order to retrieve the WCET, we impose each memory access to walk the tree up to the last level. Results show a clear decrease in the deterioration of the execution time for platforms with off-chip dynamic memory ("MD": 11.1%-19.7% and "L": 16.6%-29.8%) compared to platforms with static on-chip memory only ("S": 65.8%-94.3% and "MS": 113.3%-155.0%). Intuitively, this is due to the pipelined nature of the processor; every time a memory access is requested, the processor continues its execution and if there is another memory request, the processor starts the tree walk in the RPM while still waiting for the response to the previous request. This mechanism intrinsically hides the tree overhead by causing, in these platforms, a smaller degradation, comparable with the one caused by a table-based approach and in some cases, when the memory accesses are less frequent and closer in time, it is even slightly better in "MD" platforms with a worsening lower than 13% (e.g., matrix1, recursion, rijndael_dec, ...). In

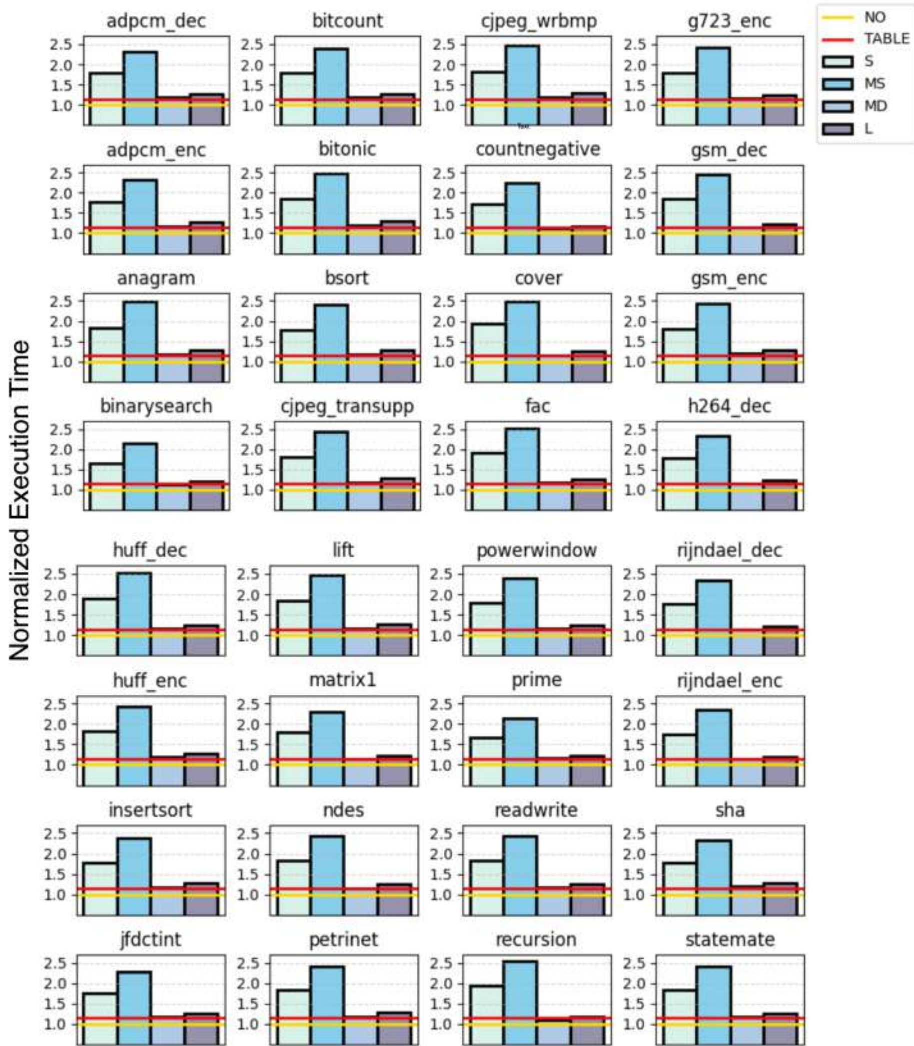


Figure 4.9: Benchmarks normalized execution time. For each benchmark, the GRT-based LPVM overhead is shown compared to the baseline and the table-based LPVM overhead. All the values are normalized to the baseline to provide a fair comparison between the benchmarks.

view of this, and given the most critical degradation shown in the first two platforms, we believe that a table-based approach is preferable for small platforms with only static memory on-chip and a small number of envisioned memory regions. On the other hand, the GRT-based approach appears to be a reasonable solution for enabling memory virtualization in medium to large microcontrollers. From a point of view of predictability, it is important to highlight that given the absence of any swap mechanism and TLB caches, the GRT-based solution implies a maximum latency to the execution times of real-time applications that is easy to compute given the maximum height of the tree. Moreover, the execution of less critical applications can benefit from the sparse and incomplete nature of the tree structure, resulting in better average performance.

4.4.4 Balancing Partitioning and Flexibility in Heterogeneous Systems

In modern heterogeneous architectures, static partitioning has emerged as the most reliable method for maintaining real-time performance as discussed also in Section 3. This approach ensures that resources are strictly allocated to specific tasks, thereby minimizing interference and preserving temporal isolation. However, this comes with a significant limitation: resources are preallocated at startup, restricting the ability to allocate them based on real-time demands dynamically.

On the other hand, fully sharing resources across tasks and cores introduces substantial challenges in predictability, as extensively discussed in Section 3.3. The unpredictability associated with resource contention can severely undermine the real-time guarantees required by critical applications.

Looking forward, we believe that with the advancement of hardware mechanisms that provide finer control over resource allocation—such as the LPVM mechanism introduced in Section 4.4, it will be possible to achieve a more flexible approach. By leveraging these kinds of technologies tailored for real-time applications, it may become feasible to dynamically allocate resources at runtime while still maintaining the strict performance and predictability requirements necessary for real-time systems. This balance between flexibility and reliability could represent the next evolution in the management of heterogeneous architectures, enabling more efficient

and adaptable systems without compromising on critical real-time performance.

Before we move to the evaluation section, where the practical usability of our proposed method in real-world nuclear fusion applications is assessed, the next section introduces how the Omnivisor can be integrated into cloud orchestration systems like Kubernetes. Specifically, the upcoming section explores this integration in detail, demonstrating how partitioning hypervisors can be seamlessly managed within such orchestration environments using our proposed solution, RunPHI.

4.5 RunPHI framework for Omnivisor Orchestration

The Industry 4.0/5.0 vision relies on the replacement of hardware elements with cloud-native components to simplify the development and management of industrial applications and reach a sustainable, flexible, and resilient environment.

Traditional cloud-native applications use virtualization to simplify the management of services and guarantee elasticity, flexibility, resiliency, and cost-effective resource usage. Hence, cloud environments mainly use virtualization technologies prioritizing consolidation over isolation, like OS-level virtualization (i.e., containers) integrated with orchestrators (e.g., Kubernetes). However, cloud-native components running in containers and VM [141] cannot yet guarantee low and predictable response times, availability, and reliability required by industrial settings.

Conversely, the recent adoption of virtualization for industrial applications, as shown previously, is pushed by the need to achieve isolation while consolidating systems to meet SWaP-C requirements [142, 143, 27]. Thus, simplicity is prioritized over features to cope with certification, e.g., ARINC-653, DO-178C, ISO 26262.

The Omnivisor model proposed in this dissertation is based on SPHs which, as detailed in Section 2.3, are gaining the limelight in this perspective, nevertheless, they require a remarkable manual configuration effort and are not supported by cloud orchestration tools modified to support real-time and critical applications [144]. Hence, real-time critical applications cannot benefit from easy and automated management at a scale,

which includes deployment, networking, scaling, and migration. The complexity is exacerbated by the heterogeneity of nodes in industrial edge settings, which may span from low-end multi-core embedded boards (even with asymmetric cores, such as application and real-time processing units) to fully-fledged server machines. Currently, such heterogeneity is seen as an obstacle for cloud technologies, as the same application must be recompiled or rewritten for a different operating system or instruction set architecture.

In this section, we aim instead to take advantage of industrial nodes' heterogeneity by introducing the *runPHI* framework, as the convergence between container orchestrators and Omnivisor. The framework works as a glue between the orchestration functionalities and the isolation capability of the Omnivisor.

The inherent node diversity allows for meeting stringent dependability requirements. In particular, we introduce unprecedented orchestration primitives to leverage diversity, namely *Diverse Replication*, *Seamless Migration*, and *Diversified Rolling Update*. For instance, a controller can be replicated on a fully-fledged edge cloud server and on an embedded board to avoid common mode failures through the diversity of the implementation and platform. If a replica fails, it could be migrated to yet another different platform, seamlessly.

In addition to implementing the runPHI framework, this section proposes a probabilistic model for managing real-time tasks running within standard containers and those deployed on an Omnivisor across a cluster of heterogeneous nodes. The model unifies the management of fault tolerance and timing unpredictability, and it can be used by the orchestration system when applying our orchestration primitives to plan the deployment of tasks implemented with diversity to meet the required dependability level.

In summary, in what follows we introduce:

1. The RunpHI framework to integrate Omnivisor in orchestration systems.

2. A set of novel orchestration primitives that leverage hardware diversity to improve reliability.

3. A probabilistic model that accounts for hardware diversity to drive

orchestration decisions.

4.5.1 Advanced Orchestration Primitives for Mixed-Criticality Systems

Containers are commonly defined as a standard way of packing applications together with their required libraries, forming a *container image* [145]. Following this definition, a real-time application (e.g., a periodic POSIX thread) could be either compiled against Linux libraries to become a Linux container or compiled against a library RTOS (e.g., Zephyr, NuttX), to be packed as a bare-metal VM.

Both containers and bare-metal VM can be managed by Cloud orchestrators with the same interface. Although the Open Container Initiative (OCI) specifies [146] how to divide a container image into layers to reuse the layers containing libraries, an image containing only a single executable file, such as a bare-metal VM is still compliant with the OCI specification.

The availability of the same application as a container or a VM for multiple platforms enables unseen orchestration primitives, particularly relevant for industrial settings. Such primitives are defined as follows:

Diverse Replication: When an application should be replicated for either redundancy (often dictated by safety-related standards) or scalability reasons, this primitive allows generating replicas considering additional constraints. The two main considered constraints are (i) *criticality*, indicating the criticality of the replicas, and (ii) *replication mode*, indicating how to configure the replicas, e.g., in Triple Modular Redundancy (TMR), 2-out-of-2 configuration using different networks. Given the number and criticality of replicas, along with the replication mode, the primitive enables easy deployment of diverse replicas by selecting the nodes to spawn the containers and VMs to comply with the constraints. For example, an orchestrator can spawn a Linux container plus two bare-metal VMs across three different platforms for an application requiring three replicas. Diversity is guaranteed thanks to the different images used.

Seamless Migration: This primitive allows the seamless migration of containers nad VMs between potentially heterogeneous platforms while

accounting for the isolation guarantees provided by the nodes. When a migration is required (e.g., upon a node failure), the primitive allows respawning an application onto a different platform with the same isolation guarantees of the previous one. For example, the primitive transparently migrates a container from a Xenomai-based node (popular co-kernel for real-time) to another real-time Linux-based one to achieve similar timing guarantees. If no node providing comparable isolation guarantees is available, the primitive can select a node with lower guarantees to provide a possibly degraded service. For instance, a complex controller runs on a powerful Linux-based edge server to leverage large computing resources. Upon a failure, the controller can be migrated as a bare-metal VM to a different platform to run a simpler implementation, which can only provide minimal functionality or bring the system to a safe state

Diversified Rolling Update: Rolling updates gradually replace the running application with updated ones, guaranteeing a minimum number of running replicas to avoid downtime. In this regard, the primitive allows performing a diversified rolling update, where the applications to update for each round are selected according to the platform. The idea is to always keep diverse containers and VMs running on different platforms to prevent common mode failures due to regression faults affecting the same platform. Thus, during the update, the primitive selects a number of diverse (if possible) containers and VM to stop, starts their updated version, and when the spawned containers are ready a new update round starts.

4.5.2 RunPHI: Managing Mixed-Criticality Heterogeneous Systems

In the proposed framework, as in standard orchestration systems, the user defines the desired state for a cluster of nodes. The desired state includes not only information about running containers and VMs, but also details about their requirements in terms of isolation levels, physical resources, real-time constraints, and other specific needs.

Each cluster node declares the maximum level of isolation that can be provided. While Linux-based (both single and co-kernel architectures) containers can guarantee isolation only to a certain extent [144], a node

supporting Omnivisor can offer the maximum level of isolation available.

The orchestration system assigns each application to a node that can host it, i.e., the node *i)* has an image of the container or VM available, *ii)* provides an isolation level that matches the application's requirements, *iii)* has the necessary resources available.

RunPHI Image Building

To have an image available for a particular node, the building process must integrate the creation of the images used for containers and VMs.

Provided that a POSIX-compliant application must be packaged to be orchestrated, the same source code can be used to build both a standard Linux container and bare-metal images. The image building process is depicted in Figure 4.10.

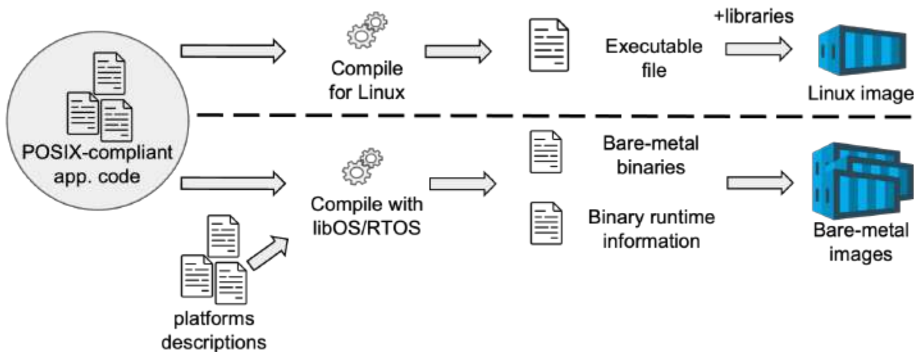


Figure 4.10: Image building workflow.

When building for Linux, the application is compiled using the application toolchain, and then an image containing both the application and its required libraries is created. The OCI specifies a standard for describing these images, which are divided into layers that can be shared and reused by multiple containers. For example, the base layer of an image might be an Ubuntu image, containing Ubuntu libraries. A second layer includes the libraries required by the application, and a third layer contains the application binary. When the application changes and a new image is built, only the last layer needs to be modified and downloaded.

When building for a bare-metal image, a POSIX-compliant RTOS or libOS (e.g., Zephyr, VxWorks, NuttX) can be used to pack the application and a minimal OS into a single bare-metal binary. In this context, the POSIX-compliant libOS RTOS acts as a minimal bare-metal runtime that abstracts hardware resources. The building process targets a specific node/boards, and relies on the platform descriptor typically shipped with the RTOS. During the building stage, in addition to producing bare-metal binaries, the process outputs *binary runtime information* files. These files contain the necessary information for the hypervisor to start a container or a VM. For example, the binary runtime information file may include the address space layout of the bare-metal binary, including the virtual start address of the binary, as described in Section 4.5.2.

We evaluated other designs, such as compiling an application to WebAssembly (WASM) code and running it on a minimal WASM runtime. However, we found that support for real-time applications in WASM is limited and immature at the time of writing. Additionally, the technology readiness level of bare-metal WASM runtimes is still in its early stages.

RunPHI Image Execution

Here, we depict the node stack needed for implementing RunPHI images, comparing it with the architecture of a Linux container stack (see Figure 4.11).

Standard Linux Container Stack. An orchestration agent present on each node of the cluster interacts with the container manager through the Container Runtime Interface (CRI) API. This interface exposes functions to manage sandboxes (i.e., a group of application containers in an isolated environment with resource constraints) and their respective containers. The container manager tracks the state and lifecycle of containers, downloads or updates the container images, and manages virtual networks for the containers. Specifically, the container manager selects a suitable image version to download, according to the target OS and ISA. Next, the container manager relies on a shim daemon (typically one per container) to interact with the low-level container runtime and the container itself. The shim layer abstracts low-level runtimes and it exists as long as the controlled containers, and redirects the stdin, stdout, and stderr streams

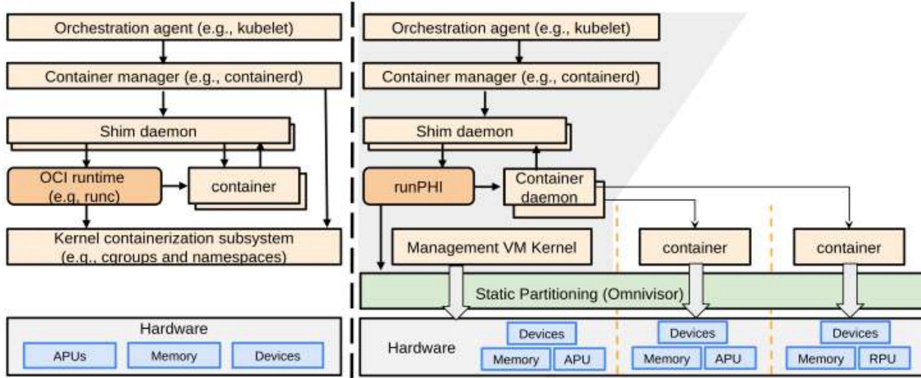


Figure 4.11: Linux container stack (on the left) vs. runPHI images stack (on the right).

to a file. The shim daemon interacts with the low-level container runtime through an API defined by the OCI standard, which mandates exposing at least a minimal set of functions to create, start, kill, delete, and get the status of a container. The low-level OCI runtime performs the system calls to configure the required resources for the container. For example, in the case of Linux containers, the low-level runtime performs system calls to configure the *cgroup* and *namespaces* for the container, creates the processes, and moves them into the container. Once a process runs in a container, its access to hardware resources is regulated by the kernel.

RunPHI Image Stack. In the runPHI architecture, the container manager is responsible for downloading images suitable for the specific platform.

The core of our proposed framework [147] is a low-level OCI runtime, responsible for creating, starting, killing, and deleting containers and bare-metal VMs. Instead of relying on the containerization subsystem of the kernel, runPHI configures a static partitioning hypervisor to create a sandbox for the containers.

The framework extends the utilization of the *Omnivisor* (see Section. 4) enabling the orchestration of images even for co-processor lacking MMU such as the RPUs. Once the static partitioning through the Omnivisor is configured and started, runPHI relies on custom daemons to

create a bridge with the partitioned VMs. In particular, the daemons are in charge of maintaining the status consistency and the communication channels concerning the containers and the VMs. The current network management in partitioning hypervisors lacks real-time guarantees. However, several approaches can enhance this aspect. One approach is to use a broker to manage the network [112]. Another method involves using a board with multiple network interfaces partitioned and assigned to different VMs [9]. Additionally, hardware support such as SR-IOV with real-time capabilities can be used [148].

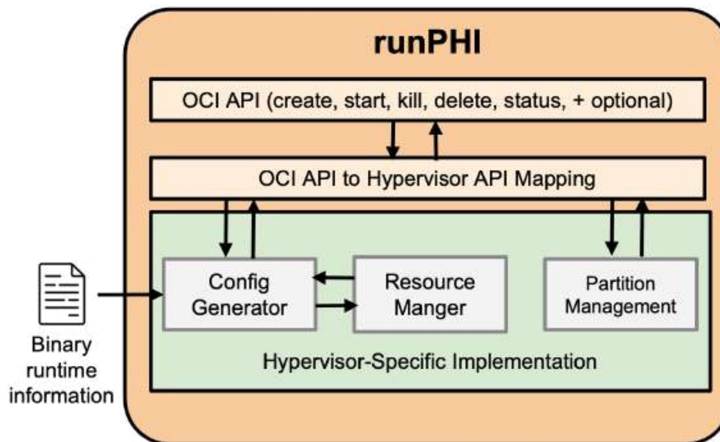


Figure 4.12: High-level architectural description of runPHI, highlighting the interactions of the configuration generator and partition management.

In Figure 4.12, a high-level architectural description of runPHI internal is depicted. The topmost layer maintains compliance with the OCI APIs. Below that, an intermediate layer addresses the semantic mismatch between parameters and data structures belonging to containers and concepts associated with VMs. The intermediate layer relies on the bottom layer, which is hypervisor-specific. This hypervisor-specific layer contains commands to manage the partition lifecycle. Most partitioning hypervisors, as well as Omnivisors, rely on a configuration file to specify the resources assigned to a partition. Hence, a component that automatically generates the configuration file is necessary. This component must adhere to the indications of a resource manager, which tracks available resources

on nodes over time. The output of the configuration generator is a file containing all details for running partitioned VMs on the target platform. To achieve this, it relies on the information provided in a binary runtime information file shipped with the bare-metal image (see Figure 4.10) and the container configuration. For instance, the image description contains the virtual start address where the binary must be loaded in memory, which is utilized to program the virtual-to-physical address translation provided by the Omnivisor. If necessary, the configuration generator may fill the gaps with predicted values for missing resources in the container description, according to requests from the orchestration platforms and the current usage of hardware.

4.5.3 Modeling Fault Tolerance and Timing Uncertainty in Heterogeneous Systems

We here introduce a model to show how the orchestration of containers and partitioned VMs can be considered in the system design. The main idea is to create a general framework that unifies the management of fault-tolerance and non-deterministic timing effects under a joint probabilistic model.

Indeed, on the one hand, in a distributed edge system, nodes have different hardware/software characteristics and hence timing guarantees. Sometimes, precisely defining the WCET [149] is either barely impossible or overly pessimistic. For example, a real-time Linux-based container running on complex COTS hardware presents fewer timing guarantees due to interferences than industrial machinery equipped with a minimal real-time system.

On the other hand, simplistic assumptions are often used in fault tolerance techniques for real-time systems [150]. For example, assumptions include a well-defined WCET, perfect error detection, guaranteed time between faults/failures, limited fault/error model, etc. For example, in [150], the authors acknowledge that dealing with permanent failures different from total system failure is complex because they also include fail-partial and fail-slow behaviors.

The timing nondeterminism (due to complex hardware) and the fault tolerance are generally addressed separately. With the proposed model, we aim to deal with different node hardware/software determinism character-

istics, schedulers, and fault tolerance techniques, with the same theoretic framework. This enables a holistic view of the system, in which we only care about a task WCRT. The WCRT hides the complexity of models and assumptions local to a node, including failure behaviors, fault tolerance techniques, and timing determinism.

To this aim, we use the probabilistic-WCET ($pWCET$ [151, 152]) random variable, to model both the inherent hardware/software unpredictability and the possible longer duration due to fault/failures handling or fail-slow behaviors. In this sense, different errors like crashes, unexpected longer executions, livelocks, and deadlocks, are all modeled like a tail in the $pWCET$ distribution.

We model our edge cloud environment as composed of a set of microservices Γ^g deployed across the cluster, which provides functionalities to clients. A microservice is a set Λ_i of k task replicas $\tau_{i,j}$ (with $j \in [1, k]$) deployed across one or more nodes. Depending on the replication scheme, the replicas can be either identical for load-balancing purposes, or diverse for fault tolerance purposes.

Without a lack of generality, we assume a fixed-priority periodic task model, but any other task model using $pWCET$ would work. A task is defined as $\tau = (p\vec{WCET}, T, D, p)$, where $p\vec{WCET}$ is the vector of $pWCET$ defined for the cluster nodes that can host the task, T is the period, D is the relative deadline, and p the priority level. A task τ_i is assigned to only one cluster node. The $pWCET$ depends on the hardware/software characteristics of the node. In particular, the assurance provided by the node in terms of resource isolation, defined as $A = f(\alpha, \beta, \gamma)$ in [144, 153] determines the skewness of the $pWCET$. In this sense, the $pWCET$ accounts for nondeterminism in hardware contention and resource interference, but also fail-slow behavior, recovery blocks, and forward error recovery.

Multiple tasks $\tau_i, ..., \tau_k$ may be assigned to the same node, competing for hardware resources and scheduled by the algorithms characterizing a node. Thus, a stochastic response time analysis determines how, from the task set on the node, a $pWCRT$ (probabilistic $WCRT$ [154]) can be derived for each task. We leave undefined the response time analysis formula used because it could be different for each node. In this sense, the $pWCRT$ accounts for the scheduling algorithms along with task re-execution, dropping, degradation, etc. that may be implemented on the

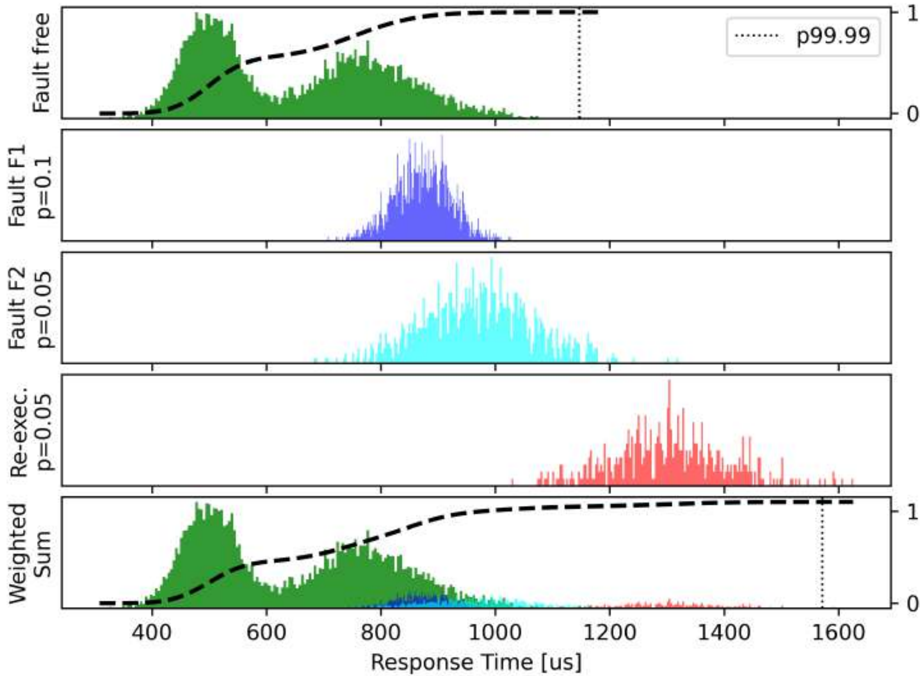


Figure 4.13: Example of combined WCRT. The fault-free execution has a long tail due to interference caused by complex hardware. F1 and F2 are two different faults that cause a fail-slow behavior. For example, F1 is a fault affecting the task itself, while F2 is a fault affecting a higher priority task affecting the preemption time. A single task re-execution may be accounted for in the WCET, assuming a single transient fault. With a single WCRT we can account for all of this weighted by its occurrence probability.

node. An example is shown in Figure 4.13

A task fails if it is not able to respect its deadline (timing failure) or it does not produce a correct output (including a wrong result, or omission and crash failures). The failure probability of a task can thus be defined as shown in (4.1), where *"correct val."* means that the task produces an output and the output is correct. Note that the event that a task does not produce a correct output can be modeled as an infinite $WCRT$, lying

in the $pWCRT$ distribution.

$$\begin{aligned} P(Failure_{\tau_i}) &= P(pWCRT_{\tau_i} > D_{\tau_i}) = \\ &= P(pWCRT_{\tau_i} > D_{\tau_i} | \tau_i \text{ correct val.}) * P(\tau_i \text{ correct val.}) + \\ &\quad + P(\neg \tau_i \text{ correct val.}) \,. \end{aligned} \tag{4.1}$$

A platform guaranteeing a high degree of isolation therefore has a lower failure probability compared to a platform that prefers workload consolidation to isolation.

The task replicas composing a microservice may present implementation diversity, and the nodes on which they are deployed may have different assurance levels. This translates to a deeply different $pWCET$, and consequently $pWCRT$, distribution for each task replica. Thus, each task $\tau_{i,j} \in \Lambda_i$ (with $j \in [0, k]$) has its own $P(Failure_{\tau_{i,j}})$.

The failure probability of a microservice depends on the fault tolerance scheme adopted. For example, assuming that the *"at least one"* scheme is adopted (i.e., the request is sent to all the task replicas, and the first response is used), the microservice fails to respond if all the task replicas in Λ_i fail to respond, as expressed in(4.2), which assumes independent failures.

$$\begin{aligned} P(Failure_{\Lambda_i}) &= \prod_{j \in [1,k]} P(Failure_{\tau_{i,j}}) = \\ &= \prod_{j \in [1,k]} P(pWCRT_{\tau_{i,j}} > D_{\tau_i}) \,. \end{aligned} \tag{4.2}$$

We plan to extend the model to include applications represented by Direct Acyclic Graph (DAG) of microservices and consider the graph response time, similarly to [155].

Finally, integrating the Omnivisor into an orchestration system using runPHI significantly enhances flexibility and reduces the amount of hardware needed to achieve the same level of system reliability by optimizing how tasks are distributed across available nodes. As a result, fewer physical machines are required, lowering costs and improving scalability without sacrificing system performance or reliability.

5

Evaluation of Nuclear Fusion Scenarios

NUCLEAR fusion is considered one of the most promising clean energy sources for the coming century, with the ITER tokamak reactor (iter.org) set to become the first fusion device to achieve net-positive energy output. As the global demand for sustainable, low-carbon electricity grows, nuclear fusion stands out as a leading technology to address this need [156].

In a tokamak (see Figure 5.1), plasma is confined using magnetic fields generated by electric currents flowing through an array of external coils. These currents are managed by the Plasma Control System (PCS) [157], a complex multi-input-multi-output control system. The PCS is composed of several sub-components, each tasked with controlling a specific plasma characteristic, with varying demands in terms of reliability, latency, and computational resources.

Vertical Stabilization. One of the essential problems to be tackled in a tokamak reactor is control of vertically unstable plasma in order to prevent it from collapsing on the reactor wall, as such an event may seriously

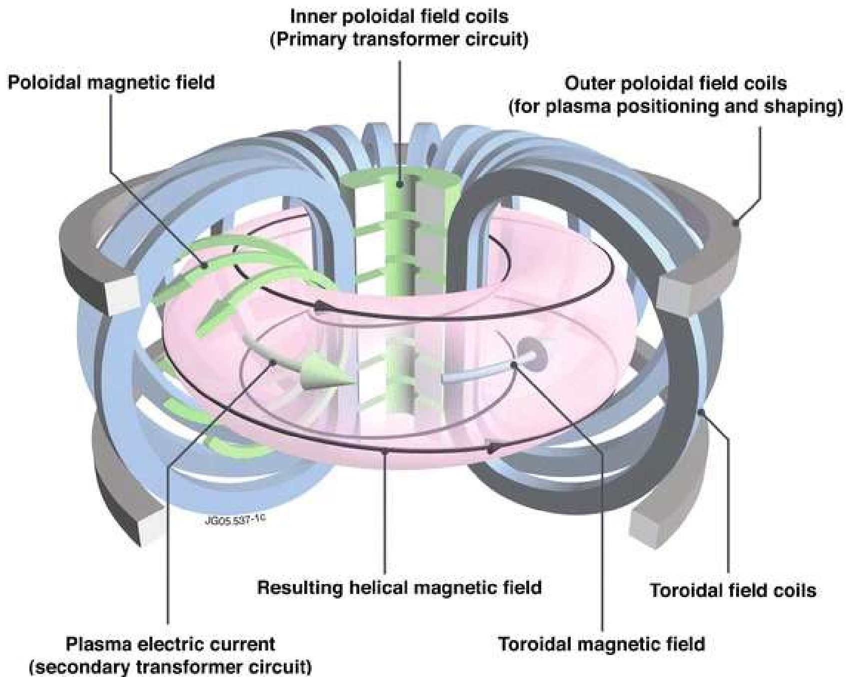


Figure 5.1: Simplified scheme of a tokamak fusion device.

damage the plant. Therefore, plasma must be controlled very precisely and any misbehavior must therefore be properly addressed [158]. Any existing solutions for the VS problem strongly depend on the value of the plasma *growth rate* γ, i.e. the value of the unstable eigenvalue associated in the plasma linearized model to the vertical instability. Adaptive control of vertical instabilities is not easily obtainable since the real-time estimation of γ is a computationally demanding task compared to the time scale in which the Vertical Stabilization (VS) should react. Therefore, the most used solution to vertical instabilities relies on model-based controllers such as the one presented in [159]. However, nowadays computational capability leads the way also to data-driven approaches, such as control based on *Extremum Seeking* [160] and on *Reinforcement Learning* [161][162] which are supposed to be tested and integrated in the near future. Clearly, such controllers suffer of higher grade of unpredictability compared to traditional model-based controllers also due to the use of

accelerators such as the GPUs and FPGA. To run and test these newly proposed approaches together with the traditional ones for safety reasons can be a good compromise between performance and predictability, and the virtualization technique proposed in this thesis can help achieve such a compromise by strongly isolating the applications on the same platform.

The ITER project plans to deploy MPSoCs [21] as near as possible to the machine as part of the computing infrastructure to execute multiple control loops such as the VS for vertical instability and signal processing algorithms—each with distinct sampling rates and reliability requirements—on the same system [78]. This makes ITER's magnetic control system a prime candidate for a Mixed-Criticality System (MCS) solution. Three key use cases that can be addressed with the proposed Omnivisor solution are:

- **[UC1] Real-Time Co-located Monitoring**: The Omnivisor architecture enables real-time sensor data monitoring alongside the control of the physical system. This enables timely detection of anomalous behaviors, enabling the system to transition to a fail-safe state if necessary.
- **[UC2] Co-running Controller Versions**: As an experimental facility, one of ITER's missions is to test advanced control schemes, such as reinforcement learning-based controllers. The Omnivisor allows complex control applications using accelerators and co-processors to run side-by-side with basic control loops for safety-critical functions.
- **[UC3]Flexible Deployment**: The Omnivisor, combined with the runPHI framework, simplifies the deployment of applications across the system. This is particularly crucial for ITER, as once the hardware is installed near the reactor, physical access will be challenging. Additionally, the framework facilitates application migration and update, leveraging unused system resources for utility applications.

The platform utilized for the evaluation phase is the ZCU already presented in Section 3.2.2 which is also used in ITER project.

This chapter evaluates the nuclear fusion control system scenarios outlined in **[UC1]**, **[UC2]**, and **[UC3]**, presenting an experimental evalua-

tion of architectural solutions based on the Omnivisor model, with a focus on deploying the Vertical Stabilization control algorithm. Section 5.1 introduces an evaluation of the model-based development process presented in Section 3.4 that unifies real-time control and monitoring software on a single platform [**UC1**]. By utilizing formal notation for the design and deployment of MPSoCs, this approach allows developers to effectively leverage embedded hypervisors for monitoring real-time applications while ensuring system predictability through hardware resource isolation. The model is applied to the ITER control system to evaluate the benefits and challenges of integrating anomaly detection monitoring alongside the control algorithm. Additionally, the Section 5.2 provides a thorough evaluation of key Omnivisor mechanisms, such as remote VM spawning and RPUGuard communication protocols. These mechanisms are tested using the Vertical Stabilization control algorithm, operating within a closed-loop configuration with an emulated plasma model. The objective is to demonstrate the feasibility of running isolated applications on the same platform, even when utilizing co-processors and accelerators in modern heterogeneous MPSoCs [**UC2**]. Finally, in Section 5.2 we also illustrate how safety-critical applications can be seamlessly integrated into a criticality-aware orchestration system, showcasing the robustness and flexibility of the proposed solution [**UC3**].

5.1 Model-Based Fog Monitoring Evaluation in Nuclear Fusion

In this section, we present and evaluate the MCS model presented in Section 3.4 in a nuclear fusion scenario that requires monitoring its application-level behavior while guaranteeing system predictability. We have considered developing prototypes of the system according to the traditional system development process.

The goals of the evaluation are:

- Showing a practical application of the model-based MCS deployment model introduced in Section 3.4;
- Showing that different architectural choices impact the predictability of the real-time scenario described in [**UC1**] differently.

5.1.1 System Description and Specification.

In this work, we are interested in monitoring the plasma during the application of the VS control algorithm. Specifically, we consider a real-time task executing a control strategy based on Extremum Seeking (ES) to vertically stabilize the plasma [10]. This experiment directly relates to **[UC1]**, where real-time monitoring applications are deployed alongside critical controllers, ensuring that the real-time behavior of the system is not compromised.

Although procedures for the assessment of the PCS performance requirements are envisaged [163, 164], no standards apply to specific safety-critical metrics for real-time control tasks when designing the PCS. However, there is significant interest in high-frequency and fine-grained monitoring of real-time control systems for performance, predictability, and safety to provide countermeasures in due time to address run-time anomalies. This is shown by the intensive exception management study being conducted in fusion projects such as ITER [165] and JET [166, 167], and by other works dealing with the monitoring of hard real-time control algorithms for performance guarantees [168, 34], or with disruption prediction systems [169, 170].

Regarding VS, this algorithm ensures that the plasma column is kept around a given equilibrium, i.e. the vertical position of the plasma current centroid is bounded in a given range during operation. Moreover, other state variables, mostly involving currents flowing throughout the plasma and actuator circuits, must follow patterns that lead to correct steady-state behavior. Figure 5.2 shows the typical response of the controlled system to a sudden disturbance, modeled as downward Vertical Displacement Event (VDE)s [120]; in particular, the time traces of the vertical position of the plasma centroid Z_c and of the plasma current I_p are shown. When VDEs exceed a given threshold, the plant may enter an operation regime that may lead to a plasma disruption, which, in turn, makes Z_c and I_p exceed normal bounds, seriously damaging the plant. Hence, prompt identification of a potentially dangerous regime should be put in place, to trigger the proper mitigation policies. This is shown in Figure 5.3, which depicts anomalous Z_c and I_p dynamics due to VDEs exceeding safety thresholds.

The set of all normal dynamics characterizes the system's normal be-

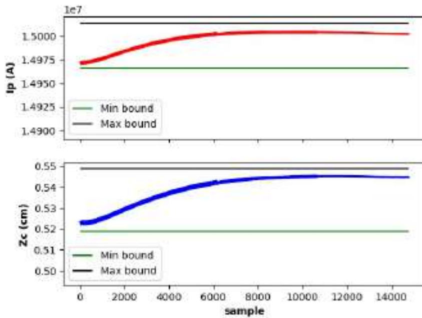


Figure 5.2: Normal behavior of the plasma current Ip and vertical position of the plasma centroid Zc.

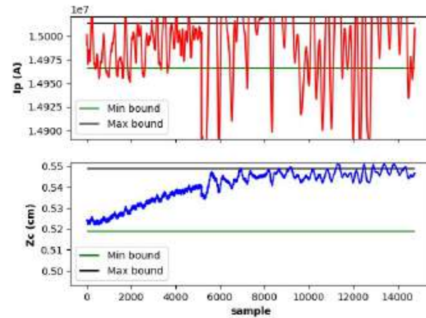


Figure 5.3: Anomalous behavior of the plasma current Ip and vertical position of the plasma centroid Zc.

havior to be used when comparing it to runtime behavior under unknown (possibly anomalous) conditions.

As Figures. 5.2 and 5.3 outline, application-level behavior is monitored through time series. This requires modeling normal behavior and checking the system at runtime for such behavior through data-driven anomaly detection for time series, of which we surveyed possible options from the literature.

Recovery actions are uniquely prescribed for each of the experimental designs in operation (e.g. ITER and JET [165, 166, 167]). In any case, the main concern for these projects is forcing the fail-safe behavior of the plant to prevent catastrophic consequences. As a result, the detection must happen within a time interval that depends on both the specific tokamak and the plasma configuration. This is to ensure that all safety shutdown processes are carried out on time so as to avoid any contact between the hot plasma and the blankets, i.e., the tokamak's walls. Indeed, a sudden shutdown can lead to faulty machine parts that are too costly and difficult to repair. However, it is worth noting that the exact procedures for plasma cooling during a fault have not yet been established for the experiments on ITER.

In order to check whether the requirements coming from fusion needs are met, such as timely application of control inputs, isolation of safety-

critical tasks from others, and fault tolerance to runtime errors and failures, there are several metrics that can be evaluated. Among these, there are:

- $WCET_\tau$, the Worst Case Execution Time (WCET) of tasks;
- $ACET_\tau$, the Average Case Execution Time (ACET) of tasks;
- $BCET_\tau$, the Best Case Execution Time (BCET) of tasks;
- TL_τ, the Tail Latency (TL) of tasks, which is the 99th percentile of the execution time of tasks;
- std_τ, the Standard Deviation (std) of tasks.

Due to the iterative nature of the system development process, it is worth noting that requirements can be refined incrementally. In fact, as the design space is explored and prototypes are developed, there may be failures that developers may initially not consider, such as crashes of critical tasks involved in, e.g., the execution of the ES algorithm during VS, application-level data sampling, or the application of recovery actions in response to anomalies. In the iteration we are considering, the concern is about the evaluation of the predictability of tasks in absence of other failures. Specifically, as outlined in **[UC1]**, this scenario demonstrates the feasibility of deploying high-frequency monitoring tasks without affecting the critical VS control algorithm.

5.1.2 System Architecture Design.

In order to comply with the requirements that were identified in the previous step, our system must have a real-time control task C, the ITER industrial plant C must control, and a set of local CPUs ($\mathrm{CPU}_{1..p,L}$) that C runs on.

Additionally, we may consider:

- A set of general-purpose tasks $G = \{G_i, i \in \{1, \ldots, n\}\}$, a real-time recovery task R, and two monitoring tasks $M = \{DS, AD\}$, in which DS concerns application-level data sampling and AD runs anomaly detection on sampled data

- A set of remote CPUs ($\mathrm{CPU}_{1\ldots q,R}$)
- A data repository to collect the nominal behavior of the ITER industrial plant
- A set of communication channels ($CT_{1\ldots s}$).

The design of the system can be driven using a plethora of models that refer to several of its different views. These models may guide designers in choosing deployments that aim to optimize a given goal function.

Taking into account the goal of evaluating changes in the system's predictability due to different deployment scenarios, we model architectural scenarios AS_0, AS_1, and AS_2 by applying the AS_MAP function of the proposed MCS deployment model (Section.3.4).

We extend each architectural scenario with a general-purpose task G deployed on ENV_L, which generates network disturbance by communicating with the Internet through a network socket. This is because a substantial amount of raw data are collected by sensors during the experiments and must be transferred to storage sources for later analysis to further assess the behavior of the plasma. Additionally, general-purpose, signal-processing tasks, which may send data to outbound servers for additional and non-urgent analysis in distributed scenarios, may use the same data collected during real-time control as well. Therefore, it is worthwhile deploying such tasks in the local environment. This leads to the d counterparts of the aforementioned architectural scenarios ($AS_{x,d}, x = \{0, 1, 2\}$), where there is the task G deployed on one of the VMs of ENV_L. We collect local and remote VMs and CPU pools, their corresponding DMs and scheduling, and the resulting quadruplets in Tables 5.1 and 5.2.

For the sake of clarity, we depict one of the architectural scenarios (AS_2) in Figure 5.4, though it is worth noting the opportunity to formalize the scenario through a well-defined mathematical language makes graphical representations superfluous, eliminating ambiguity and reducing the time required to describe several different scenarios.

The metrics previously mentioned can now be specialized for the C and DS tasks of our design:

- $WCET_C$, $BCET_C$, $ACET_C$, TL_C, and std_C;
- $WCET_{DS}$, $BCET_{DS}$, $ACET_{DS}$, TL_{DS}, and std_{DS}.

Table 5.1: Local and remote VMs, CPU pools, DMs, and scheduling schemes of the designed architectural scenario.

Local environments				
	$\boldsymbol{VM_L}$	$\boldsymbol{P_L}$	$\boldsymbol{DM_L}$	$\boldsymbol{SS_{DM}}$
$\boldsymbol{AS_0}$	$1, L : \{C\}$	$1, L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1, L : (VM_{1,L}, P_{1,L})$	$DM_{1,L} : (-, FPS)$
$\boldsymbol{AS_{0,d}}$	$1, L : \{C\}$ $2, L : \{G\}$	$1, L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1, L : (VM_{1,L}, P_{1,L})$ $2, L : (VM_{2,L}, P_{1,L})$	$DM_{1,L} : (RTDS, FPS)$ $DM_{2,L} : (RTDS, FPS)$
$\boldsymbol{AS_1}$	$1, L : \{C, R\}$	$1, L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1, L : (V, M_{1,L}, P_{1,L})$	$DM_{1,L} : (-, FPS)$
$\boldsymbol{AS_{1,d}}$	$1, L : \{C, R\}$ $2, L : \{G\}$	$1, L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1, L : (VM_{1,L}, P_{1,L})$ $2, L : (VM_{2,L}, P_{1,L})$	$DM_{1,L} : (RTDS, FPS)$ $DM_{2,L} : (RTDS, FPS)$
$\boldsymbol{AS_2}$	$1, L : \{C, R\}$ $2, L : \{DS, AD\}$	$1, L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1, L : (VM_{1,L}, P_{1,L})$ $2, L : (VM_{2,L}, P_{1,L})$	$DM_{1,L} : (RTDS, FPS)$ $DM_{2,L} : (RTDS, FPS)$
$\boldsymbol{AS_{2,d}}$	$1, L : \{C, R\}$ $2, L : \{DS, AD\}$ $3, L : \{G\}$	$1, L : \bigcup_{i=1\ldots p} CPU_{i,L}$	$1, L : (VM_{1,L}, P_{1,L})$ $2, L : (VM_{2,L}, P_{1,L})$ $3, L : (VM_{3,L}, P_{1,L})$	$DM_{1,L} : (RTDS, FPS)$ $DM_{2,L} : (RTDS, FPS)$ $DM_{3,L} : (RTDS, FPS)$

Remote environments				
	$\boldsymbol{VM_R}$	$\boldsymbol{P_R}$	$\boldsymbol{DM_R}$	$\boldsymbol{SS_{DM}}$
$\boldsymbol{AS_0}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\boldsymbol{AS_{0,d}}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\boldsymbol{AS_1}$	$1, R : \{DS, AD\}$	$1, R : \bigcup_{i=1\ldots q} CPU_{i,L}$	$1, R : (V, M_{1,R}, P_{1,R})$	$DM_{1,R} : (-, FPS)$
$\boldsymbol{AS_{1,d}}$	$1, R : \{DS, AD\}$	$1, R : \bigcup_{i=1\ldots q} CPU_{i,L}$	$1, R : (V, M_{1,R}, P_{1,R})$	$DM_{1,R} : (-, FPS)$
$\boldsymbol{AS_2}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\boldsymbol{AS_{2,d}}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 5.2: The deployment quadruplets of the designed architectural scenarios.

Deployment quadruplets				
	$\boldsymbol{ENV_L}$	$\boldsymbol{ENV_R}$	$\boldsymbol{SS}$	$\boldsymbol{CS}$
$\boldsymbol{AS_0}$	$\{DM_{1,L}\}$	$\emptyset$	$\{SS_{DM_{1,L}}\}$	$\emptyset$
$\boldsymbol{AS_{0,d}}$	$\{DM_{1,L}, DM_{2,L}\}$	$\emptyset$	$\{SS_{DM_{1,L}}, SS_{DM_{2,L}}\}$	$\emptyset$
$\boldsymbol{AS_1}$	$\{DM_{1,L}\}$	$\{DM_{1,R}\}$	$\{SS_{DM_{1,L}}, SS_{DM_{1,R}}\}$	$\{Socket, VM_{1,L}, VM_{1,R}\}$
$\boldsymbol{AS_{1,d}}$	$\{DM_{1,L}, DM_{2,L}\}$	$\{DM_{1,R}\}$	$\{SS_{DM_{1,L}}, SS_{DM_{1,R}}\}$	$\{Socket, VM_{1,L}, VM_{1,R}\}$
$\boldsymbol{AS_2}$	$\{DM_{1,L}, DM_{2,L}\}$	$\emptyset$	$\{SS_{DM_{1,L}}, SS_{DM_{2,L}}\}$	$\{SHM, VM_{1,L}, VM_{2,L}\}$
$\boldsymbol{AS_{2,d}}$	$\{DM_{1,L}, DM_{2,L}, DM_{3,L}\}$	$\emptyset$	$\{SS_{DM_{1,L}}, SS_{DM_{2,L}}, SS_{DM_{3,L}}\}$	$\{SHM, VM_{1,L}, VM_{2,L}\}$

As mentioned, these metrics drive the evaluation of the predictability of the prototypes deployed according to AS_0, $AS_{0,d}$, AS_1, $AS_{1,d}$, AS_2, and $AS_{2,d}$.

It is worth noting that access policies to shared memory are strictly

dependent on the specific protocol chosen by the designer. Since this is out of the scope of this work, we decided to use a simple asynchronous protocol, which we briefly describe in the following.

The C task asynchronously sends data to the DS task within each of its execution periods. In turn, the DS task reads the newest data sent by C from the communication channel. The most important factor is to receive the newest data possible, even if it could imply the loss of some samples. For the same reason, we decided to use the User Datagram Protocol (UDP) protocol in the scenario where network sockets are used as a communication technique. Conversely, suppose two or more applications communicate synchronously. In that case, this may result in temporal variations, which should be appropriately handled by the communicating tasks, regardless of the communication channel they send/receive data to/from.

Finally, anomaly detection, performed by task AD and based on process mining, allows characterizing the nominal behavior as locally-stored patterns and comparing new time series with such behavior at runtime [171]. However, please note that the goal of our experimentation is not evaluating the predictability of process mining algorithms, which is an open challenge on its own. Rather, in the following, we evaluate the ability of each architectural scenario to isolate the non-deterministic nature of AD. In light of this, we do not consider any metric for AD in our experimentation.

5.1.3 Prototype Development.

The prototype for each of the designed architectural scenarios employs a ZCU104 (described in Section. 2.2) as the node where ENV_L is deployed, whereas a workstation with an Intel(R) Core(TM) i7-4790 CPU, 16Gb RAM, and 512Gb HDD is used for ENV_R deployment. When virtualized, ENV_L is managed by the Xen hypervisor.

In order to generate normal and anomalous system dynamics, we used an ITER nuclear plant simulator that allows injecting VDEs. During the offline phase, VDEs are injected within normal ranges, allowing the storage of normative patterns as the ES algorithm runs and controls the plant to its equilibrium. These normative patterns are used during online monitoring and as the simulator runs together with the C task running the ES algorithm; throughout online monitoring, the nuclear plant is in-

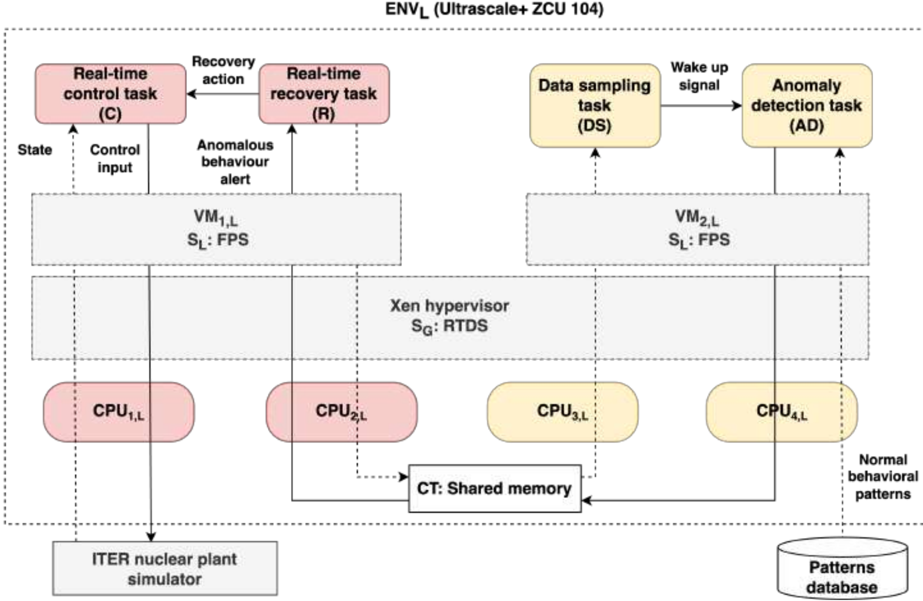


Figure 5.4: The AS_2 architectural scenario.

jected both with normal and anomalous VDEs, so as the DS task collects samples from the C task, checking whether there are anomalies or not.

In each experiment, we have performed 50 runs per architectural scenario, where, for each run, the C task is a periodic task with period $1000\mu s$ and acquires, for each task instance, a sample from the ITER nuclear plant simulator while it runs under unknown conditions. Each sample is a vector of 5 floating point values, including the previously cited I_p and Z_p state variables.

In AS_0, these samples are not sent to any other task, whereas in AS_1 and AS_2 they are sent to the DS task through a network socket and shared memory, respectively.

Once 2000 samples are collected by DS, the AD task executes and classifies the behavior according to the normal patterns that were characterized during the offline phase. For each run, a total of 60000 samples are collected from the simulated plant, thus there are 60000 execution times collected from both the C and DS tasks.

Table 5.3: Metrics per scenario related to tasks C and DS (N/A: Not Applicable).

	$WCET_\tau(\mu s)$	$ACET_\tau(\mu s)$	$BCET_\tau(\mu s)$	$TL_\tau(\mu s)$	$std_\tau(\mu s)$
AS_0	C : 14.00 DS : N/A	C : 13.07 DS : N/A	C : 11.00 DS : N/A	C : 14.00 DS : N/A	C : 0.90 DS : N/A
$AS_{0,d}$	C : 15.00 DS : N/A	C : 13.41 DS : N/A	C : 11.00 DS : N/A	C : 15.00 DS : N/A	C : 0.91 DS : N/A
AS_1	C : 43.00 DS : 14.00	C : 41.87 DS : 3.94	C : 11.00 DS : 1.00	C : 43.00 DS : 13.00	C : 0.97 DS : 3.27
$AS_{1,d}$	C : 69.00 DS : 33.00	C : 45.29 DS : 6.24	C : 12.00 DS : 1.00	C : 67.00 DS : 27.00	C : 4.12 DS : 6.31
AS_2	C : 15.00 DS : 3.00	C : 13.09 DS : 1.38	C : 10.00 DS : 1.00	C : 14.00 DS : 3.00	C : 0.88 DS : 0.59
$AS_{2,d}$	C : 20.00 DS : 15.00	C : 15.01 DS : 4.28	C : 11.00 DS : 1.00	C : 19.00 DS : 14.00	C : 1.85 DS : 3.28

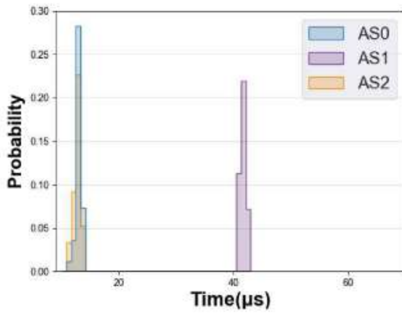
Table 5.3 collects the metrics per scenario. These are computed globally, which means all execution times from all runs are considered at once and all metrics are computed accordingly. For instance, in order to compute $WCET_C$ for scenario AS_0, execution times from all 50 runs are collected and the maximum is computed.

In Figure 5.5 the metrics per scenario are visualized as histograms to highlight how the distributions of execution times of tasks C and DS shift from one setup to another.

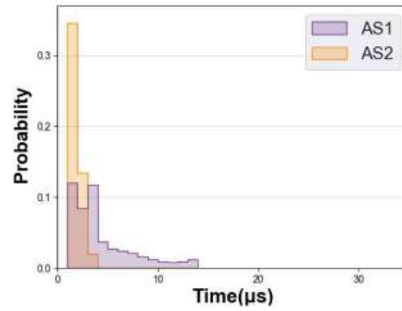
Figs. 5.6a and 5.6b show violin plots of tasks C and DS per scenario in order to provide another view of the results with a greater focus on the variability of the data across scenarios.

The results presented show that each architectural scenario impacts the predictability of the system differently. As shown in Figs.5.5 and in Figure 5.6, the execution times are generally higher when data between C and DS are sent across the network (AS_1). Furthermore, network disturbance impacts the predictability of the MCS slightly more when deploying network sockets for inter-VM communication (AS_1) than hypervisor-managed shared memory (AS_2).

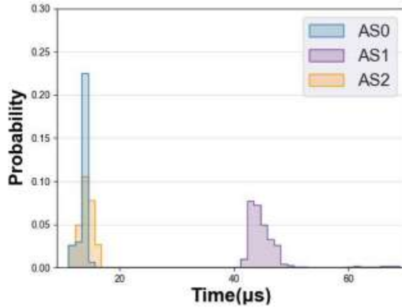
Firstly, the observed $WCET_C$ and $ACET_C$ in each scenario AS_1 and AS_2 (see Figure 5.5a) suggests that hypervisor-based shared memory (AS_2) should lead to execution times that are considerably closer to the non-



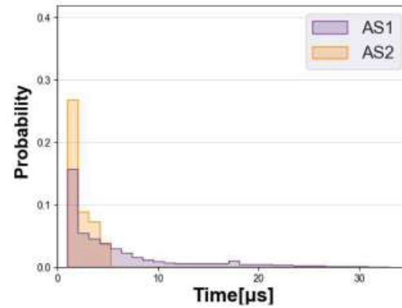
(a) No network disturbance.



(b) No network disturbance.



(c) With network disturbance.



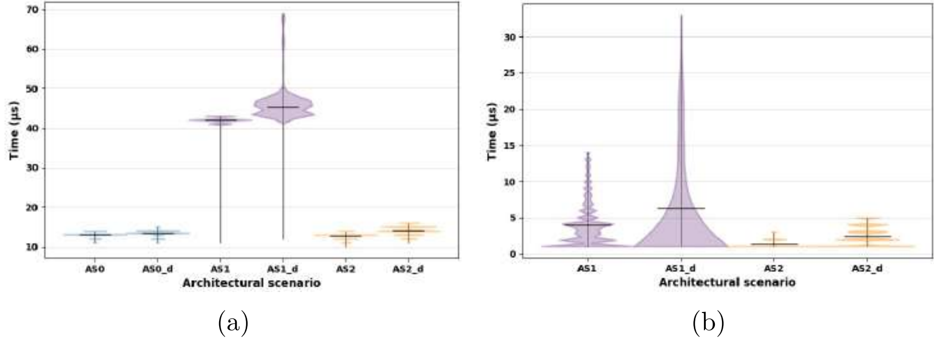
(d) With network disturbance.

Figure 5.5: Density function plots of execution times of tasks C (a), (c) and DS (b), (d), per scenario.

virtualized baseline (AS_0). In fact, compared to the baseline, shared memory only worsens $WCET_C$ and $ACET_C$ by 7.14% and 0.15% (AS_2), whereas network sockets worsen $WCET_C$ and $ACET_C$ by 207.14% and 220.35% (AS_1).

Secondly, as Table 5.3 highlights, there are $WCET_C$ shifts when adding network disturbance to all architectural scenarios (see Figs. 5.5a and 5.5c). Specifically, network disturbance make $WCET_C$ worse by 7.14%, 60.46%, and 33.33% in architectural scenarios AS_0, AS_1 and AS_2, respectively. Similarly, network disturbance makes std_C worse by 1.11%, 324.74%, and 110.22%.

Furthermore, by directly comparing scenarios AS_1 and AS_2 under net-



(a) (b)

Figure 5.6: Violin plots of execution times of tasks C (a) and DS (b), per scenario.

work disturbance we can notice an increase in time variability when network sockets are used. In fact, the value of std_C from scenario AS_2 to AS_1 worsens by 122.7%. Clearly, such worsening is due to the interference of tasks when accessing shared network resources.

It is worth noting that, as Figs.5.5b and 5.5d show, also execution times of DS, both with and without network disturbance, show less variability when deployed as a VM on the same board where C is running (AS_2) compared to the scenario where these two tasks communicate over the Internet through network sockets (AS_1). Specifically, in case of network disturbance, the value of std_{DS} in AS_1 is 92.37% higher than the one in AS_2.

Although shared memory is the best option with respect to reducing WCET and std worsening of tasks, virtualization still causes a slight variation in execution times in case of network disturbance, regardless of the deployed communication channel between C and DS. This is because, despite resource partitioning and deployment of real-time schedulers, there still is interference caused by both context switches between VMs and interrupt handling.

Despite hardware support for virtualization speeds up the process due both to the presence of a hypervisor-dedicated execution mode, such as the ARM Exeption Level (EL)3, and a set of additional registers, context switches still lead to non-negligible delay.

Furthermore, each time a device sends an interrupt, as the network

device deployed in our experiment does, the hypervisor intercepts it to determine which VM has to serve it. This behavior causes a delay to the running VM. Though modern hardware interrupt managers are extended to support virtualization allowing direct access to VMs when possible; The ARM Generic Interrupt Controller (GIC)v4, currently, is limited to Message Signaled Interrupt (MSI)s and IPIs and does not avoid triggering the hypervisor.

Therefore, in spite of the problems we have outlined, and given the experimental results, we believe that virtualization is a good choice to implement high-frequency co-located monitoring of control algorithms via shared memory **[UC1]**, as non-virtualized communication via network sockets not only impacts the predictability of the system more, it also makes performance worse due to use of the software network stack and the more intensive use of the I/O.

5.2 Omnivisor Experimental Validation

In this section, we provide an evaluation of the Omnivisor model and implementation both with standard benchmarks and in Nuclear Fusion scenarios. The reference implementation, along with a set of scripts to reproduce the basic experiments, is openly available as open-source software [172]. The platform under test is the ZCU described in Section 2.2. The evaluation aims to address the following questions:

- *Is the boot time of a VM on a remote core comparable to that on main cores? If so, is it possible to flexibly and dynamically reconfigure the system within a reasonable time?*
- *What degree of spatio-temporal isolation does the Omnivisor guarantee for VMs on main and remote cores?*
- *Can the Ominvisor be a turnkey solution to achieve controlled degradation?*
- *Can the Ominvisor guarantee the isolation in a real scenario where a high-frequency vertical stabilization controller runs alongside strong disturbance running on processors, co-processors, and accelerators?*

It's important to note that the additional functionalities introduced in our Omnivisor, as described in Section 4.2.2, are only invoked during the startup of newly created VMs, not at runtime. Therefore, the overhead of Omnivisor is consistent with prior findings on Jailhouse [55].

5.2.1 Boot Time Performance Assessment

This section shows that booting a VM on a remote core using Omnivisor is comparable in time to booting a VM via Jailhouse on a main core. Thus, with Omnivisor, users can deploy VMs on either main or remote cores with negligible differences in boot times, enabling flexibility for scenarios like real-time migration [173], reboot after failure [174], system rejuvenation [175] and Over-The-Air (OTA) updates [176, 177].

Figure 5.7 shows the boot times obtained by deploying a VM on RPU and RISC-V soft-core, using the Omnivisor, compared to the boot time on APU using vanilla Jailhouse. In each case, the binary contains the identical bare-metal application. However, running the application on the APU with the Jailhouse hypervisor necessitates linking a tiny 'inmates' library for initialization whose overhead is negligible during boot times. To obtain the boot time values, the root-cell acquires the initial value from a global platform timer just before initiating the new cell (Create). The same timer is used to measure the length of the load sequence (Create + Load). Finally, the newly started cell captures the third timer sample (Create + Load + Start), representing the boot time, and records it in a shared memory page. The described process has been repeated 100 times for ten different VM image sizes, specifically from 1 to 90 megabytes. It is possible to observe in Figure 5.7 in more detail the three phases that comprise the boot times: create (blue line), load (orange line), and start (green line).

We first compare the boot times of a cell on APU and RPU. The results exhibit significant similarity, indicating that starting a VM on a microcontroller-level CPU does not result in performance losses. On the contrary, the RPU boot shows a slight speed advantage during the configuration phase. This difference arises because the APU needs to reorganize the page tables for the new cell, while the RPU does not use page tables. However, the final boot time is quite similar, partially due to the lower frequency of the RPU (600MHz) compared to the APU (1.5GHz), leading

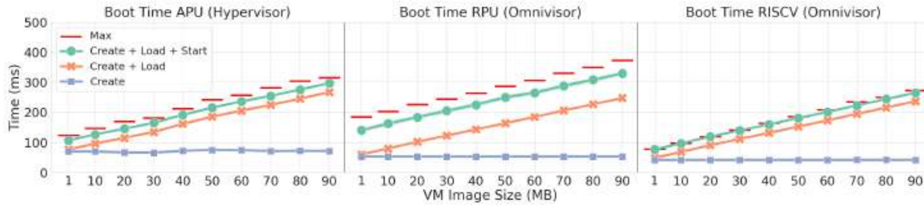


Figure 5.7: Comparison of boot times across heterogeneous processors in the ZCU Platform

to the longer RPU wake-up procedure that involves both the PSCI and the PMU, as detailed in Section 4.2.3. The results for the RISC-V soft-core exhibit similar creation and loading times as the RPU, as there is no necessity for configuring the page tables in either case. Nonetheless, the boot time is notably faster. Despite the soft-core lower frequency (100Mhz), the boot time disparity arises because the soft-core is always powered on in the FPGA. Removing it from its reset mode via the FPGA's configuration port is a fast operation compared to the RPU boot procedure.

In the ITER project, a fusion experiment enforces multiple stages of the plasma: ramp-up, flattop, and ramp-down [119]. Each stage requires different controllers to effectively manage the plasma. Leveraging the Omnivisor ensures flexibility in dynamically reconfiguring these controllers, deployed as VMs on the board, by rebooting them on main or remote cores.

5.2.2 Omnivisor's Isolation Capability

To demonstrate the Omnivisor isolation capabilities we initially highlight the vulnerabilities that arise when executing unprotected code across various cores within an MPSoC. Subsequently, we activate the Omnivisor with solely spatial protection mechanisms. Finally, we show the effectiveness of the full-fledged Omnivsior by enabling also temporal isolation. This test setup is representative of **[UC2]**, where the ability to deploy and isolate multiple versions of safety-critical controllers on the same platform aids in testing and redundancy.

Experiment Setup. Figure 5.8b (left) depicts our experimental setup: we run a VM under test both on RPU-0 and RISC-V soft-core, while other

managers, such as the APUs, the other RPU (RPU-1), and the FPGA, create interference by accessing the memory area owned by the VMs under test. The deployed application is the same on both remote cores. It involves a simple periodic task that reads an array from memory, calculates the sum of its values, and then writes the result back into memory at a different location. The only difference is that, due to the frequency difference between the soft-core (100Mhz) and the RPU (600Mhz), the matrix used in the soft-core application is smaller than that used in the RPU application to ensure comparable results in terms of execution time. The RPUs are configured with disabled caches and operate in split mode, where RPU-0 operates independently from RPU-1. The RISC-V soft-core is deployed on FPGA without any cache. Since the experiments are the same for both VMs under test, we will generally refer to both cores as *"rCPU"* for simplicity.

To assess the isolation capability of the Omnivisor against the vanilla Jailhouse hypervisor, we augment the Jailhouse hypervisor with minimal code necessary to execute applications on remote cores, a functionality not available by default. This enables us to compare the isolation achieved using Jailhouse alone with those obtained using an Omnivisor.

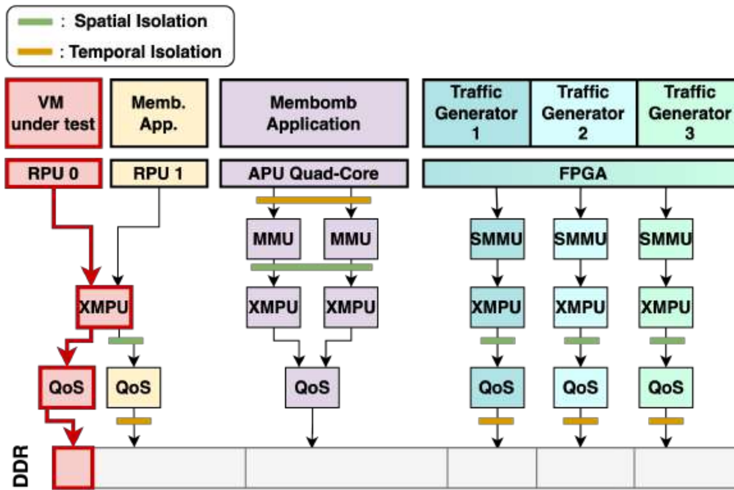
Employing a traditional hypervisor, as illustrated in Table 5.8a, will cause applications running on remote cores (RPU, RISC-V) to experience failures when other managers access their memory (e.g., RPU1 and FPGA). Conversely, when the Omnivisor extension is enabled, we expect these applications to continue functioning without failures.

Spatial Isolation Evaluation. The results in Figure 5.9 show that without explicitly programming the isolation layer, a manager can break both spatial and temporal isolation of VMs. In the test, the VM under test starts on one *rCPU* and, after two seconds of execution, an interfering application starts on one of the other managers. We repeat the test using Jailhouse vanilla (no protection mechanisms) and using the Omnivisor extension first with only spatial isolation and then the full-fledged version with temporal isolation too.

The interference application deployed on the APU is the well-known IsolBench *bandwidth* benchmark [178] from the RT-Bench framework [179]. The test is launched on three APU cores out of four and it reaches a utilization factor close to 1 on each processor. The free core is used to launch

Interference	Hypervisor	Omnivisor
APU(□)	No Failure	No Failure
RPU-1(□)	Crash	No Failure
FPGA(□, □)	Crash	No Failure

(a) Fault behavior of a VM under test



(b) Architectural View of the experiments

Figure 5.8: Expected fault outcomes (top) and experimental configuration (bottom) of VMs running on rCPU subjected to interference from various sources (APU, RPU-1, FPGA): a comparative analysis between traditional Hypervisor and Omnivisor.

the scripts, start the tests, and save the results. On RPU-1, we deploy a synthetic bare-metal application mirroring the bandwidth benchmark behavior. Finally, in the FPGA, we deploy two instances of the AXI traffic generator IP from Xilinx [180].

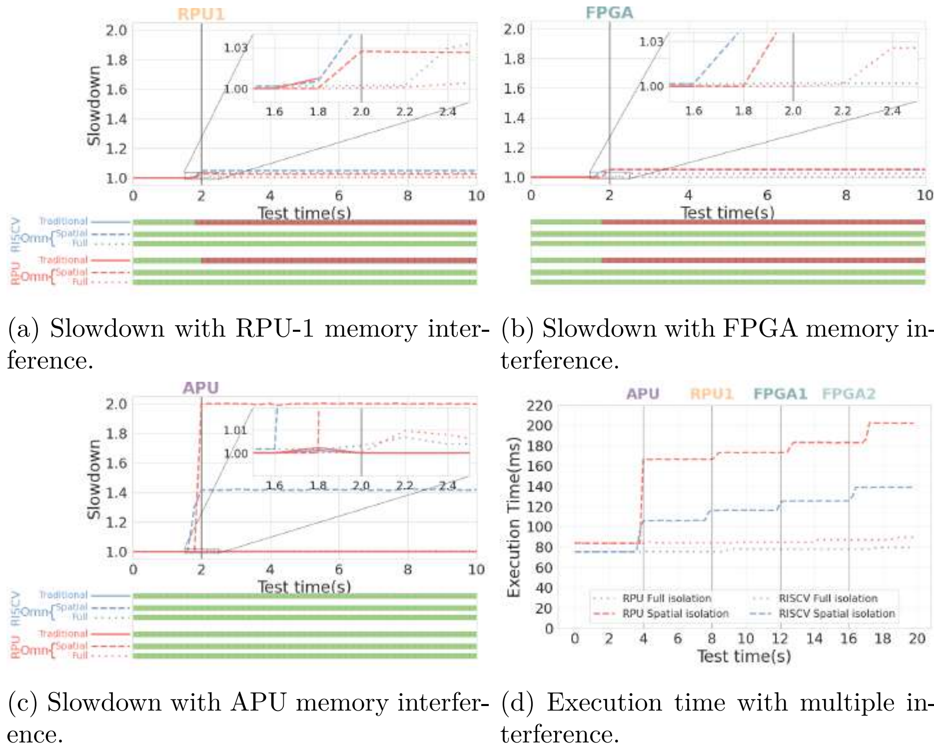
We can observe the results when the RPU-1, FPGA, and APU managers are the source of interference in Figs. 5.9a, 5.9b, and 5.9c respectively. The lower bars represent the execution state of the VM, where

green indicates that the application is running, while red denotes that the application has failed. We can observe that, without the Omnivisor, all managers can break the spatial containment of the cell, causing the virtual machine to fail except when the APU is the source of interference (5.8a). This is because the Jailhouse hypervisor already uses the MMU to protect the memory areas of the VMs. Since the APU is the only manager that accesses memory using the MMU, it is also the only one for which spatial permissions are enforced with traditional hypervisors. Notably, the access of the cell running on the APU to the memory belonging to a different cell causes the APU-bound VM to be shut down by the hypervisor while the latter continues undisturbed. That is the reason why, in Figure 5.9c, the execution time of the VM under test is not impacted when the vanilla hypervisor is deployed.

To run this evaluation, we have integrated the code to run VMs on remote cores in Jailhouse. Without this upgrade, launching an application on remote cores at run-time is possible with the `remoteproc` driver [181]. In that case, since the hypervisor has no vision of the memory used by the RPU, it would not offer any form of isolation, leading to a fallback in the same failing scenario, even when the APU is the source of traffic.

In real-world scenarios, like the ITER project, diverse applications, often developed by separate groups, introduce the potential for bugs that can adversely affect other components. For instance, as described in **[UC2]**, a control application running on the RPUs might inadvertently overwrite memory used by APUs for a second safer control algorithm, resulting in the possible loss of control during expensive experiments. Thus, the containment level provided by the Omnivisor emerges as a crucial feature, mitigating the risk of system failures caused by the malfunction of individual applications and facilitating seamless integration.

Temporal Isolation Evaluation. Figure 5.9d illustrates the temporal behavior of the periodic task running on the rCPU when all other managers access the memory. Every four seconds, a new manager is activated and starts creating contention over the memory communication channels. From the result, it is clear that hardware mechanisms such as MMU, SMMU, and XMPUs/XPPUs can provide spatial isolation between VMs but cannot guarantee temporal isolation and, therefore, cannot ensure real-time performance. This is intuitively due to the resources that



(a) Slowdown with RPU-1 memory interference.

(b) Slowdown with FPGA memory interference.

(c) Slowdown with APU memory interference.

(d) Execution time with multiple interference.

Figure 5.9: Execution time slowdown of simple periodic task running in a VM over both RPU-0 and RISC-V soft-core. The behavior of the applications under different sources of interference is shown first when using a plain jailhouse, then a partial Omnivisor implementation only with spatial isolation mechanisms enabled, and finally the full Omnivisor implementation.

are still shared on board, such as the bus and the memory controller. Therefore, temporal isolation can be enforced using mechanisms for bandwidth regulation.

As discussed in Section 4.2.3, the Omnivisor provides the knobs to regulate the memory bandwidth of different managers in the system by leveraging a QoS and MemGuard implementation. Since there are already papers exploring these mechanisms in detail [59, 52, 60], in this experi-

ment, we are interested in demonstrating that the Omnivisor can use these mechanisms to reduce the temporal interference caused on a VM running on remote cores.

To isolate the VM running on rCPU from the other managers, the Omnivisor first configures the QoS for the FPGA and RPUs channels. In this experiment, each channel has a request rate bounded to 11, which, using the formula from [60], translates to a memory bandwidth of 4.7 MB/s. Regarding the APU, on the other hand, we enabled a MemGuard regulation of 78 cache refills each millisecond for all the cores, which corresponds to having 4.997 MB/s of available bandwidth. Combining the two approaches strongly reduces the performance impact on the rCPUs, as shown in Figure 5.9d. Specifically, the maximum slowdown drops from 142% to 7% on RPU and from 85% to 6% on RISC-V.

Integrating state-of-the-art monitoring and profiling applications into real safety-critical systems is often sidestepped in favor of legacy methods. This hesitation primarily stems from the difficulty in demonstrating that these applications don't disrupt the temporal behavior of the critical application under observation. However, with the Omnivisor, integrating such mechanisms becomes significantly easier, thanks to the utilization of a fully temporally isolated VM running on remote cores.

Parameter Tuning for Controlled Degradation

To comprehensively evaluate and demonstrate the usability of the Omnivisor beyond synthetic benchmarks, we execute a realistic benchmark suite on the remote cores. Specifically, our choice has gone towards using the benchmark set called TACLeBench provided in [140]. It is a collection of 56 benchmark programs from several research groups and tool vendors worldwide. However, while we were able to execute all the benchmarks on the RPU, due to the limitations related to the absence of a floating-point extension of the RISC-V processor (Pico32) deployed on FPGA, we used a subset of them for our RISC-V experiments.

The objective of this evaluation is twofold: first, to demonstrate how the Omnivisor can induce controlled degradation in the execution time of a VM running on remote cores, and second, to elucidate how the Omnivisor streamlines the parameter tuning process for achieving an acceptable performance degradation level. Therefore, we first determine the band-

(a) Benchmarks on VMs running on RPU.

(b) Benchmarks on VMs running on RISC-V.

Figure 5.10: Comparative evaluation of TACLeBench: The bar plots depict the execution time slowdown with and without temporal constraints. Each benchmark showcases the bandwidth limitation imposed on other managers to achieve the desired 20% maximum degradation on the VM.

width allocation required to ensure unrestricted memory transactions on every manager. Specifically, leveraging findings from [60] and experimental evaluations, we established that a bandwidth limit of 950 MB/s for each manager is sufficient to maintain a comparable rate of memory transfers to what was observed without regulation. Then, using these values as a starting point, we developed a script iterating the execution of the benchmarks employing a binary search algorithm to calculate the bandwidth allocation parameters. Specifically, we search for those parameters

that ensure a maximum slowdown of 20% for each benchmark. Still, the script is generic and can be used to find the parameters for any value of degradation. The slowdown is calculated in comparison with the observed maximum execution time over thirty repetitions of the benchmarks without any interference. Furthermore, in between each change of parameters, we execute thirty repetitions and consider the worst result as the target value for the slowdown; when the target value is below the decided threshold, we consider the bandwidth allocation quota used in that iteration as a possible candidate. However, we stop the binary search after 15 iterations or when the slowdown is strictly between 19% and 20%. Figure 5.10 presents the slowdown over thirty repetitions for each benchmark under two scenarios. First, the slowdown without any bandwidth regulation is depicted. Next, the case where the bandwidth is configured to incur at most a 20% degradation is shown. In the same figure, we can also observe the level of bandwidth regulation in MB/s applied to obtain the controlled degradation for each benchmark. The results demonstrate that it is possible to achieve the desired slowdown even when the unconstrained slowdown exceeds 350%, provided you are willing to significantly constrain the rest of the system (e.g., max bandwidth limit of 4 MB/s).

Naturally, given specific application constraints, an ad-hoc policy that chooses the parameters based on the importance of the VMs can be implemented to further improve the utilization of the cores while maintaining the real-time guarantees of critical applications [52].

5.2.3 HIL Evaluation of Vertical Stabilization Controller

To evaluate the isolation guarantees provided by Omnivisor in a realistic nuclear fusion scenario, we deploy a hardware-in-the-loop (HIL) setup (see Figure 5.11) to emulate plasma behavior and control its vertical instabilities. Specifically, a dSPACE SCALEXIO target [182] is used to emulate the complete ITER plasma model as defined in [159]. The emulator generates plasma position and current data and transmits it via UDP to a ZCU platform (see Section 2.2). On the ZCU platform, the vertical stabilization control algorithm [159], running at a frequency of 5 kHz, is implemented using the Linux-based version of the MARTe2 framework (described in Section 2.1.2). Based on the received plasma data, the controller calculates and sends back the required tension adjustments.

Figure 5.11: Hardware-in-the-loop Simulation Setup

The goal of this experiment is to assess the isolation capabilities of Omnivisor when running the controller alongside memory-intensive workloads on CPUs, co-processors, and accelerators. This setup is compared to a baseline scenario without a hypervisor and without any memory bandwidth regulation. We aim to demonstrate that deploying MCSs) in complex cyber-physical systems CPS, like nuclear fusion control, is feasible when the software is fully isolated in VM and the memory bandwidth is correctly regulated.

To evaluate the controller behavior, a gain of 2000V is injected into the input tension value received by the simulator every three seconds. The controller must correct the instability and restore the plasma's position to its nominal state. Figure 5.12 shows the plasma model's output (current and vertical position) and input (tension), indicating successful stabilization despite the injected tension gain. In this scenario, the plasma vertical position is subject to a shift of at most 1.8cm that is not enough to lose the plasma control.

MARTe2 Controller on Vanilla Linux. In the first scenario, the controller is executed on a standard ARM Linux system without Omnivisor and without co-located applications. The time between receiving and sending packets is measured as the controller's execution time, using *tcpdump* to capture the data. As shown in Figure 5.13, while the average execution time is within the required range, Linux's non-real-time nature leads to sporadic deadline misses. Since the control loop must respond at a 5 kHz frequency, any execution time exceeding 200 microseconds results in a missed control signal, as indicated in the figure.

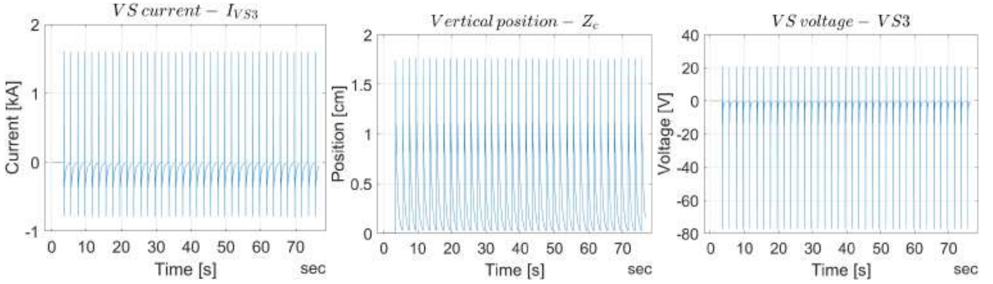


Figure 5.12: Input and Output of the simulated plasma model with periodic injected tension gain. In this scenario, the controller is able to stabilize the plasma.

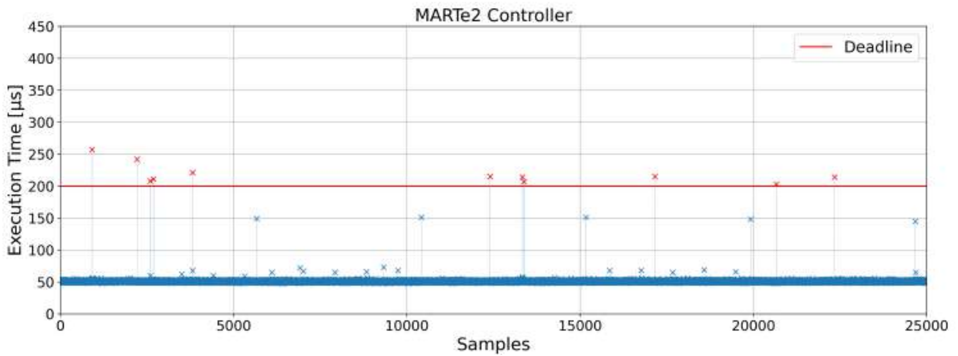


Figure 5.13: Execution time of MARTe2 Controller running on a Linux Vanilla without any co-located workload.

MARTe2 Controller on Real-Time Linux. In the second scenario, we try to remove the Linux interference. To do so, we patch Linux with *PREEMPT-RT* and isolate a specific core for the MARTe2 controller using the *isolcpu* feature of Linux. Using this feature we avoid the periodic Linux timer interrupting the controller and all the Linux services are managed by non-isolated cores [183].

This setup achieves more consistent results, with no missed deadlines. However, as shown in Figure 5.14, the average execution time is slightly higher due to the MARTE2 application being restricted to a single CPU.

As a result, even if the controller can stabilize the plasma in due time, the board remains highly underutilized, and only one core is used in an MPSoC having four application cores, two real-time cores, and various accelerators.

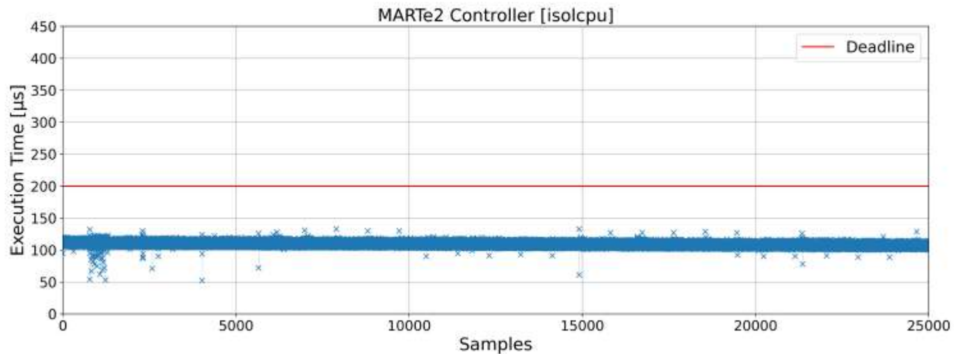


Figure 5.14: Execution time of MARTe2 Controller running on a Linux PREEMPT-RT and isolated on a single cpu without any co-located workload.

MARTe2 Controller with co-located applications. Next, we simulate a MCS by deploying the Jailhouse hypervisor, where the controller is the high-criticality task. The controller is placed in a Linux PREEMPT-RT patched VM with two cores. The MARTe2 application, as in the previous scenario, is isolated on one of the cores using *isolcpu* feature, while Linux services run on the remaining core. Additionally, we deploy two memory-intensive *membomb* applications in separate VMs on the APUs, each with one core, while running a similar *membomb* application on the RPU and a traffic generator on the FPGA.

As shown in Figure 5.15, without any memory bandwidth regulation, the shared resources such as cache and memory controller lead to significant interference between applications, resulting in multiple missed deadlines for the controller, despite its isolation on a dedicated CPU core.

In this scenario, the controller fails to stabilize the plasma, causing the vertical position to shift nearly 400cm in the simulation as shown in Figure 5.16. Since a deviation of just 15 cm is sufficient to lose plasma

control and impact the coils, this situation could potentially lead to severe damage to the reactor.

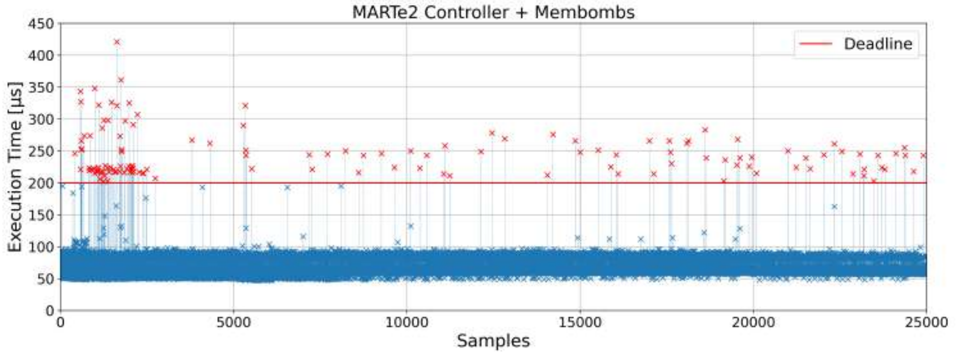


Figure 5.15: Execution time of MARTe2 Controller running on a Linux PREEMPT-RT VM on Jailhouse hypervisor and isolated on a single cpu. The VM runs with membomb co-located workloads on other VMs on APUs, RPUs, and FPGA.

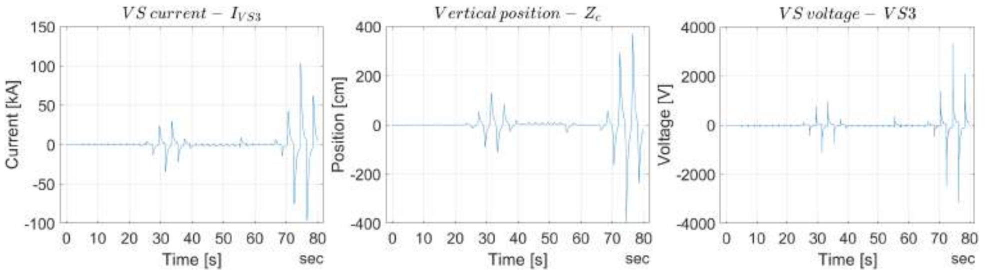


Figure 5.16: Input and Output of the simulated plasma model with periodic injected tension gain. In this scenario, the controller is not able to stabilize the plasma in time due to the deadline misses.

MARTe2 Controller with Cache Coloring. In the fourth scenario, we attempt to mitigate interference between VMs using the cache-coloring technique discussed in Section 2.2. By dedicating a portion of the cache to

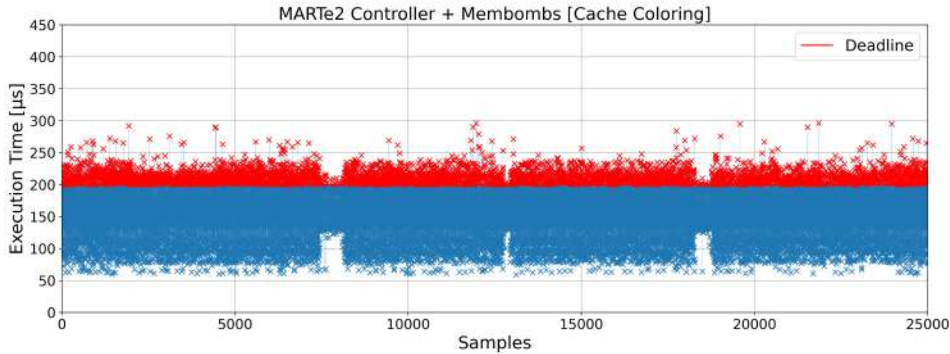


Figure 5.17: Execution time of MARTe2 Controller running on a Linux PREEMPT-RT VM and isolated on a single cpu on Omnivisor. The VM runs with membomb co-located workloads on other VMs on APUs, RPUs, and FPGA. The Omnivisor employs cache protection through cache-coloring but it doesn't provide bandwidth regulation.

the MARTe2 application, we aim to improve predictability, as other VMs running on the APUs cannot evict cache lines used by the critical VM. However, as shown in Figure 5.17, the results are actually worse in terms of deadline misses. The MARTe2 application is unable to cache all necessary data, leading to continuous cache line replacements. This process is negatively impacted by the membomb application due to contention at the memory controller level.

To avoid introducing latencies on the APUs, we opt not to use the memguard mechanism from the previous section. Instead, we leverage Omnivisor's capability to spawn VMs on remote cores and regulate memory bandwidth using one RPU. Specifically, we utilize the *mempol* mechanism proposed in [51, 184] to manage memory bandwidth for the APUs, along with QoS for the RPUs and FPGAs. In this setup, we allocate 5 MB/s to non-critical cores, leaving the remaining bandwidth for the controller. This approach yields good, predictable results, as depicted in Figure 5.18.

MARTe2 Controller with Omnivisor. However, even in this scenario, execution times are still not as optimal as in cases without co-

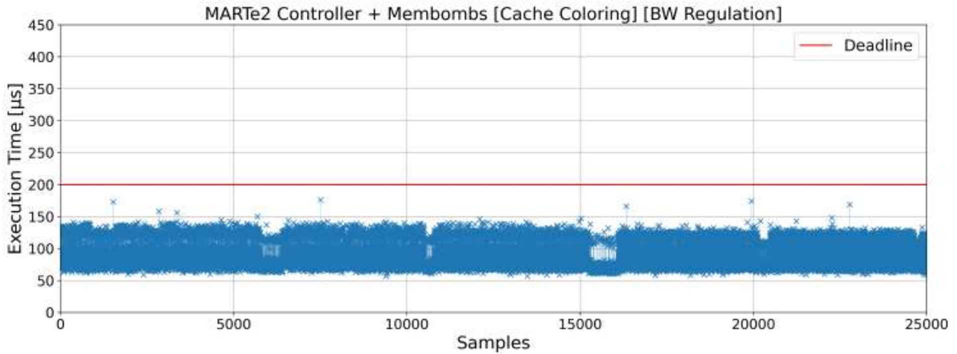


Figure 5.18: Execution time of MARTe2 Controller running on a Linux PREEMPT-RT VM and isolated on a single cpu on Omnivisor. The VM runs with membomb co-located workloads on other VMs on APUs, RPUs, and FPGA. The Omnivisor employs cache and memory bandwidth protection through cache-coloring and mempol regulation

located applications. Two main factors contribute to this. First, cache partitioning reduces the available cache for the controller, leading to more memory transactions. Second, the controller runs on Linux, which even with the PREEMPT-RT patch and the *isolcpu* feature can still introduce timing inconsistencies.

While the cache limitation can be addressed by allocating more cache to the critical application or optimizing the controller itself, the second issue can be mitigated by using the bare-metal version of MARTe2 [9]. However, the bare-metal version does not currently run as a VM in the jailhouse hypervisor. Future work will focus on integrating this bare-metal framework into Omnivisor, supporting execution on both APU and RPU.

In this way, it becomes feasible to deploy and test a second, more complex controller alongside the primary one, as described in **[UC2]**, with the assurance that the two controllers remain fully isolated from each other. This isolation ensures that any interference is minimized, allowing the testing phase to be accelerated, as the new controllers can be evaluated in a controlled environment without affecting the principal controller's performance.

5.2.4 RPUGuard Application and Validation

Until now we have considered only scenarios where RPUs are used to run isolated software with real-time performance. As demonstrated in the last section, this is possible by using the hardware/software management of protection mechanisms guaranteed by the Omnivisor. However, other important scenarios in Nuclear Fusion may consider the use of the RPUs as co-processors that provide real-time services to the applications running on the APUs. In this case, the communication between the VMs and the RPU needs to guarantee real-time performance. For this reason, we have developed the RPUGuard mechanism described in Section 4.4.

A key use case in the context of plasma magnetic control that can benefit from this technology involves the estimation of plasma parameters critical for control, many of which cannot be directly measured. As a result, computationally intensive (and often iterative) reconstruction algorithms are required to estimate these parameters. One such example is the estimation of the plasma growth rate γ which is essential for deploying adaptive VS systems. During a tokamak discharge, the plasma configuration changes, which in turn alters the linearized model that describes the plasma's behavior. Adaptive VS solutions rely on mechanisms that estimate γ, using a linearized plasma model. This estimation requires the execution of equilibrium reconstruction codes and corresponding linearization procedures during the discharge [185].

Due to the high computational demands of this task, it should ideally be performed on the APU, potentially with assistance from GPUs or FPGAs, while ensuring it does not interfere with real-time control operations. A plausible scenario involves two isolated VMs running on the APU. The first VM pre-processes data to compute current and velocity values, while the second VM calculates the growth rate (γ).

The first VM must transmit data to the VS controller on the RPU with hard real-time guarantees. Meanwhile, the second VM sends its data as soon as it completes the calculations, without stringent timing requirements. Crucially, the data from the second VM must not interfere with the real-time transmission from the first VM, ensuring that the time-sensitive control data remains unaffected.

The reference platform for this experimental section is also the ZCU (see Section 2.2). On the APU, we decided to deploy the official Linux

kernel provided by Xilinx (linux-xlnx) patched with the PREEMPT-RT patch, in order to reduce as much as possible the latency introduced by the operating system. On RPU, we run freeRTOS which is a widely used open-source real-time operating system.

We first show the bandwidth limitation of OpenAMP as a starting point for our experiments. In Section 5.2.4 we measure the message latency due to interference from parallel communication, pointing out the problems caused by exceeding the band boundary. Finally, in Section 5.2.4 the benefits of RPUGuard are shown and the results are discussed.

OpenAMP Bandwidth Limitation

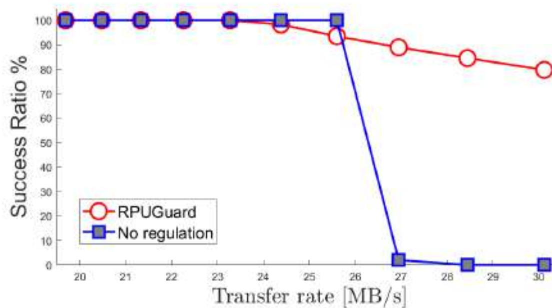


Figure 5.19: Transfers success ratio

Before testing the isolation of the communication channels, we experimentally tried to figure out the maximum bandwidth that OpenAMP can achieve using a single channel and a single endpoint, and which is its behaviour under stress. Given a fixed amount of memory dedicated to OpenAMP's virtqueues (262144 bytes, that is, the default OpenAMP virtqueues' size), we verify that if the frequency of message requests grows beyond a certain maximum limit, the queues fill up and messages begin to show high latency. The objective is to measure the maximum frequency of message transmission above which high latency occurs, for a given virtqueue size. To find this frequency, we implemented a periodic task on Linux that sends messages to the RPU using OpenAMP. The RPMsg message size is fixed by design to 512 bytes, where 496 bytes are used

for the payload and 16 bytes are used for the header. Therefore, to measure the maximum bandwidth, each sent message fills the payload to its maximum in all experiments. Therefore, the transmission bandwidth is calculated by dividing 512, that is the number of sent bytes, by the send frequency in microseconds. As an example, if the frequency of activation of the task is $20us$ then the used bandwidth is $BW = \frac{512}{20} = 25.6$ MB/s.

For our experiments, we consider the success ratios as the number of messages that are delivered in expected time within a time interval equal to two seconds. As we can see in Figure 5.19, when the transfer rate is greater than 25.6 MB/s OpenAMP, without any regulation, presents an impressive loss of performance; the success ratios decrease below 5%. This results answers *[Q2]* showing the limits of current AMP communication technologies, which can not be used without regulation in safety-critical real-time applications, since critical messages, sent by critical VMs, could not be delivered on time due to the requests of other concurrent VMs. However, with the RPUguard mechanism, despite the fact that the performance decrease starts at a transfer rate of 24.3 MB/s due to the overhead introduced by the guarding mechanism, the success ratio decreases as expected considering the maximum bandwidth admitted by RPUGuard imposed at $25,6$ MB/s. Therefore, to assure 100% of success rate in the following experiments, RPUGuard is set with a $BW_{tot} = 23.28$ MB/s, and this bandwidth is then divided into several isolated communication channels with different requirements. Let us note that a different virtqueue size would impact the bandwidth measurement, so the desired bandwidth (up to reasonable memory limits) can be tuned by configuring the virtqueue size.

Isolation Tests Without Regulation

In order to measure the data transfer latency caused by parallel communications, we deploy the ITER VS algorithm (critical task) together with a disturbance algorithm on our platform (control algorithm details can be found in [159]). The VS algorithm is implemented in the RPU and each millisecond ($1khz$ frequency) receives from the APU the value of the plasma's current and velocity; it executes the control algorithm in a predictable way and sends back the voltage value to the APU as output. At the same time, a disturbance task deployed on the APU sends purpose-

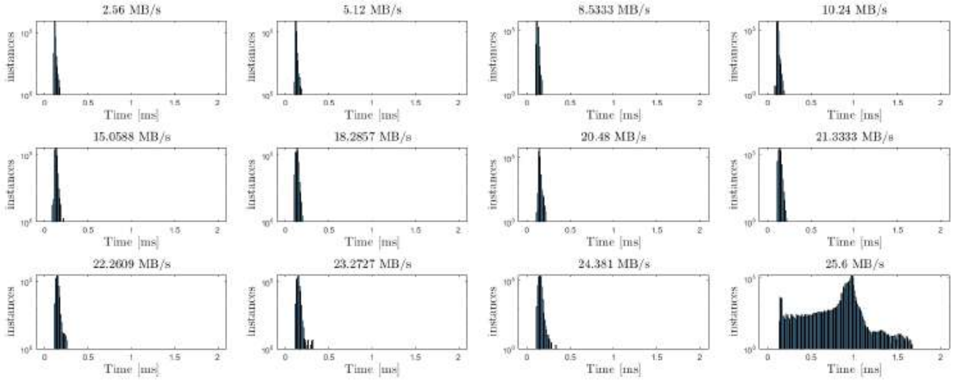


Figure 5.20: Histograms of APU-RPU round-trip latency without regulation, as a function of the disturbance bandwidth

less messages to a different service running on the RPU. So, in principle, the VS algorithm and its APU counterpart are completely separated from the disturbance tasks and related RPU service, so they should not interfere with each other.

The frequency of the disturbance task is incremented step by step, so we can verify how much this factor influences the task that carries out the VS algorithm. Therefore, we calculate the time between the transmission of the current and velocity values from theAPU to the RPU and the moment when the voltage is returned.Considering that the execution time of the algorithm deployed on the RPU has a negligible variance, the differences that we can observe in the round-trip time are only due to the communication channel.

As expected, looking at the histograms of the latencies in Figure 5.20, we can notice that, above a certain transmission bandwidth used by the disturbance task, the occurrences with high latency increase, and some of them exceed the VS deadline (one millisecond).This happens when the disturbance algorithm uses a communication bandwidth of around 25.6 MB/s. This behavior was expected since the communication bandwidth used by the VS algorithm is around 1.02 MB/s.The total bandwidth for our platform without any regulation is at $BW_{tot} = 25.6$ MB/s as previously stated, so, since the disturbance task uses all the available transfer bandwidth, the

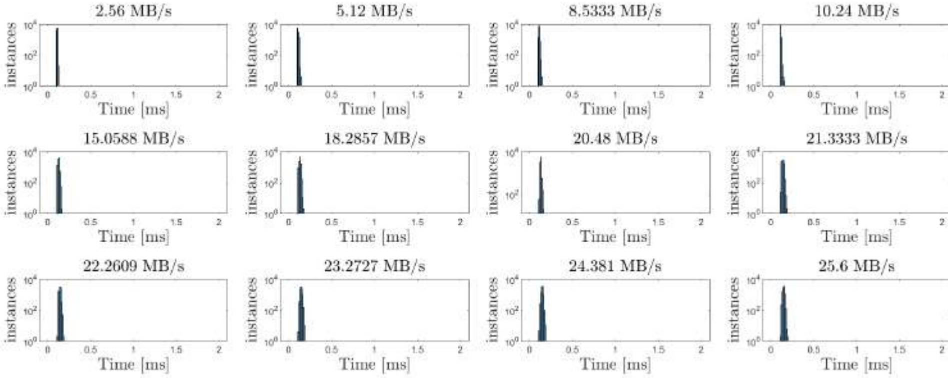


Figure 5.21: Histograms of APU-RPU round-trip latency with RPUGuard, as a function of the disturbance bandwidth

critical task sees its response time increase dramatically. This behavior is not acceptable in mixed-criticality systems, as the correct functioning of the VS critical task is not guaranteed.

Isolation Tests With Regulation

The same experiment seen in Section 5.2.4 is now carried out using the RPUGuard mechanism. Looking at Figure 5.21, it is clear that the interference caused by the disturbance task is now perfectly mitigated. To ensure the independence of the communication channel at the software layer, we have assigned a maximum bandwidth of $23,28 - 1.02 = 22.26$ MB/s to the disturbance task, which corresponds to at most one request per $23\,us$, and we have assigned a bandwidth of 1.02 MB/s to the VS task. We have decided to set the maximum bandwidth at $BW_{tot} = 23.28$ MB/s since, as shown in Figure 5.19, the experiments have a success rate of 100%. Each time the number of requests for each channel is higher than the mentioned limits, RPUGuard delays the requests. Hence, when the frequency of the disturbance task is higher than the limit, the transmission latency perceived by the task is very high, but this has no impact on the transmission latency perceived by the VS task that remains completely isolated. This result answers the *[Q3]* and demonstrates that RPUGuard isolates the communication channels in software using existing AMP mechanisms.

5.3 runPHI Preliminary Orchestration Experiments

As a last evaluation, we show the feasibility and potential of orchestrating containers and partitioning VMs. This scenario reflects the broader context of [**UC3**], where distributed safety-critical systems must be orchestrated across heterogeneous environments, ensuring system robustness. The runPHI implementation (see Figure 4.12) compiles the partition configuration file, loads the binary into the partition, and finally starts it. We measured two metrics commonly of interest for containers: *boot times* and *isolation from interference*. We run the experiments on two ARM-based boards commonly used in embedded edge computing scenarios [186]: a Raspberry Pi 4B and a ZCU (see Section 2.2).

The Raspberry Pi software setup included Docker v24.0.5 using runc v1.1.11, a Linux kernel v5.15 patched with *PREEMPT_RT*, and configured for real-time (debug options disabled, no frequency scaling), *CONFIG_RT_GROUP_SCHED* enabled, and cgroups v1 configured. The Docker daemon was configured to have all the cpu-rt-runtime available. Inside the container, tasks run at 95 FIFO priority. The ZCU software setup included the Jailhouse hypervisor v0.12 patched with the Omnivisor extension [172] to run VMs on heterogeneous cores. The privileged VM in Jailhouse (i.e., root-cell) runs Linux kernel v5.15 patched with *PREEMPT_RT*. In the RPUs, Zephyr (a popular lib-OS RTOS) v3.22.1 runs as a VM.

To compare partitioned containers against Linux containers, we utilized POSIX-compliant applications. We compiled the same application source code twice: once for a Linux process running within an Ubuntu Linux container, and once against Zephyr to run bare-metal as a partitioned VM. Note that the code of the partitioned VM is moved to the Tightly Coupled Memory (TCM) (i.e., programmable on-chip memories) of the RPU before running.

Boot Times

In each experiment, we measured boot times by sampling the container creation timing. The start time was recorded right before the execution

of the low-level container runtime (i.e., runc for Linux-based containers, runPHI for partitioned VM), while the end time was sampled as the first instruction executed within the container. At the end of each experiment, we cleaned the caches in both testbeds to avoid dependencies. The image used in the experiment for the Linux container is an Ubuntu image (76MB) containing our executable file. The image used for the partitioned VM is a micro-ROS (16.8MB) compiled to run over an RPU of the ZCU104.

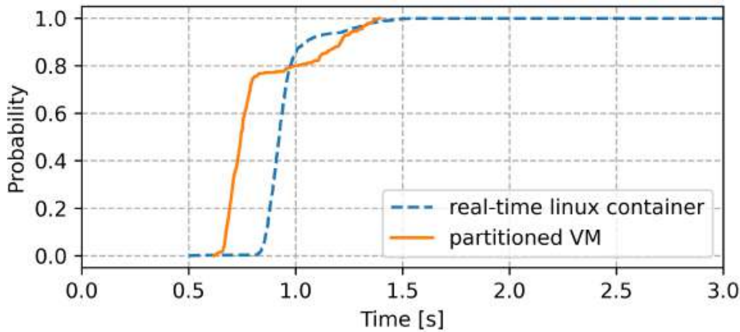


Figure 5.22: Cumulative distribution function of the boot times of the solutions compared. The solutions show comparable boot times.

Figure 5.22 shows the results. The time to boot a partitioned container is comparable to a standard Linux container. Breaking down its boot times, loading and starting the application on the RPU takes $0.142 \pm 0.001s$. The rest of the time, which contributes to almost the totality of the variability, is spent compiling the Jailhouse partition configuration file.

As shown previously in Section 5.2, the application loading times with the Omnivisor depend linearly on the image size. On the other hand, Linux container boot times are comparable with the ones measured in [187], where authors showed that multiple factors can impact Linux containers' boot times.

Isolation from Interference

We compared the isolation from interference between a partitioned container and a Linux real-time container. The aim is to show the potential of integrating the Omnivisor into container orchestration, rather than eval-

uating the isolation of partitioning hypervisor, already explored in-depth in [55]. This allows leveraging the flexibility provided by container orchestration tools while keeping the isolation.

We compiled a periodic POSIX task performing matrix calculations for both Linux and Zephyr OS (targeting the ZCU104 RPU). We used the same compilation flags in both building processes. In each experiment, we run the task for 1000 iterations and measure the execution time of each iteration. We run the experiments in varied stress conditions, including an experiment with no co-located stress on the node and experiments with co-located stress generated through *stress-ng*. In particular, for each experiment, we apply one of two stress types between *memcpy* and *UDP*, and we repeat the experiment with increasing stress intensities, i.e., 1, 2, 4, and 8 threads. These tests were chosen because they stress memory and interrupt handling subsystems, known to be the two major sources of interference, even in partitioning systems [55].

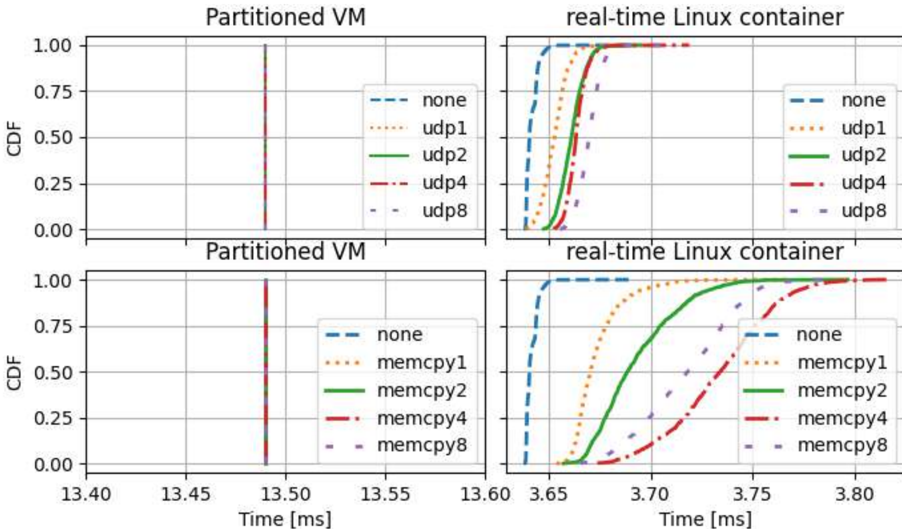


Figure 5.23: Cumulative distribution function of the execution times in varied stress conditions. Partitioned containers show constant times.

Figure 5.23 shows the results. The partitioned container is about 3 times slower than the real-time Linux container: this is due to the RPU

clock frequency of the ZCU104, which is one-third of the clock frequency of the Raspberry Pi. However, the partitioned container outperforms the real-time Linux container in terms of predictability. Despite the co-located stress, the partitioned container exhibits the same execution time (with a μs resolution) across all test repetitions. Indeed, the code loaded in the TCM allows the application in the partitioned container to avoid resource contention regarding cache and RAM.

Furthermore, by utilizing the runPHI framework, we can orchestrate both partitioned VMs and traditional containers seamlessly. The results demonstrate that the isolation achieved with partitioned VMs is notably effective, enabling us to move applications within a safety-critical distributed environment such as nuclear fusion, as described in [**UC3**]. This capability enhances the flexibility and scalability of deploying control systems in complex settings, ensuring that high levels of performance and predictability can be maintained even in dynamic operational contexts.

6

Conclusions and Future Activities

THIS dissertation addressed the critical challenge of optimizing hardware resource utilization in the complex and safety-critical environment of nuclear fusion reactors. These systems currently face a significant issue of *overprovisioning*: although the hardware platforms used are extremely powerful, only a fraction of the available computational resources are fully utilized. This inefficiency stems from the need to maintain strict real-time guarantees, which limits the degree to which the system can allocate resources to non-critical applications. As a result, large portions of processing units remain underutilized to prevent interference with critical real-time tasks.

To tackle this problem and improve both utilization and performance in hardware that must reliably operate for decades in fusion reactors, this dissertation presents an in-depth analysis of real-time virtualization techniques. The work explores how virtualization can improve the resource efficiency of heterogeneous platforms, especially those used in critical environments like nuclear fusion.

The study begins with a comprehensive modeling and Design of Exper-

iments (DoE) analysis, aimed at improving temporal isolation in mixed-criticality systems (MCSs). This ensures that real-time performance is maintained, even when applications of varying criticality are co-located on the same hardware platform. To address the challenges of resource management in MPSoC platforms, we present the *Omnivisor* model, which extends partitioning virtualization frameworks to effectively manage asymmetric cores (e.g., ARM64, ARM32, and RISC-V) and accelerators. The Omnivisor introduces mechanisms that guarantee memory bandwidth reservation for asymmetric cores and accelerators, ensuring robust isolation for critical applications while optimizing system resource utilization. As part of the *Omnivisor*, the *RPUGuard* mechanism enables fine-grained control over communication between virtualized processors, enhancing real-time performance. Additionally, we developed the *RunPHI* framework, which integrates the Omnivisor into an orchestration system, enabling efficient task management and deployment across heterogeneous cores and accelerators.

6.1 Summary of Contributions

Through this work, we have demonstrated how innovative virtualization models can facilitate the deployment of complex safety-critical applications on modern MPSoCs, ensuring that each application operates within a well-isolated, well-controlled environment. By doing so, we provide a significant step forward in maximizing the use of available hardware resources while maintaining the strict real-time requirements of critical systems.

The contributions of this dissertation can be summarized as follows:

Comprehensive Model for Mixed-Criticality Systems (MCS) Deployment: We proposed a standardized model for deploying MCSs across virtualized MPSoCs and distributed systems within a cloud-to-thing continuum. This model improves scalability and simplifies integration, specifically applied to complex systems like ITER's magnetic control, ensuring reliable operation across distributed environments.

- **Temporal Isolation Assessment and Hypervisor Optimization**: We developed a comprehensive methodology to assess temporal isolation in virtualized systems. This assessment highlights the

importance of optimizing hypervisor configurations for real-time systems, especially when managing high-criticality applications that require strict timing guarantees. Partitioning-based hypervisors were identified as optimal for these environments.

- **Comprehensive Model for MCS Deployment**: We proposed a standardized model for deploying MCSs across virtualized MPSoCs and distributed systems within a cloud-to-thing continuum. This model improves scalability and simplifies integration, specifically applied to complex systems like ITER's magnetic control, ensuring reliable operation across distributed environments.

- **The Omnivisor Framework**: Extending traditional static partitioning hypervisors, Omnivisor manages asymmetric cores (ARM64, ARM32, RISC-V) and accelerators (such as FPGAs and RPUs). This model enhances system dependability and facilitates the isolation and resource allocation in mixed-criticality applications.

- **RPUGuard Communication Framework**: Introduced as part of the Omnivisor model, RPUGuard provides precise control over inter-processor communication in virtualized asymmetric cores. By minimizing interference between cores, it ensures reliable real-time performance in mixed-criticality environments.

- **Lightweight Virtualization of Microcontroller-Level Cores**: A novel design for virtualizing microcontroller-level cores (e.g., RPUs) that enhances resource utilization. This approach ensures that these cores can be effectively managed through virtualization without compromising their real-time performance, paving the way for future architectures with increased system efficiency.

- **RunPHI Integration for Orchestration**: We introduced the RunPHI framework, which integrates Omnivisor into an orchestration system, allowing seamless management of virtualized applications across distributed nodes. This facilitates flexible deployment while ensuring strong isolation between co-located applications.

- **Empirical Validation**: Through evaluation in three nuclear fusion use cases, particularly the ITER vertical stabilization control sys-

tem, we demonstrated the effectiveness of the Omnivisor in ensuring real-time performance and resource isolation across heterogeneous processing units.

6.2 Implications for Nuclear Fusion

The contributions of this dissertation have significant implications for the development of nuclear fusion control systems, especially for large-scale projects like ITER. Our experiments demonstrate how the innovations introduced can enhance the design, deployment, and performance of these systems.

For instance, we tackled the challenge of co-locating real-time safety controllers and monitoring applications on the same MPSoC platform. By utilizing the Omnivisor framework, we successfully deployed real-time controllers alongside non-critical applications without compromising performance or predictability. This approach not only mitigates hardware costs and complexity in the control room but also boosts overall system efficiency, as evidenced in [**UC1**].

In addition, the flexibility of the Omnivisor allowed for the deployment of multiple versions of control algorithms on the same platform. This enabled parallel testing and validation without the need for additional hardware for each iteration. In the high-stakes environment of nuclear fusion, where safety and precision are paramount, this innovation accelerates the testing and certification process. For example, in our experiments, we deployed memory-intensive workloads that behave as a complex controller alongside the main one, ensuring their isolation from each other. This method not only enhanced testing efficiency but also reinforced system redundancy, as highlighted in [**UC2**].

Furthermore, our research explored the implementation of distributed control in fog and edge computing scenarios. Our findings demonstrate that integrating the Omnivisor in an orchestration system can ease the deployment of critical applications while maintaining the needed real-time performance, as described in [**UC3**].

These findings collectively underscore the potential of virtualization and the Omnivisor framework to streamline and future-proof the development of control systems in nuclear fusion environments.

6.3 Future Research Directions

Looking forward, several areas present exciting opportunities for further research and development. A key direction involves extending the Omnivisor framework to enable dynamic FPGA reconfiguration at runtime. This would allow critical systems to adapt their hardware resources on the fly, deploying only the necessary hardware accelerators when needed. This dynamic capability could be vital for responding to changing workloads or fault conditions in nuclear fusion environments, providing additional flexibility and enhancing system efficiency.

Further research can also explore:

- **Run-Time Hardware Reconfiguration**: Integrating reconfigurable FPGAs into the Omnivisor framework by leveraging existing technologies such as the Xilinx Dynamic Partial Reconfiguration (DPR) opens the possibility of altering hardware architecture dynamically, allowing systems to load different hardware configurations as needed without disturbing critical real-time applications. By leveraging FPGA partial reconfiguration, it would be possible to swap in hardware modules that can enhance performance, such as AI accelerators or measurement tools while using the Omnivisor to regulate the bandwidth allocation and assure the spatial isolation of multiple accelerators and processors.

- **Expanding Memory and Resource Management**: Enhancing the Omnivisor's control over shared resources, such as memory and interconnects, will be essential for further improving isolation and system predictability. Techniques such as dynamic memory partitioning and resource throttling could ensure that critical applications have guaranteed access to the resources they need when needed.

- **Deeper Integration of Machine Learning for Predictive Control**: With the increasing interest in integrating AI and machine learning into real-time control systems, future versions of Omnivisor could support AI-driven decision-making by dynamically optimizing resource allocation based on workload predictions while maintaining one back-up simple controller which has its resource guarantee.

By continuing to advance these areas of research, we can further improve the reliability, performance, and safety of real-time systems used in nuclear fusion and other critical fields, moving closer to the realization of a fully scalable, integrated, and efficient control infrastructure.

Publications

Heterogeneous Virtualization

- 1) M. Cinque, G. De Tommasi, S. Dubbioso and **D. Ottaviano**, *"Virtualizing Real-Time Processing Units in Multi-Processor Systems-on-Chip"* in Proceeding of **6th IEEE International Forum on Research and Technology for Society and Industry (RTSI)**, Naples, Italy, 2021, pp. 329-333,

  DOI: 10.1109/RTSI50628.2021.9597281

- 2) M. Cinque, G. De Tommasi, S. Dubbioso and **D. Ottaviano**, *"RPUGuard: Real-Time Processing Unit Virtualization for Mixed-Criticality Applications"* in Proceeding of **18th IEEE European Dependable Computing Conference (EDCC)**, Zagaroza, Spain, 2022, pp. 97-104,

  DOI: 10.1109/EDCC57035.2022.00025

- 3) M. Cinque, L. De Simone, N. Mazzocca, **D. Ottaviano** and F. Vitale, *"Evaluating virtualization for fog monitoring of real-time*

applications in mixed-criticality systems" in **Real-Time Systems**, 59(4), 534–567,

DOI: 10.1007/s11241-023-09410-4

- 4) **D. Ottaviano**, Ciraolo F., R. Mancuso and M. Cinque, *"The Omnivisor: A Real-Time Static Partitioning Hypervisor Extension for Heterogeneous Core Virtualization over MPSoCs"* In Proceeding of **36th Euromicro Conference on Real-Time Systems (ECRTS)**, Leibniz International Proceedings in Informatics (LIPIcs), Volume 298, pp. 7:1-7:27, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2024),

  DOI: 10.4230/LIPIcs.ECRTS.2024.7

- 5) M. Cinque, L. De Simone and **D. Ottaviano**, *"Temporal isolation assessment in virtualized safety-critical mixed-criticality systems: A case study on Xen hypervisor"* in **Journal of Systems and Software (JSS)**, 216, 112147,

  DOI: 10.1016/j.jss.2024.112147

Fusion Engineering

- 6) G. De Tommasi, M. Cinque, M. Mattei, **D. Ottaviano**, A. Pironti, S. Rosiello, F. Villone, P. de Vries, T. Ravensbergen and L. Zabeo, *"System-Engineering approach for the ITER PCS design: The correction coils current controller case study"* in **Fusion Engineering and Design**, 185, 113317,

  DOI: 10.1016/j.fusengdes.2022.113317

- 7) **D. Ottaviano**, M. Cinque, Manduchi G. and S. Dubbioso, *"Virtualization of accelerators in embedded systems for mixed-criticality: RPU exploitation for fusion diagnostics and control"*, in **Fusion Engineering and Design**, Vol. 190, pp. 113518, 2023,

  DOI: 10.1016/j.fusengdes.2023.113518

- 8) S. Dubbioso, **D. Ottaviano**, F. Fiorenza, N. Ferron, G. Manduchi, R. Ambrosino, G. De Tommasi, *"Rapid prototyping of control*

modules for the DTT Plasma Control System" **33rd Symposium on Fusion Technology (SOFT'24)**, Dublin, Ireland, September 2024.

Microcontroller Virtualization

- 9) S. Mercogliano, **D. Ottaviano**, A. Cilardo and M. Cinque, *"Lightweight and Predictable Memory Virtualization on Medium-Size Microcontroller"*, 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), Valencia, Spain, 2024, pp. 1-2,

  DOI: 10.23919/DATE58400.2024.10546637

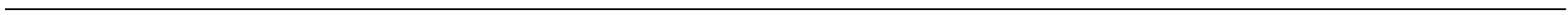
Mixed-Criticality Containers

- 10) M. Barletta, M. Cinque, L. D. Simone, R. D. Corte, G. Farina and **D. Ottaviano**, *"RunPHI: Enabling Mixed-criticality Containers via Partitioning Hypervisors in Industry 4.0"*, 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Charlotte, NC, USA, 2022, pp. 134-135,

  DOI: 10.1109/ISSREW55968.2022.00058

- 11) M. Barletta, M. Cinque, L. De Simone, R. D. Corte, G. Farina and **D. Ottaviano**, *"Partitioned Containers: Towards Safe Clouds for Industrial Applications"*, 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), Porto, Portugal, 2023, pp. 84-88,

  DOI: 10.1109/DSN-S58398.2023.00029



Bibliography

[1] J. Shalf, “The future of computing beyond moore’s law,” *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190061, 2020.

[2] R. R. Schaller, “Moore’s law: past, present and future,” *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[3] T. N. Theis and H.-S. P. Wong, “The end of moore’s law: A new beginning for information technology,” *Computing in science & engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[4] A. Jerraya and W. Wolf, *Multiprocessor systems-on-chips.* Elsevier, 2004.

[5] A. Pironti and M. Walker, “Fusion, tokamaks, and plasma control: an introduction and tutorial,” *IEEE Control Systems Magazine*, vol. 25, no. 5, pp. 30–43, 2005.

[6] G. De Tommasi, D. Alves, T. Bellizio, R. Felton, A. Neto, F. Sartori, R. Vitelli, L. Zabeo, R. Albanese, G. Ambrosino *et al.*, “Real-time systems in tokamak devices. a case study: The jet tokamak,” *IEEE Transactions on Nuclear Science*, vol. 58, no. 4, pp. 1420–1426, 2011.

[7] A. C. Neto, F. Sartori, F. Piccolo, R. Vitelli, G. De Tommasi, L. Zabeo, A. Barbalace, H. Fernandes, D. F. Valcarcel, and A. J. Batista, “Marte: A multiplatform real-time framework,” *IEEE Transactions on Nuclear Science*, vol. 57, no. 2, pp. 479–486, 2010.

[8] M. Cinque, G. De Tommasi, S. Dubbioso, and D. Ottaviano, "RPU-Guard: Real-time processing unit virtualization for mixed-criticality applications," in *2022 18th European Dependable Computing Conference (EDCC)*. IEEE, 2022, pp. 97–104.

[9] G. Avon, A. Buscarino, E. De Marchi, L. Fortuna, A. C. Neto, F. Sartori, and F. Zanon, "Marte2 on arm platforms integration challenges: An asymmetric multiprocessing approach for the iter magnetics diagnostics," *Fusion Engineering and Design*, vol. 202, p. 114370, 2024.

[10] S. Dubbioso *et al.*, "Vertical stabilization of tokamak plasmas via extremum seeking," *IFAC Journal of Systems and Control*, vol. 21, p. 100203, 2022.

[11] S. Dubbioso, G. De Tommasi, A. Mele, G. Tartaglione, M. Ariola, and A. Pironti, "A deep reinforcement learning approach for vertical stabilization of tokamak plasmas," *Fusion Engineering and Design*, vol. 194, p. 113725, 2023.

[12] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, "An industrial view on the common academic understanding of mixed-criticality systems," *Real-Time Systems*, vol. 54, pp. 745–795, 2018.

[13] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–37, 2017.

[14] CENELEC, "EN 50128," *Railway applications-Communication, Signaling and Processing Systems-Software for Railway Control and Protection Systems*, 2011.

[15] RTCA, "DO-178B Software Considerations in Airborne Systems and Equipment Certification," *Requirements and Technical Concepts for Aviation*, 1992.

[16] ISO, "Product Development: Software Level," *ISO 26262: Road vehicles – Functional safety*, vol. 6, 2011.

[17] S. Vestal, “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance,” in *28th IEEE international real-time systems symposium (RTSS 2007)*. IEEE, 2007, pp. 239–243.

[18] A. Burns and R. I. Davis, “Mixed criticality systems-a review:(february 2022),” York, 2022.

[19] IEEE, “IEEE 1934-2018 - IEEE standard for adoption of OpenFog reference architecture for Fog Computing,” 2018.

[20] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE IoT-J*, vol. 3, no. 5, pp. 637–646, 2016.

[21] G. Avon, A. Buscarino, A. C. Neto, and F. Sartori, “MARTe2 embedded signal processing unit for the ITER magnetics diagnostics,” in *IECON 2021–47th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2021, pp. 1–6.

[22] P. Bellavista, J. Berrocal, A. Corradi, S. K. Das, L. Foschini, and A. Zanni, “A survey on fog computing for the internet of things,” *Pervasive and mobile computing*, vol. 52, pp. 71–99, 2019.

[23] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, “A comprehensive survey of industry practice in real-time systems,” *Real-Time Systems*, vol. 58, no. 3, pp. 358–398, 2022.

[24] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, “A survey of techniques for reducing interference in real-time applications on multicore platforms,” *IEEE Access*, vol. 10, pp. 21 853–21 882, 2022.

[25] J. P. Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende, “Multi-core devices for safety-critical systems: A survey,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–38, 2020.

[26] A. Cilardo, M. Cinque, L. De Simone, and N. Mazzocca, “Virtualization over multiprocessor system-on-chip: an enabling paradigm for industrial iot,” *arXiv preprint arXiv:2112.15404*, 2021.

[27] M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello, “Virtualizing mixed-criticality systems: A survey on industrial trends and issues,” *Future Generation Computer Systems*, 2021.

[28] L. Abeni and D. Faggioli, “Using xen and kvm as real-time hypervisors,” *Journal of Systems Architecture*, vol. 106, p. 101709, 2020.

[29] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, “Look mum, no vm exits!(almost),” *arXiv preprint arXiv:1705.06932*, 2017.

[30] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, “Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems,” in *Workshop on next generation real-time embedded systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[31] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, “Worst case delay analysis for memory interference in multicore systems,” in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 741–746.

[32] K. Jeffay, “Scheduling sporadic tasks with shared resources in hard-real-time systems.” in *RTSS*, vol. 92. Citeseer, 1992, pp. 89–99.

[33] M. Cavinato, G. Manduchi, A. Luchetta, and C. Taliercio, “General-purpose framework for real time control in nuclear fusion experiments,” *IEEE Transactions on Nuclear Science*, vol. 53, no. 3, pp. 1002–1008, 2006.

[34] A. C. Neto *et al.*, “A survey of recent MARTe based systems,” *IEEE Transactions on Nuclear Science*, vol. 58, pp. 1482–1489, 2011.

[35] P. Perek, D. Makowski, M. Kadziela, W.-R. Lee, A. Zagar, S. Simrock, L. Abadie, J.-h. Lee, S.-j. Lee, and H.-j. Kim, “Evaluation of iter real-time framework in plasma diagnostics applications,” *Fusion Engineering and Design*, vol. 192, p. 113623, 2023.

[36] Y.-W. Zhang, J.-P. Ma, H. Zheng, and Z. Gu, “Criticality-aware edf scheduling for constrained-deadline imprecise mixed-criticality systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[37] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Workshop on Mixed-Criticality Systems (colocated with RTSS)*, 2013.

[38] Y.-W. Zhang, R.-K. Chen, and Z. Gu, "Energy-aware partitioned scheduling of imprecise mixed-criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 3733–3742, 2023.

[39] B. Ranjbar, T. D. Nguyen, A. Ejlali, and A. Kumar, "Power-aware runtime scheduler for mixed-criticality systems on multicore platform," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 10, pp. 2009–2023, 2020.

[40] S. Altmeyer, L. Cucu-Grosjean, and R. I. Davis, "Static probabilistic timing analysis for real-time systems using random replacement caches," *Real-Time Systems*, vol. 51, pp. 77–123, 2015.

[41] B. Ranjbar, A. Hoseinghorban, S. S. Sahoo, A. Ejlali, and A. Kumar, "Improving the timing behaviour of mixed-criticality systems using chebyshev's theorem," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 264–269.

[42] L. Abeni, A. Biondi, and E. Bini, "Hierarchical scheduling of real-time tasks over linux-based virtual machines," *Journal of Systems and Software*, vol. 149, pp. 234–249, 2019.

[43] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *2008 Euromicro Conference on Real-Time Systems*. IEEE, 2008, pp. 181–190.

[44] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing mixed criticality applications on modern heterogeneous MPSoC platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[45] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded*

Technology and Applications Symposium (RTAS). IEEE, 2013, pp. 45–54.

[46] A. Sarkar, F. Mueller, and H. Ramaprasad, "Predictable task migration for locked caches in multi-core systems," in *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, 2011, pp. 131–140.

[47] A. Asaduzzaman, F. N. Sibai, and M. Rani, "Improving cache locking performance of modern embedded systems via the addition of a miss table at the l2 cache level," *Journal of Systems Architecture*, vol. 56, no. 4-6, pp. 151–162, 2010.

[48] B. Sun, T. Kloda, S. A. Garcia, G. Gracioli, and M. Caccamo, "Minimizing cache usage with fixed-priority and earliest deadline first scheduling," *Real-Time Systems*, pp. 1–40, 2024.

[49] D. Tarapore, S. Roozkhosh, S. Brzozowski, and R. Mancuso, "Observing the invisible: Live cache inspection for high-performance embedded systems," *IEEE Transactions on Computers*, vol. 71, no. 3, pp. 559–572, 2021.

[50] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 55–64.

[51] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, "MemPol: Policing core memory bandwidth from outside of the cores," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 235–248.

[52] S. Garcia-Esteban, A. Serrano-Cases, J. Abella, E. Mezzetti, and F. J. Cazorla, "Quasi isolation QoS setups to control MPSoC contention in integrated software architectures," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[53] M. Zini, D. Casini, and A. Biondi, "Analyzing arm's mpam from the perspective of time predictability," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 168–182, 2022.

[54] D. Costa, L. Cuomo, D. Oliveira, I. M. Savino, B. Morelli, J. Martins, F. Tronci, A. Biasci, and S. Pinto, "IRQ coloring: Mitigating interrupt-generated interference on ARM multicore platforms," in *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[55] J. Martins and S. Pinto, "Shedding light on static partitioning hypervisors for ARM-based mixed-criticality systems," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 2023, pp. 40–53.

[56] AMD, "Microblaze reference guide," https://www.amd.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug984-vivado-microblaze-ref.pdf, [Accessed 21-02-2024].

[57] YosysHQ, "picorv32," https://github.com/YosysHQ/picorv32, [Accessed 27-02-2024].

[58] S. A. Panchamukhi and F. Mueller, "Providing task isolation via tlb coloring," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium.* IEEE, 2015, pp. 3–13.

[59] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, "Leveraging hardware QoS to control contention in the Xilinx Zynq UltraScale+ MPSoC," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[60] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms," *Real-Time Systems*, vol. 58, no. 3, pp. 235–274, 2022.

[61] 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.

[62] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[63] 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," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[64] D. Cotroneo, L. De Simone, and R. Natella, "On temporal isolation assessment in virtualized railway signaling as a service systems," in *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, 2022, pp. 1–5.

[65] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, vol. 9. Citeseer, 2009.

[66] R. West, Y. Li, E. Missimer, and M. Danish, "A virtualized separation kernel for mixed-criticality systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 3, pp. 1–41, 2016.

[67] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 1–14.

[68] S. H. VanderLeest, "Designing a future airborne capability environment (face) hypervisor for safety and security," in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*. IEEE, 2017, pp. 1–9.

[69] G. Bloom and J. Sherrill, "Harmonizing arinc 653 and realtime posix for conformance to the face technical standard," in *2020 IEEE 23rd international symposium on real-time distributed computing (ISORC).* IEEE, 2020, pp. 98–105.

[70] A. Zuepke, M. Bommert, and D. Lohmann, "Autobest: a united autosar-os and arinc 653 kernel," in *21st IEEE real-time and embedded technology and applications symposium.* IEEE, 2015, pp. 133–144.

[71] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran, "Breaking the boundaries in heterogeneous-ISA datacenters," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 645–659, 2017.

[72] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–16.

[73] F. X. Lin, Z. Wang, and L. Zhong, "K2: A mobile operating system for heterogeneous coherence domains," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 285–300, 2014.

[74] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 29–44.

[75] Y.-J. Huang, H.-H. Wu, Y.-C. Chung, and W.-C. Hsu, "Building a kvm-based hypervisor for a heterogeneous system architecture compliant system," in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2016, pp. 3–15.

[76] P. Olivier, B. Ravindran, and A. Barbalace, "The multihype: Virtualizing heterogeneous-ISA architectures," in *9th Workshop on Systems for Multi-core and Heterogeneous Architectures (SFMA)*, 2019.

[77] F. Baum and A. Raghuraman, "Making full use of emerging ARM-based heterogeneous multicore SoCs," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[78] D. Ottaviano, M. Cinque, G. Manduchi, and S. Dubbioso, "Virtualization of accelerators in embedded systems for mixed-criticality: RPU exploitation for fusion diagnostics and control," *Elsevier Fusion Engineering and Design*, 2023.

[79] S. Alonso, J. Lazaro, J. Jimenez, L. Muguira, and U. Bidarte, "Evaluating the OpenAMP framework in real-time embedded SoC platforms," in *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*. IEEE, 2021, pp. 1–6.

[80] C. Moratelli, S. Zampiva, and F. Hessel, "Full-virtualization on mips-based mpsocs embedded platforms with real-time support," in *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design*, 2014, pp. 1–7.

[81] T. Mück, A. A. Fröhlich, G. Gracioli, A. M. Rahmani, J. G. Reis, and N. Dutt, "CHIPS-AHOy: A predictable holistic cyber-physical hypervisor for MPSoCs," in *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2018, pp. 73–80.

[82] A. Biondi, D. Casini, G. Cicero, N. Borgioli, G. Buttazzo, G. Patti, L. Leonardi, L. L. Bello, M. Solieri, P. Burgio *et al.*, "SPHERE: A Multi-SoC architecture for next-generation cyber-physical systems based on heterogeneous platforms," *IEEE Access*, vol. 9, pp. 75 446–75 459, 2021.

[83] T. Xia, Y. Tian, J.-C. Prévotet, and F. Nouvel, "Ker-one: A new hypervisor managing fpga reconfigurable accelerators," *Journal of Systems Architecture*, vol. 98, pp. 453–467, 2019.

[84] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory fpga platforms," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 827–844.

[85] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3, pp. 134–155, 2018.

[86] B. Schulz and B. Annighöfer, "Evaluation of adaptive partitioning and real-time capability for virtualization with xen hypervisor," *Transactions on Aerospace and Electronic Systems*, vol. 58, no. 1, pp. 206–217, 2022.

[87] S. Alonso, J. Lázaro, J. Jiménez, L. Muguira, and A. Largacha, "Analysing the interference of xen hypervisor in the network speed," in *Conference on Design of Circuits and Integrated Systems.* IEEE, 2020, pp. 1–6.

[88] FuSa SIG. FuSa SIG Charted. https://wiki.xen.org/wiki/FuSa_SIG/Charter.

[89] ——. FuSa SIG/Presentations. https://wiki.xenproject.org/wiki/FuSa_SIG/Presentation.

[90] Linux Foundation. Xen Project 4.18 Feature List. [Online]. Available: https://wiki.xenproject.org/wiki/Xen_Project_4.18_Feature_List

[91] International Electrotechnical Commission, "Software Requirements," *IEC 61508-3*, 1998.

[92] RTCA, "DO-178C - Software Considerations in Airborne Systems and Equipment Certification."

[93] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *International Conference on Computational Science and Engineering.* IEEE, 2013, pp. 685–692.

[94] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: a dynamic cache partitioning system using page coloring," in *International Conference on Parallel Architecture and Compilation Techniques.* IEEE, 2014, pp. 381–392.

[95] R. A. Fisher, "Design of experiments," *British Medical Journal*, vol. 1, no. 3923, p. 554, 1936.

[96] L. St, S. Wold *et al.*, "Analysis of variance (anova)," *Chemometrics and intelligent laboratory systems*, vol. 6, no. 4, pp. 259–272, 1989.

[97] C. I. King, "Stress-ng," *URL: http://kernel. ubuntu. com/git/ck-ing/stressng. git/(visited on 28/03/2018)*, 2017.

[98] L. Tran, P. J. Radcliffe, and L. Wang, "Simulation is essential for embedded control systems with task jitter," *Design Automation for Embedded Systems*, vol. 25, pp. 177–191, 2021.

[99] Y. H. Jo and B. W. Choi, "Performance evaluation of real-time linux for an industrial real-time platform," *International journal of advanced smart convergence*, vol. 11, no. 1, pp. 28–35, 2022.

[100] A. Hughes and A. Awad, "Quantifying performance determinism in virtualized mixed-criticality systems," in *international symposium on real-time distributed computing*. IEEE, 2019, pp. 181–184.

[101] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *Computing Surveys*, vol. 52, no. 1, pp. 1–36, 2019.

[102] Z. Jiang, K. Yang, Y. Ma, N. Fisher, N. Audsley, and Z. Dong, "I/o-guard: Hardware/software co-design for i/o virtualization with guaranteed real-time performance," in *Design Automation Conference*. IEEE, 2021, pp. 1159–1164.

[103] S. Jan and G. Shieh, "Sample size determinations for welch's test in one-way heteroscedastic anova," *British Journal of Mathematical and Statistical Psychology*, vol. 67, no. 1, pp. 72–93, 2014.

[104] The Linux Foundation, "XenBus." [Online]. Available: https://wiki.xenproject.org/wiki/XenBus

[105] A. Biondi, G. C. Buttazzo, and M. Bertogna, "Schedulability analysis of hierarchical real-time systems under shared resources," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1593–1605, 2015.

[106] J. Lee, S. Xi, S. Chen, L. T. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky, "Realizing compositional scheduling through virtualization," in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. IEEE, 2012, pp. 13–22.

[107] Xilinx, "Zynq Ultrascale+ Device Technical Reference Manual," https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm, [Accessed 07-02-2024].

[108] ——, "Versal Device Technical Reference Manual," https://docs.xilinx.com/r/en-US/am011-versal-acap-trm, [Accessed 07-02-2024].

[109] NXP, "i.MX8-series processors," https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-applications-processors:IMX8-SERIES, [Accessed 07-02-2024].

[110] D. Mvondo, B. Teabe, A. Tchana, D. Hagimont, and N. De Palma, "Closer: A new design principle for the privileged virtual machine os," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019, pp. 49–60.

[111] S. Stabellini, "Xen project blog," https://xenproject.org/2019/12/16/true-static-partitioning-with-xen-dom0-less/, 2019, [Accessed 27-02-2024].

[112] G. Schwäricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A real-time VirtIO-based framework for predictable inter-VM communication," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 27–40.

[113] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable virtualization on memory protection unit-based microcontrollers," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 62–74.

[114] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on trustzone-enabled microcontrollers? voilà!" in *2019*

IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2019, pp. 293–304.

[115] D. Oliveira, T. Gomes, and S. Pinto, "utango: an open-source tee for iot devices," *IEEE Access*, vol. 10, pp. 23 913–23 930, 2022.

[116] Minerva, "Jailhouse-RT GitLab repository," https://gitlab.com/minervasys/public/jailhouse/-/tree/minerva/public?ref_type=heads, [Accessed 08-02-2024].

[117] ARM, "PSCI specification — developer.arm.com," https://developer.arm.com/Architectures/Power%20State%20Coordination%20Interface, [Accessed 20-02-2024].

[118] Intel, "ACPI specification — intel.com," https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/processor-vendor-specific-acpi-specification.pdf, [Accessed 20-02-2024].

[119] Y. Gribov, A. Kavin, V. Lukash, R. Khayrutdinov, G. Huijsmans, A. Loarte, J. A. Snipes, and L. Zabeo, "Plasma vertical stabilisation in ITER," *Nuclear Fusion*, vol. 55, no. 7, p. 073021, 2015.

[120] G. Ambrosino, M. Ariola, G. De Tommasi, and A. Pironti, "Plasma vertical stabilization in the ITER tokamak via constrained static output feedback," *IEEE Transactions on Control Systems Technology*, vol. 19, no. 2, pp. 376–381, 2010.

[121] "ITER website," https://www.iter.org/.

[122] OpenAMP. Openamp-rpmsg-virtio-implementation. [Online]. Available: https://github.com/OpenAMP/open-amp/wiki

[123] R. Russell, "Virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Sys. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.

[124] D. Oliveira, M. Costa, S. Pinto, and T. Gomes, "The future of low-end motes in the Internet of Things: A prospective paper," *Electronics*, vol. 9, no. 1, p. 111, 2020.

[125] M. Ashwathnarayan, J. Vaishnavi, A. Kamath, J. Guddeti *et al.*, "Virtualised Ecosystem to Envisage Numerous Applications on an Automotive Microcontroller," in *CS & IT Conference Proceedings*, vol. 12, no. 6. CS & IT Conference Proceedings, 2022.

[126] D. Buttle and S. Gold, "MCUs and Virtualization in Zone E/E Architectures," *ATZelectronics worldwide*, vol. 17, no. 10, pp. 18–24, 2022.

[127] C. Meenderinck, A. Molnos, and K. Goossens, "Composable virtual memory for an embedded SoC," in *2012 15th Euromicro Conference on Digital System Design*. IEEE, 2012, pp. 766–773.

[128] I. Puaut and D. Hardy, "Predictable paging in real-time systems: A compiler approach," in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*. IEEE, 2007, pp. 169–178.

[129] M. Bennett and N. C. Audsley, "Predictable and efficient virtual addressing for safety-critical real-time systems," in *Proceedings 13th Euromicro Conference on Real-Time Systems*. IEEE, 2001, pp. 183–190.

[130] W. Puffitsch and M. Schoeberl, "Time-predictable virtual memory," in *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2016, pp. 158–165.

[131] Texas Instruments. Hercules RM57Lx LaunchPad Development Kit. Accessed: November 28, 2024. [Online]. Available: https://www.ti.com/tool/LAUNCHXL2-RM57L#description

[132] R. E. Corporation. Microprocessors for Real-Time Control and Networking of Industrial Equipment by only one chip. Accessed: November 28, 2024. [Online]. Available: https://www.renesas.com/en/products/microcontrollers-microprocessors/rz-mpus/rzt1-microprocessors-real-time-control-and-networking-industrial-equipment-only-one-chip

[133] O. Burkacky, J. Deichmann, G. Doll, and C. Knochenhauer, "Rethinking car software and electronics architecture," McKinsey & Company - Center for future mobility, Tech. Rep., 2018.

[134] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, S. Leonard, P. Pannuto, P. Dutta, and P. Levis, “The tock embedded operating system,” in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, pp. 1–2.

[135] W. Zhou, L. Guan, P. Liu, and Y. Zhang, “Good motive but bad design: Why ARM MPU has become an outcast in embedded systems,” *arXiv preprint arXiv:1908.03638*, 2019.

[136] J. Liedtke, “Address space sparsity and fine granularity,” *ACM SIGOPS Operating Systems Review*, vol. 29, no. 1, pp. 87–90, 1995.

[137] S. Mercogliano, “cv32e41s, omitted due to double-blind review,” https://github.com/Granp4sso/cv32e41s, 2023.

[138] S. Mercogliano and D. Ottaviano, “cv32e41s SoC Environment, omitted due to double-blind review,” https://github.com/Granp4sso/cv32e41s_SoC_env, 2023.

[139] D. Ottaviano and S. Mercogliano, “Xpmp experiments repository, omitted due to double-blind review,” https://github.com/DanieleOttaviano/XPMP_experiments, 2023.

[140] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, “Taclebench: A benchmark collection to support worst-case execution time research,” in *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[141] D. C. Van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, “Towards a flexible, lightweight virtualization alternative,” in *Proc. SYSTOR*, 2014, pp. 1–7.

[142] Bloomberg. (2022) Automotive Chip-Shortage Cost Estimate Surges to $110 Billion. [Online]. Available: https://tinyurl.com/2p9akbhu

[143] BlackBerry Limited. (2021) Are Hypervisors the Answer to the Coming Silicon Shortages? (White Paper). [Online]. Available: https://blackberry.qnx.com/content/dam/blackberry-com/Docum

ents/pdf/BlackBerry_QNX_Hypervisor_WhitePaper_22April2021_FINAL.pdf

[144] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, "Criticality-aware monitoring and orchestration for containerized industry 4.0 environments," *ACM TECS*, vol. 23, no. 1, pp. 1–28, 2024.

[145] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," *ACM CSUR*, 2020.

[146] OpenContainers. OCI Image Format Specification. https://github.com/opencontainers/image-spec.

[147] M. Barletta, M. Cinque, L. De Simone, R. Della Corte, G. Farina, and D. Ottaviano, "Runphi: Enabling mixed-criticality containers via partitioning hypervisors in industry 4.0," in *Proc. ISSREW*. IEEE, 2022, pp. 134–135.

[148] Z. Jiang, N. Audsley, P. Dong, N. Guan, X. Dai, and L. Wei, "Mcs-iov: Real-time i/o virtualization for mixed-criticality systems," in *Proc. RTSS*. IEEE, 2019, pp. 326–338.

[149] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. ISORC*. IEEE, 2008, pp. 363–369.

[150] F. Reghenzani, Z. Guo, and W. Fornaciari, "Software fault tolerance in real-time systems: Identifying the future research questions," *ACM Computing Surveys*, vol. 55, no. 14s, pp. 1–30, 2023.

[151] G. Bernat, A. Colin, and S. M. Petters, "Wcet analysis of probabilistic hard real-time systems," in *Proc. RTSS*. IEEE, 2002, pp. 279–288.

[152] S. Bozhko, F. Marković, G. von der Brüggen, and B. B. Brandenburg, "What really is pwcet? a rigorous axiomatic proposal," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 13–26.

[153] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, "Introducing k4. 0s: a model for mixed-criticality container orchestration

in industry 4.0," in *Proc. DASC/PiCom/CBDCom/CyberSciTech*, 2022.

[154] R. Davis and L. Cucu-Grosjean, "A survey of probabilistic schedulability analysis techniques for hard real-time systems." *Leibniz Transactions on Embedded Systems (LITES)*, 2019.

[155] L. Abeni, R. Andreoli, H. Gustafsson, R. Mini, and T. Cucinotta, "Fault tolerance in real-time cloud computing," in *Proc. ISORC*. IEEE, 2023, pp. 170–175.

[156] EUROfusion, "European Research Roadmap to the Realisation of Fusion Energy," 2018, https://www.euro-fusion.org/fileadmin/user_upload/EUROfusion/Documents/2018_Research_roadmap_long_version_01.pdf, Accessed November 2022.

[157] J. Snipes *et al.*, "ITER plasma control system final design and preparation for first plasma," *Nuclear Fusion*, 2021.

[158] G. De Tommasi, "Plasma magnetic control in tokamak devices," *J. Fus. Energy*, vol. 38, no. 3-4, pp. 406–436, 2019.

[159] R. Albanese, R. Ambrosino, A. Castaldo, G. De Tommasi, Z. Luo, A. Mele, A. Pironti, B. Xiao, and Q. Yuan, "Iter-like vertical stabilization system for the east tokamak," *Nuclear Fusion*, vol. 57, no. 8, p. 086039, 2017.

[160] G. De Tommasi, S. Dubbioso, A. Mele, and A. Pironti, "Stabilizing elongated plasmas using extremum seeking: the iter tokamak case study," in *2021 29th Mediterranean Conference on Control and Automation (MED)*. IEEE, 2021, pp. 472–478.

[161] J. Degrave *et al.*, "Magnetic control of tokamak plasmas through deep reinforcement learning," *Nature*, vol. 602, pp. 414–419, 2022.

[162] G. De Tommasi *et al.*, "A RL-based Vertical Stabilization System for the EAST tokamak," in *2022 American Control Conf. (ACC 2022)*, 2022.

[163] M. Walker *et al.*, "Assessment of controllers and scenario control performance for ITER first plasma," *Fusion Engineering and Design*, vol. 146, pp. 1853–1857, 2019.

[164] G. De Tommasi *et al.*, "System-Engineering approach for the ITER PCS design: The correction coils current controller case study," vol. 185, pp. 1–6, 2022.

[165] G. Raupp *et al.*, "Event generation and simulation of exception handling with the ITER PCSSP," *Fusion Engineering and Design*, vol. 89, pp. 523–528, 2014.

[166] G. De Tommasi, F. Maviglia, A. Neto, P. Lomas, P. McCullen, and F. G. Rimini, "Plasma position and current control system enhancements for the JET ITER-like wall," *Fusion Engineering and Design*, vol. 89, pp. 233–242, 2014.

[167] Valcárcel *et al.*, "The JET real-time plasma-wall load monitoring system," *Fusion Engineering and Design*, vol. 89, pp. 243–258, 2014.

[168] A. Barbalace, G. Manduchi, A. Neto, G. De Tommasi, F. Sartori, and D. F. Valcarcel, "Performance comparison of epics ioc and marte in a hard real-time control application," *IEEE Transactions on Nuclear Science*, vol. 58, pp. 3162–3166, 2011.

[169] A. Murari *et al.*, "Adaptive predictors based on probabilistic SVM for real time disruption mitigation on JET," *Nuclear Fusion*, vol. 58, no. 5, p. 056002, 2018.

[170] J. Vega *et al.*, "Disruption prediction with artificial intelligence techniques in tokamak plasmas," *Nature Physics*, pp. 741–750, 2022.

[171] A. Hemmer, M. Abderrahim, R. Badonnel, J. François, and I. Chrisment, "Comparative Assessment of Process Mining for Supporting IoT Predictive Security," *IEEE Transactions on Network and Service Management*, vol. 18, pp. 1092–1103, 2021.

[172] D. Ottaviano, "The Omnivisor Source Code," https://github.com/DanieleOttaviano/Omnivisor, 2024, accessed: May 7, 2024.

[173] H. Koziolek, A. Burger, and A. P. Peedikayil, “Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems,” *Journal of Systems and Software*, p. 112004, 2024.

[174] E. Missimer, R. West, and Y. Li, “Distributed real-time fault tolerance on a virtualized multi-core system,” *OSPERT 2014*, p. 17, 2014.

[175] F. A. T. Abad, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo, “Reset-based recovery for real-time cyber-physical systems with temporal safety constraints,” in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2016, pp. 1–8.

[176] D. J. Coe, J. H. Kulick, A. Milenkovic, and L. Etzkorn, “Virtualized in situ software update verification: verification of over-the-air automotive software updates,” *IEEE Vehicular Technology Magazine*, vol. 15, no. 1, pp. 84–90, 2019.

[177] D. Helms, P. Uven, and K. Grüttner, “Modular over-the-air software updates for safety-critical real-time systems,” *INSIGHT*, vol. 25, no. 4, pp. 85–88, 2022.

[178] P. K. Valsan, H. Yun, and F. Farshchi, “Taming non-blocking caches to improve isolation in multicore real-time systems,” in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–12.

[179] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, “Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications,” in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, 2022, pp. 184–195.

[180] Xilinx, “AXI Traffic Generator LogiCORE IP Product Guide (PG125),” https://docs.xilinx.com/r/en-US/pg125-axi-traffic-gen/Introduction, [Accessed 12-02-2024].

[181] "Remote Processor Framework; The Linux Kernel documentation — docs.kernel.org," https://docs.kernel.org/staging/remoteproc.html, [Accessed 07-02-2024].

[182] dSpace, "SCALEXIO System," [Accessed 20-09-2024].

[183] H. Akkan, M. Lang, and L. Liebrock, "Understanding and isolating the noise in the linux kernel," *The International journal of high performance computing applications*, vol. 27, no. 2, pp. 136–146, 2013.

[184] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, "Mempol: polling-based microsecond-scale per-core memory bandwidth regulation," *Real-Time Systems*, pp. 1–44, 2024.

[185] N. Bao *et al.*, "A real-time disruption prediction tool for VDE on EAST," *IEEE Trans. Plasma Sci.*, vol. 48, no. 3, pp. 715–720, 2020.

[186] A. Kalantar, Z. Zimmerman, and P. Brisk, "Fpga-based acceleration of time series similarity prediction: From cloud to edge," *ACM TRETS*, vol. 16, no. 1, pp. 1–27, 2022.

[187] M. Straesser, A. Bauer, R. Leppich, N. Herbst, K. Chard, I. Foster, and S. Kounev, "An empirical study of container image configurations and their impact on start times," in *Proc. CCGrid*. IEEE, 2023, pp. 94–105.