The Omnivisor: A Real-Time Static Partitioning Hypervisor Extension for Heterogeneous Core Virtualization over MPSoCs

Daniele Ottaviano ✉ 🆔
Università degli Studi di Napoli Federico II, Italy

Francesco Ciraolo ✉ 🆔
Boston University, MA, USA

Renato Mancuso ✉ 🆔
Boston University, MA, USA

Marcello Cinque ✉ 🆔
Università degli Studi di Napoli Federico II, Italy

───── **Abstract** ─────

Following the needs of industrial applications, virtualization has emerged as one of the most effective approaches for the consolidation of mixed-criticality systems while meeting tight constraints in terms of space, weight, power, and cost (SWaP-C). In embedded platforms with homogeneous processors, a wealth of works have proposed designs and techniques to enforce spatio-temporal isolation by leveraging well-understood virtualization support. Unfortunately, achieving the same goal on heterogeneous MultiProcessor Systems-on-Chip (MPSoCs) has been largely overlooked. Modern hypervisors are designed to operate exclusively on main cores, with little or no consideration given to other co-processors within the system, such as small microcontroller-level CPUs or soft-cores deployed on programmable logic (FPGA). Typically, hypervisors consider co-processors as I/O devices allocated to virtual machines that run on primary cores, yielding full control and responsibility over them. Nevertheless, inadequate management of these resources can lead to spatio-temporal isolation issues within the system. In this paper, we propose the Omnivisor model as a paradigm for the holistic management of heterogeneous platforms. The model generalizes the features of real-time static partitioning hypervisors to enable the execution of virtual machines on processors with different Instruction Set Architectures (ISAs) within the same MPSoC. Moreover, the Omnivisor ensures temporal and spatial isolation between virtual machines by integrating and leveraging a variety of hardware and software protection mechanisms. The presented approach not only expands the scope of virtualization in MPSoCs but also enhances the overall system reliability and real-time performance for mixed-criticality applications. A full open-source reference implementation of the Omnivisor based on the Jailhouse hypervisor is provided, targeting ARM real-time processing units and RISC-V soft-cores on FPGA. Experimental results on real hardware show the benefits of the solution, including enabling the seamless launch of virtual machines on different ISAs and extending spatial/temporal isolation to heterogenous cores with enhanced regulation policies.

1   Introduction

The current approach to address the increasing number of functional requirements in industries that deal with safety-critical systems such as automotive [8], avionics [73], and nuclear fusion [25] is toward an integrated development model rather than a federated one, where several services with varying degrees of criticality coexist on shared hardware platforms. These software architectures are usually referred to as mixed-criticality systems (MCSs) [18, 17]. Developing mixed-criticality systems on multiprocessor architectures to reduce the size, weight, power, and cost (SWaP-C) is a challenge that, despite strong community interest, has not yet found a unique standard solution [3, 45, 19]. Among the proposed approaches, one of the most valuable in the scope of high-performance real-time systems is the use of real-time virtualization [22]. While traditional virtualization is a feature-rich technology that allows efficient resource utilization, real-time virtualization leans toward minimalist architectures focusing on safety, security, and predictability. In the wide spectrum of real-time virtualization technologies [23], the minimal and safest virtualization approach is static partitioning [64]. This partitioning-based approach is suitable for critical systems where the lack of determinism can significantly increase the validation and certification cost.

This virtualization approach has shown outstanding isolation performance in symmetric multi-core architectures, as evidenced in recent studies [47]. However, alongside symmetric platforms, asymmetric architectures are increasingly gaining traction in the market; the complexity and heterogeneity of multi-core systems and Commercial Off-The-Shelf (COTS) boards are gradually increasing to meet the requirements of bleeding-edge industrial applications. Therefore, we are currently witnessing the growing adoption of asymmetric MultiProcessor Systems-on-Chip (MPSoCs) in various industrial applications from automotive [8, 16, 65, 39], to avionics [73], and nuclear fusion [25, 7]. With the increase in hardware complexity within these systems, the already well-known challenges with predictability and security are further exacerbated. Modern MPSoCs, such as AMD/Xilinx Zynq UltraScale+ [78], Versal [77], NVIDIA Orin [56] and Xavier [57], Google Coral [33] and NXP i.MX8 [58], embed a heterogeneous set of processing elements. These include general-purpose microprocessor-level CPUs, sometimes referred to as Application Processing Units (APUs), and microcontroller-level CPUs that are simpler and more predictable, such as those within the ARM Cortex-M/R families. Additionally, some of these systems incorporate accelerators (e.g., Graphical Processing Units – GPUs, and/or Tensor Processing Units – TPUs), and, in some cases, also Field-Programmable Gate Arrays (FPGAs), that is, re-programmable hardware capable of integrating various types of special-purpose accelerators or additional cores (e.g., RISC-V soft-cores). All of these processing elements in the system are intricately interconnected and share numerous platform resources. From now on, to be consistent with ARM's terminology, we will utilize the term *"managers"* to denote all hardware capable of initiating memory transactions. Additionally, we will refer to all cores that are not general-purpose application cores (main cores), as *"remote cores"* to be compliant with the terminology used by Linux (e.g. `remoteproc` driver [44]).

To provide code running on such complex architectures with real-time guarantees, researchers have focused on mitigating temporal interference due to resource contention across MPSoCs. Over the years, considerable effort has been invested in exerting control over

the memory hierarchy, including the last-level cache [42], DDR memory [80], and memory controller [82]. Significant attention has also gone into minimizing interrupt latency [29] and managing the sharing of memory channels among modules in the programmable logic [27]. However, comparatively less attention has gone into the inherent limitation of static partitioning hypervisors in efficiently managing heterogeneous platforms. Specifically, modern architectures present cores that manufacturers provide ad-hoc to execute specialized software. Examples include Real-Time Processing Units (RPUs) used to run critical applications and Deep Learning Processing Units (DPUs) used to improve the performance of AI applications. In a mixed-criticality system, we expect the execution time of code running on RPUs to remain unaffected by other independent applications, such as AI workload running on DPUs.

Currently, remote cores are not managed by the hypervisors in the same way as the main CPUs; rather, these cores are either ignored entirely or, at best, treated as I/O devices allocated to virtual machines (VMs) running on primary cores. This means a VM controlling one or more remote cores can load and execute any code on them. Unfortunately, a remote processing core usually possesses enough privileges to access critical platform resources, becoming a threat to the other VMs running on the board from a spatial and temporal isolation point of view. In contrast, a hypervisor designed for heterogeneous MPSoCs should:

- Offer a unified and transparent interface to the user to flexibly deploy virtual machines on any core within the platform, regardless of the Instruction Set Architecture (ISA).
- Guarantee comprehensive spatial and temporal isolation between VMs across the platform.

Research Question. The question that inspired this paper is: *Can next-generation real-time static partitioning hypervisors adapt to the evolving landscape of modern heterogeneous platforms? Specifically, can they offer seamless and flexible mechanisms for deploying VMs across heterogeneous processing cores, all while ensuring robust isolation guarantees for mixed-criticality deployment?*

Contribution. To tackle such a question, in this paper, we propose the *Omnivisor* model. This model extends the traditional static partitioning hypervisor paradigm to take control over heterogeneous cores in MPSoCs platforms. Thus, we make the following contributions:

- We propose a novel model that generalizes the features of real-time static partitioning hypervisors to integrate the management of heterogeneous cores, improving their flexibility and usability in MPSoCs platforms.
- We show how a combination of various hardware-software protection mechanisms can be seamlessly orchestrated at runtime by our Omnivisor to ensure high isolation between VMs running on heterogeneous cores.
- We provide an open source reference implementation [61] and an evaluation of the proposed model on a COTS board (AMD/Xilinx's UltraScale+) by extending Jailhouse, a real-time static partitioning hypervisor, to run virtual machines over remote cores with different ISAs (Aarch32 RPUs and RISC-V soft-cores).

Experimental results on the board show that a user can seamlessly launch a VM on heterogeneous cores via the Ominivisor with comparable boot times. These experiments highlight the Omnivisor's flexibility which enables compelling scenarios such as real-time live migration [41], reboot after failure [51], system rejuvenation [1], and over-the-air (OTA) updates [28, 36]. Experiments also demonstrate the isolation capabilities of the Ominivisor by executing critical workload on remote cores in the presence of severe disturbances generated by the other cores and the FPGA on the same board. Finally, by using realistic benchmarks, we show how the Ominivisor can enforce a controlled degradation policy to keep real-time guarantees while not limiting the overall system performance.

Paper Structure.   In Sec. 2, we review modern hardware protection mechanisms on MPSoCs and discuss traditional hypervisor models' limitations. Sec. 3 introduces the Omnivisor model, highlighting its benefits and differences from traditional models. We also discuss Omnivisor's requirements, responsibilities, and features. In Sec. 4, we walk through the implementation of the Omnivisor on a Xilinx Ultrascale+ board, assessing strengths and weaknesses. Sec. 5 and 6 present experimental analysis and practical use cases. Sec. 7 compares Omnivisor with related works.Conclusive remarks and future works are provided in Sec. 8.

2   Background and Motivations

Considering the high heterogeneity of processing elements deployed on MPSoCs that act as managers – i.e., heterogeneous CPUs, GPUs, DMAs, 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.

This section aims 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. Following that, we explain how traditional static partitioning hypervisors utilize these mechanisms only to a limited extent, highlighting why this presents a significant constraint compared to the extensive capabilities provided by modern COTS platforms.

2.1   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 (MMU-based, see Sec. 2.2.1). 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 Application Processing Units (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 (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 Real-Time Processing Units (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 the SMMU (see Sec. 2.2.1). This category includes soft-cores such as the AMD MicroBlaze [5], or the RISC-V Pico32 [79].

2.2 MPSoCs Protection Mechanisms

The MPSoCs protection mechanisms can be systematically categorized as follows.

2.2.1 Spatial Isolation

Address Translation (MMU/SMMU).   The Memory Management Unit (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. 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. 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. Microcontroller-level CPUs typically employed to run bare-metal software or Real-Time Operating Systems (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 [62]. In these scenarios, CPUs are equipped with more straightforward mechanisms known as Memory Protection Units (MPUs). These are implemented as hardware tables deployed between the manager (CPU) and the subordinate (Memory). Using the tables, an MPU enforces specific permissions to fixed address space regions.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 (SMPUs) when used to protect memory; we use the term System Peripheral Protection Units (SPPUs) when they are used to protect memory-mapped I/O.

2.2.2 Temporal Isolation

Hardware Bandwidth Allocation.   In modern ARM-based platforms, Quality of Service (QoS) support offers a mechanism to manage memory traffic at the level of bus managers. 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 regulation in QoS-aware on-chip memory. 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 [67, 32]. However, there is a common trend in existing QoS-enabled platforms [69]: 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 [81] [82].

2.3   From Traditional to Static Partitioning Hypervisors

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 [40], Xen [11], and many others. These are widely used, and researchers have extended their capabilities to accommodate various use cases, including real-time scenarios [2, 30].

Static Partitioning hypervisors.   Real-time static partitioning hypervisors (SPHs), such as Jailhouse [63], Bao [48], Xtratum [49], and Quest-V [74], 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.

2.4   SPH Shortcoming over Asymmetric MPSoCs

SPHs 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 other 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.

3 The Omnivisor

In this paper, we introduce the Omnivisor, a novel hypervisor model that generalizes static partitioning hypervisors to enable the transparent execution of VMs on heterogeneous cores over commercial off-the-shelf (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.

Model Purpose. As depicted in Fig. 1, while conventional hypervisors are designed to manage microprocessor-level CPUs, our 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 spatially, 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 Omivisor, 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.

3.1 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.
- *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.
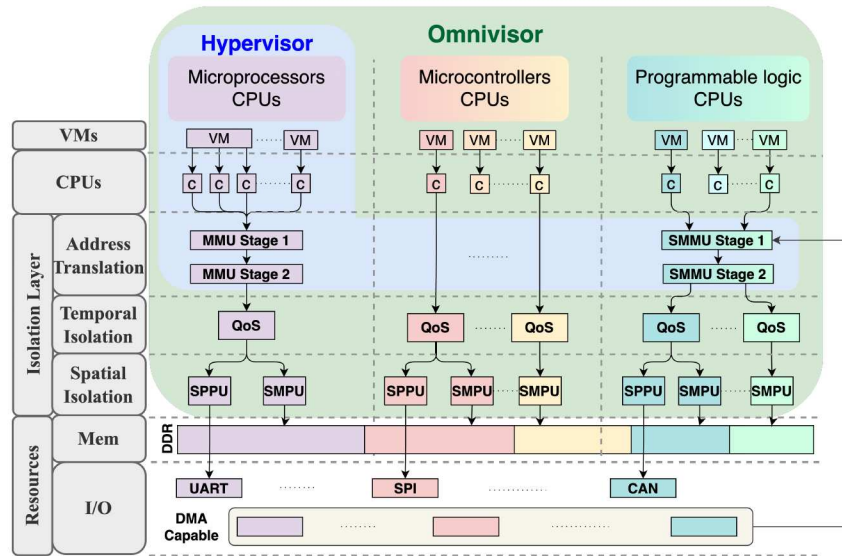
Figure 1 A block diagram illustrates the Omnivisor model, showcasing varied temporal and spatial isolation mechanisms across CPU clusters, emphasizing their heterogeneity. The arrows indicate the flow of a request from an initiator to the accessed resource (memory or I/O).

- *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 heterogeneous cores power management.

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 [78, 77, 58], tailored to meet industrial demands.

3.2   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
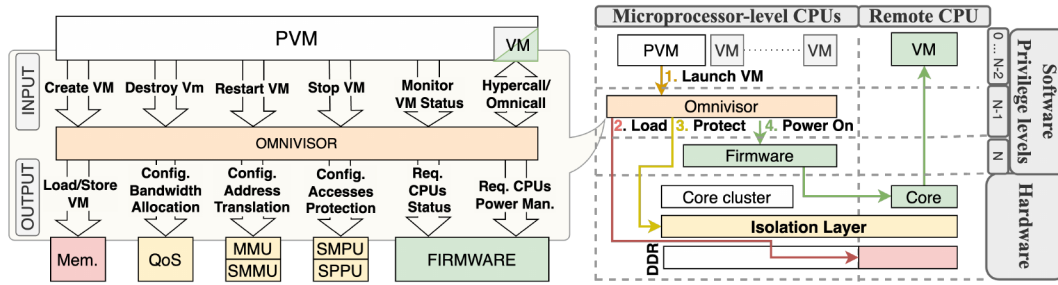
Figure 2 Omnivisor feature set (left) and remote core VM startup process (right).

SMMU, the Omnivisor must leverage SMPUs to shield the resources from the VMs running on microcontroller-level CPUs. Consequently, as shown in Fig.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 Fig.2 (right) are color-coded based on the operation and are enumerated in temporal order.

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 Fig. 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.

3.3 Features

The Omnivisor provides a set of features that includes that of the traditional SPHs while expanding them to encompass heterogeneous processing elements (see Fig. 2). Given the diversity among existing hypervisors, 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 VM* (PVM), which is a known concept in hypervisor's literature [54], and is the only VM with the ability to manage other VMs. A few examples are the *root-cell* in Jailhouse [63], and the

Dom0 in Xen [70]. The Omnivisor provides the PVM with the same interface for managing VMs for both the main and remote cores. For instance, as shown in Fig. 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 Interrupts (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. [66], 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.

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, remaining 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   Omnivisor Implementation

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 [63] because it has low overhead [47] while maintaining an easy-to-use interface to manage VMs at runtime. Furthermore, the Jailhouse-RT branch, overseen by Minerva Systems [50], already implements MemGuard-like regulators for the APUs, page coloring, and basic SMMU drivers. It also provides a rudimental interface to control ARM Quality of Service (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 Sec. 3.1: 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 Xilinx Memory Protection Units (XMPU) and Xilinx Peripherals Protection Units (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 shut-down, 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.

4.1   Omnivisor Usage Workflow

One of the key objectives of the Omnivisor is to simplify the utilization of complex heterogeneous architectures for users. Therefore, the Omnivisor 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 [79]). 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.

4.1.1   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 Fig. 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 after compilation will consist of the Omnivisor binary along with the binary images for the VMs.

4.1.2   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 Sec. 4. Different from the vanilla Jailhouse, the configuration in our Omnivisor may also include a field for the *rCPUs*. If
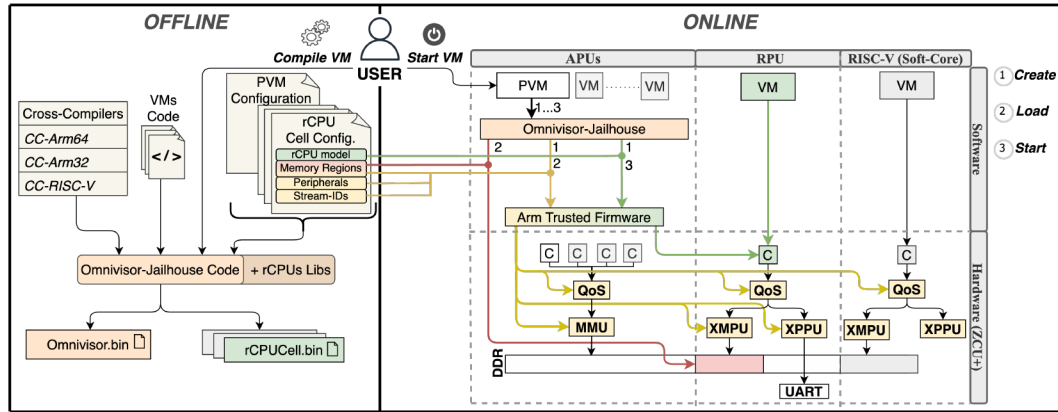
Figure 3 Architectural view of VM compiling and start procedures using the Omnivisor implementation on top of Jailhouse and the Zynq Ultrascale+ board.

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 Fig. 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 [50]. 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 PSCI* [6] or the *Intel ACPI* [38]. 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.

5   Use Cases

In this section, we report a few use cases that inspired us toward the creation of the Omnivisor.

Real-time control in nuclear fusion power reactors. Nuclear fusion is foreseen as a promising clean energy source for the next century, and the ITER tokamak reactor (iter.org) is set to be the first fusion device with a net-positive energy output. In a tokamak, magnetic confinement of the plasma is achieved using several magnetic fields generated by the electric current that flows in an array of external coils. These currents are controlled by the so-called plasma control system (PCS) [68], which is a complex and multi-input-multi-output control system. The PCS includes several subcomponents, each aiming to control a specific plasma feature with different requirements in terms of reliability, latency, and needed computational resources. The ITER project intends to use MPSoCs [7] to run multiple control loops and signal conditioning algorithms with different sampling times and reliability requirements on the same system [60]. Being an experimental facility, one of the missions of ITER is to test the efficiency of advanced control schemes, e.g., using reinforcement learning, running side-by-side with basic control loops, for safety reasons.

The use of the Omnivisor in this context can speed up the development and testing phases by enabling the deployment of advanced and computationally heavy control algorithms, launched as VMs on the APUs, along with stable safety controllers, launched as VMs on RPUs or soft-cores, while assuring spatial and temporal isolation between them.

MPSoCs for advanced system management research. Researchers adopt heterogeneous architectures to run computation-intensive applications in safety-critical [19] and mission-critical scenarios, such as vision control units for self-driving vehicles [20]. Moreover, the real-time community has shown significant interest in leveraging MPSoCs resources, like remote cores, for monitoring and management. Executing monitoring or management tasks on the same platform as the monitored applications can introduce overhead and interference,

while remote monitoring (e.g., over a network connection) suffers from communication latency [24]. Therefore, utilizing on-board resources, when available, is a good compromise. For instance, the work described in [82] utilizes RPUs on a ZCU+ to finely monitor memory transactions and control the bandwidth of APUs using the board's debug infrastructure. In [32], instead, authors employ QoS setups on the memory controller to ensure high-degree isolation of critical applications across heterogeneous cores. Additionally, in [21], the progress of a critical application running on APUs is monitored by an RPU on a ZCU+ to provide online regulation based on the application's state.

Integrating an Omnivisor can greatly simplify utilizing these cutting-edge mechanisms in real-world industrial scenarios by providing an easy way to deploy and isolate the applications. Furthermore, it can speed up the experimental phase for researchers aiming to implement complex applications on heterogeneous platforms.

6 Experimental Analysis

In this section, we provide an evaluation of the Omnivisor model and its implementation. The reference implementation, along with a set of scripts to reproduce the experiments, is openly available as open-source software [61]. The platform under test is the ZCU+ described in Sec. 4. 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?*
- *What degree of spatio-temporal isolation does the Omnivisor guarantee for remote VMs?*
- *Can the Ominvisor be a turnkey solution to achieve controlled degradation?*

It's important to note that the additional functionalities introduced in our Omnivisor, as described in Section 4, 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 [47]. Consequently, we do not present runtime overhead results in this paper.

6.1 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 [41], reboot after failure [51], system rejuvenation [1] and OTA updates [28, 36].

Fig. 4 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 Fig. 4 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
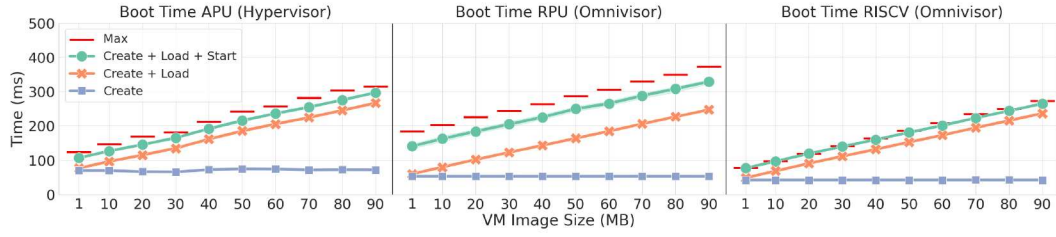
Figure 4 Comparison of boot times across heterogeneous processors in the ZCU+ Platform.

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 to the longer RPU wake-up procedure that involves both the PSCI and the PMU, as detailed in Sec. 4.1.2. 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 [35]. 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.

6.2   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.

Experiment Setup.   Fig. 5a (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.
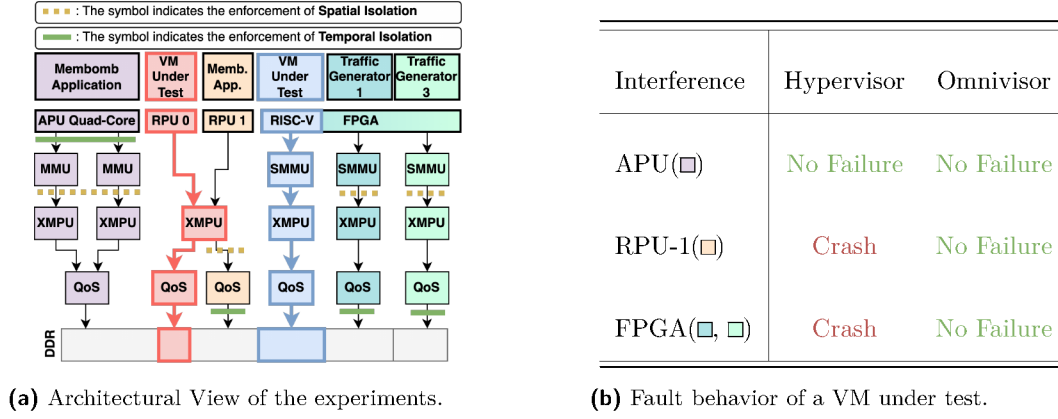
(a) Architectural View of the experiments.

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

(b) Fault behavior of a VM under test.

Figure 5 Experimental configuration (left) and expected fault outcomes (right) of VMs running on rCPU subjected to interference from various sources (APU, RPU-1, FPGA): a comparative analysis between traditional Hypervisor and Omnivisor.

Employing a traditional hypervisor, as illustrated in Table 5b, will cause applications running on remote cores (RPU, RISCV) 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 Fig. 6 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 [71] from the RT-Bench framework [55]. 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 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 [76].

We can observe the results when the RPU-1, FPGA, and APU managers are the source of interference in Figs. 6a, 6b, and 6c 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 (5b). 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 Fig. 6c, 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
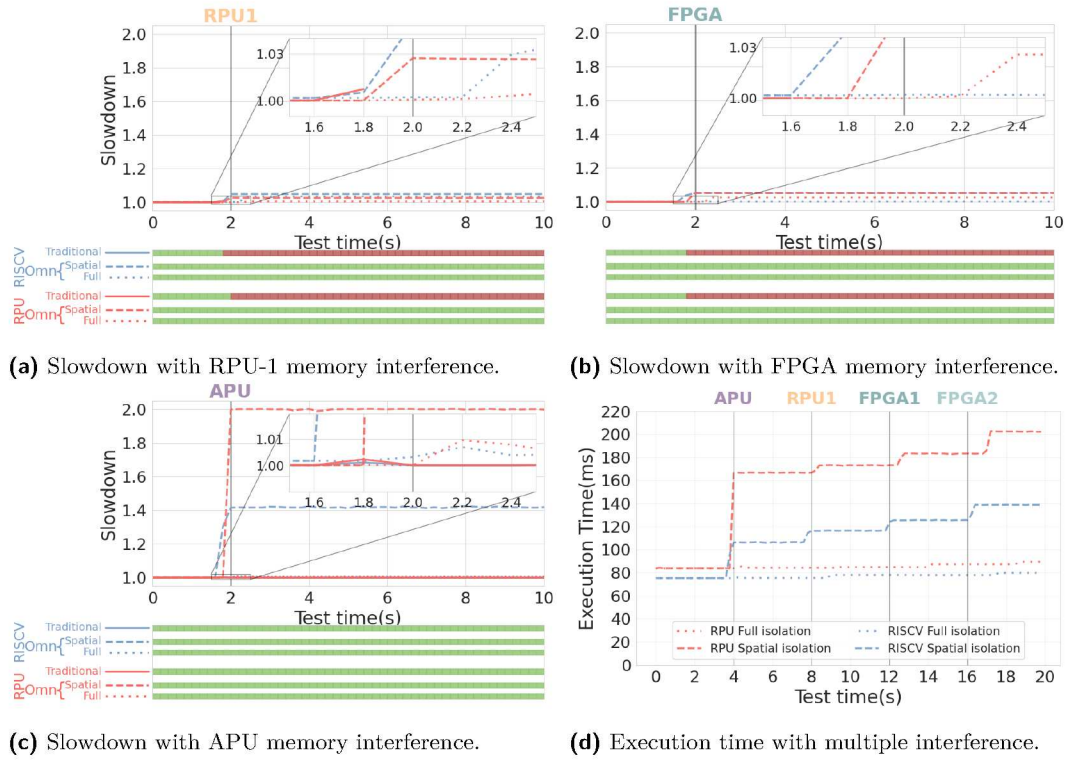
(a) Slowdown with RPU-1 memory interference.

(b) Slowdown with FPGA memory interference.

(c) Slowdown with APU memory interference.

(d) Execution time with multiple interference.

Figure 6 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.

possible with the `remoteproc` driver [44]. 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, a control application running on the RPUs might inadvertently overwrite memory used by APUs for critical log information, resulting in the loss of invaluable insights 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.   Fig. 6d 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 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 Sec. 4.1.2, 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 [67, 32, 69], in this experiment, 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 [69], 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 Fig. 6d. Specifically, the maximum slowdown drops from 142% to 7% on RPU and from 85% to 6% on RISCV.

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.

6.3 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 [31]. 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 bandwidth allocation required to ensure unrestricted memory transactions on every manager. Specifically, leveraging findings from [69] 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%. Fig. 7 presents the slowdown over thirty repetitions for each benchmark under two scenarios. First, the slowdown without any bandwidth regulation is depicted. Next, the
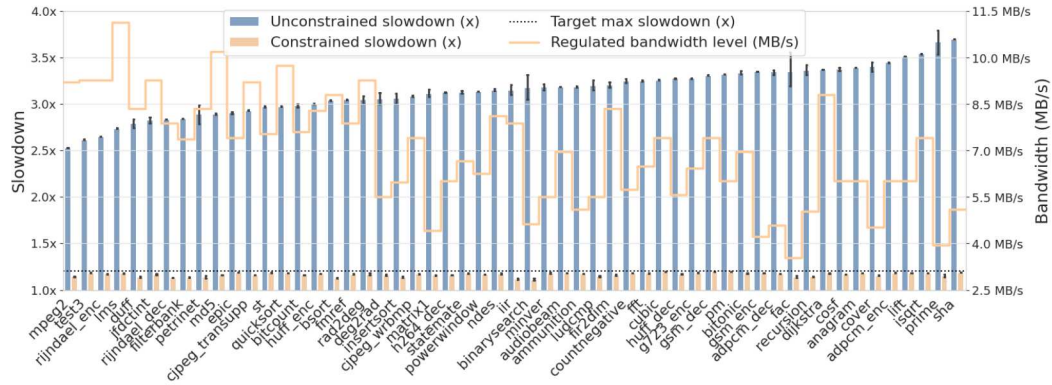
(a) Benchmarks on VMs running on RPU.
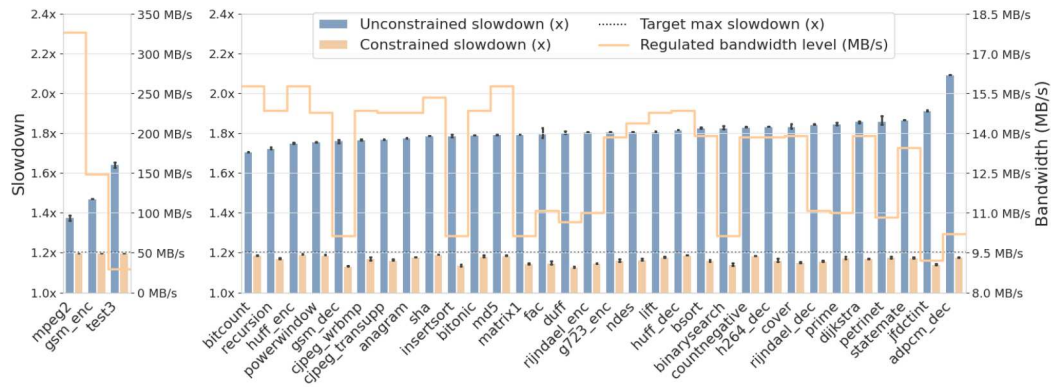
(b) Benchmarks on VMs running on RISC-V.

■ **Figure 7** 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.

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 4MB/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 [32].

7 Related

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 [72, 15]. Instead, the Omnivisor distinguishes itself by offering partitioning with spatio-temporal isolation for a diverse range of processor categories. Unlike works such as [83], which propose a partitioning microkernel-based design targeting microcontrollers-level cores, and [72], 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.

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 [9, 10, 43, 13] and hypervisors [37, 59] 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 [12] 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 [26, 60] and [4] 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.

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 [52]. 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. [34] 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 focuses specifically on addressing temporal and spatial isolation issues between asymmetric cores and it also offers flexible and seamless control over remote cores through the hypervisor. CHIPS-AHOy is a predictable holistic hypervisor [53] 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 heterogeneous VMs and neither consider using bandwidth regulation mechanisms to improve temporal isolation. Biondi *et al.* present the SPHERE project [14], 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 ISA cores as the Omnivisor, 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 [75, 46]. In contrast, Omnivisor acknowledges the presence of cores in FPGA, which run entire and isolated VMs.

Although the Omnivisor model 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.

8   Conclusions

The increasing complexity of next-generation industrial applications has led to the widespread adoption of feature-rich heterogeneous MPSoCs. However, as the number of features within a single hardware platform increases, so does the complexity of deployment and the challenges of maintaining temporal guarantees for software. In this paper, we have introduced the Omnivisor, a novel model that extends static partitioning hypervisors to manage heterogeneous processing elements within asymmetric architectures. Our experimental results have demonstrated that deploying 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 spatial and temporal isolation of VMs, achieved through a combination of software/hardware mechanisms. Additionally, we have showcased how the Omnivisor enhances the user's control over MPSoCs. Specifically, we utilized Omnivisor features to precisely regulate the degradation of a real-time virtual machine executing on a remote core.

For future research directions we intend to (1) integrate a library of remote core utilities sourced from open-source scientific works in order to enhance the monitoring and management capabilities of MPSoCs. Following this (2), we aim to elevate the flexibility of these platforms to the next level by introducing dynamic FPGA hardware reconfiguration at the hypervisor level. Our objective is to integrate the capability to reconfigure portions (tiles) of the programmable logic as an additional Omnivisor feature, enabling the instantiation of soft-cores ad-hoc and on the fly to launch a VM with specific requirements.

Overall, our work showcases the potential of the Omnivisor in addressing the challenges posed by modern industrial applications, offering a promising solution for the efficient utilization of heterogeneous MPSoCs.

References

1 Fardin Abdi Taghi Abad, Renato Mancuso, Stanley Bak, Or Dantsker, and Marco 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)*, pages 1–8. IEEE, 2016.

2 Luca Abeni and Dario Faggioli. Using xen and kvm as real-time hypervisors. *Journal of Systems Architecture*, 106:101709, 2020.

3 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. A comprehensive survey of industry practice in real-time systems. *Real-Time Systems*, 58(3):358–398, 2022.

4 Sara Alonso, Jesus Lazaro, Jaime Jimenez, Leire Muguira, and Unai Bidarte. Evaluating the OpenAMP framework in real-time embedded SoC platforms. In *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2021.

5 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].

6 ARM. PSCI specification – developer.arm.com. `https://developer.arm.com/Architectures/Power%20State%20Coordination%20Interface`. [Accessed 20-02-2024].

7 Giuseppe Avon, Arturo Buscarino, André C Neto, and Filippo Sartori. MARTe2 embedded signal processing unit for the ITER magnetics diagnostics. In *IECON 2021–47th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6. IEEE, 2021.

8 Victor Bandur, Gehan Selim, Vera Pantelic, and Mark Lawford. Making the case for centralized automotive e/e architectures. *IEEE Transactions on Vehicular Technology*, 70(2):1230–1245, 2021.

9 Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. *ACM SIGARCH Computer Architecture News*, 45(1):645–659, 2017.

10 Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

11 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

12 Felix Baum and Arvind 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.

13 Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

14 Alessandro Biondi, Daniel Casini, Giorgiomaria Cicero, Niccolò Borgioli, Giorgio Buttazzo, Gaetano Patti, Luca Leonardi, Lucia Lo Bello, Marco Solieri, Paolo Burgio, et al. SPHERE: A Multi-SoC architecture for next-generation cyber-physical systems based on heterogeneous platforms. *IEEE Access*, 9:75446–75459, 2021.

15 Gedare Bloom and Joel 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)*, pages 98–105. IEEE, 2020.

16 Paolo Burgio, Marko Bertogna, Nicola Capodieci, Roberto Cavicchioli, Michal Sojka, Přemysl Houdek, Andrea Marongiu, Paolo Gai, Claudio Scordino, and Bruno Morelli. A software stack for next-generation automotive systems on many-core heterogeneous platforms. *Microprocessors and Microsystems*, 52:299–311, 2017.

17 Alan Burns and Robert I Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys (CSUR)*, 50(6):1–37, 2017.

18 Alan Burns and Robert Ian Davis. Mixed criticality systems-a review:(february 2022). York, 2022.

19 Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys (CSUR)*, 53(4):1–38, 2020.

20   Ravikumar V Chakaravarthy, Hyun Kwon, and Hua Jiang. Vision control unit in fully self driving vehicles using Xilinx MPSoC and opensource stack. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 311–317, 2021.

21   Weifan Chen, Ivan Izhibirdeev, Denis Hoornaert, Shahin Roozkhosh, Patrick Carpanedo, Sanskriti Sharma, and Renato Mancuso. Low-overhead online assessment of timely progress as a system commodity. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

22   Alessandro Cilardo, Marcello Cinque, Luigi De Simone, and Nicola Mazzocca. Virtualization over multiprocessor system-on-chip: an enabling paradigm for industrial iot. *arXiv preprint*, 2021. arXiv:2112.15404.

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

24   Marcello Cinque, Luigi De Simone, Nicola Mazzocca, Daniele Ottaviano, and Francesco Vitale. Evaluating virtualization for fog monitoring of real-time applications in mixed-criticality systems. *Real-Time Systems*, 59(4):534–567, 2023.

25   Marcello Cinque, Gianmaria De Tommasi, Sara Dubbioso, and Daniele Ottaviano. Virtualizing real-time processing units in multi-processor systems-on-chip. In *2021 IEEE 6th International Forum on Research and Technology for Society and Industry (RTSI)*, pages 329–333. IEEE, 2021.

26   Marcello Cinque, Gianmaria De Tommasi, Sara Dubbioso, and Daniele Ottaviano. RPUGuard: Real-time processing unit virtualization for mixed-criticality applications. In *2022 18th European Dependable Computing Conference (EDCC)*, pages 97–104. IEEE, 2022.

27   Edoardo Cittadini, Mauro Marinoni, Alessandro Biondi, Giorgiomaria Cicero, and Giorgio Buttazzo. Supporting ai-powered real-time cyber-physical systems on heterogeneous platforms via hypervisor technology. *Real-Time Systems*, pages 1–27, 2023.

28   David J Coe, Jeffrey H Kulick, Aleksandar Milenkovic, and Letha Etzkorn. Virtualized in situ software update verification: verification of over-the-air automotive software updates. *IEEE Vehicular Technology Magazine*, 15(1):84–90, 2019.

29   Diogo Costa, Luca Cuomo, Daniel Oliveira, Ida Maria Savino, Bruno Morelli, Jose Martins, Fabrizio Tronci, Alessandro Biasci, and Sandro 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.

30   Domenico Cotroneo, Luigi De Simone, and Roberto 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)*, pages 1–5. IEEE, 2022.

31   Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

32   Sergio Garcia-Esteban, Alejandro Serrano-Cases, Jaume Abella, Enrico Mezzetti, and Francisco 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.

33   Google. Dev-Board Coral datasheet. https://coral.ai/docs/dev-board/datasheet. [Accessed 21-02-2024].

34   Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing mixed criticality applications on modern heterogeneous MPSoC platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

35 Yuri Gribov, Andrey Kavin, Victor Lukash, Rustam Khayrutdinov, Guido Huijsmans, Alberto Loarte, Joseph A. Snipes, and Luca Zabeo. Plasma vertical stabilisation in ITER. *Nuclear Fusion*, 55(7):073021, 2015.

36 Domenik Helms, Patrick Uven, and Kim Grüttner. Modular over-the-air software updates for safety-critical real-time systems. *INSIGHT*, 25(4):85–88, 2022.

37 Yu-Ju Huang, Hsuan-Heng Wu, Yeh-Ching Chung, and Wei-Chung 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*, pages 3–15, 2016.

38 Intel. ACPI specification – intel.com. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/processor-vendor-specific-acpi-specification.pdf`. [Accessed 20-02-2024].

39 Shravan Karthik, Karthik Ramanan, Nikhil Devshatwar, Subhajit Paul, Vishal Mahaveer, Sheng Zhao, Manoj Vishwanathan, and Chetan Matad. Hypervisor based approach for integrated cockpit solutions. In *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*, pages 1–6. IEEE, 2018.

40 Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

41 Heiko Koziolek, Andreas Burger, and Abdulla Puthan Peedikayil. Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems. *Journal of Systems and Software*, page 112004, 2024.

42 Yoojin Lim and Hyoseung Kim. Cache-aware real-time virtualization for clustered multi-core platforms. *IEEE Access*, 7:128628–128640, 2019.

43 Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. *ACM SIGPLAN Notices*, 49(4):285–300, 2014.

44 Remote Processor Framework; The Linux Kernel documentation — docs.kernel.org. `https://docs.kernel.org/staging/remoteproc.html`. [Accessed 07-02-2024].

45 Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022.

46 Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris 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*, pages 827–844, 2020.

47 José Martins and Sandro 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)*, pages 40–53. IEEE, 2023.

48 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro 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.

49 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, volume 9. Citeseer, 2009.

50 Minerva. Jailhouse-RT GitLab repository. `https://gitlab.com/minervasys/public/jailhouse/-/tree/minerva/public?ref_type=heads`. [Accessed 08-02-2024].

51 Eric Missimer, Richard West, and Ye Li. Distributed real-time fault tolerance on a virtualized multi-core system. *OSPERT 2014*, page 17, 2014.

52 Carlos Moratelli, Samir Zampiva, and Fabiano 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*, pages 1–7, 2014.

53   Tiago Mück, Antonio A Fröhlich, Giovani Gracioli, Amir M Rahmani, João Gabriel Reis, and Nikil 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*, pages 73–80, 2018.

54   Djob Mvondo, Boris Teabe, Alain Tchana, Daniel Hagimont, and Noel 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)*, pages 49–60. IEEE, 2019.

55   Mattia Nicolella, Shahin Roozkhosh, Denis Hoornaert, Andrea Bastoni, and Renato 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*, pages 184–195, 2022.

56   NVIDIA. Jetson AGX Orin technical brief. `https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf`. [Accessed 21-02-2024].

57   NVIDIA. Jetson Xavier Series. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/`. [Accessed 21-02-2024].

58   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].

59   Pierre Olivier, Binoy Ravindran, and Antonio Barbalace. The multihype: Virtualizing heterogeneous-ISA architectures. In *9th Workshop on Systems for Multi-core and Heterogeneous Architectures (SFMA)*, 2019.

60   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.

61   Daniele Ottaviano. The Omnivisor Source Code. `https://github.com/DanieleOttaviano/Omnivisor`, 2024. Accessed: May 7, 2024.

62   Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via tlb coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13. IEEE, 2015.

63   Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look mum, no vm exits!(almost). *arXiv preprint*, 2017. `arXiv:1705.06932`.

64   Ralf Ramsauer, Jan Kiszka, and Wolfgang Mauerer. A novel software architecture for mixed criticality systems. In *Digital Transformation in Semiconductor Manufacturing: Proceedings of the 1st and 2nd European Advances in Digital Transformation Conference, EADTC 2018, Zittau, Germany and EADTC 2019, Milan, Italy*, pages 121–128. Springer International Publishing, 2020.

65   Falk Rehm, Jörg Seitter, Jan-Peter Larsson, Selma Saidi, Giovanni Stea, Raffaele Zippo, Dirk Ziegenbein, Matteo Andreozzi, and Arne Hamann. The road towards predictable automotive high-performance platforms. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1915–1924. IEEE, 2021.

66   Gero Schwäricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alexander Zuepke, and Marco Caccamo. A real-time VirtIO-based framework for predictable inter-VM communication. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 27–40. IEEE, 2021.

67   Alejandro Serrano-Cases, Juan M Reina, Jaume Abella, Enrico Mezzetti, and Francisco 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.

68   J. A. Snipes et al. ITER plasma control system final design and preparation for first plasma. *Nuclear Fusion*, 2021.

69 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms. *Real-Time Systems*, 58(3):235–274, 2022.

70 Stefano Stabellini. Xen project blog. `https://xenproject.org/2019/12/16/true-static-partitioning-with-xen-dom0-less/`, 2019. [Accessed 27-02-2024].

71 Prathap Kumar Valsan, Heechul Yun, and Farzad 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)*, pages 1–12. IEEE, 2016.

72 Steven H VanderLeest. Designing a future airborne capability environment (face) hypervisor for safety and security. In *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pages 1–9. IEEE, 2017.

73 Steven H VanderLeest and Dagan White. Mpsoc hypervisor: The safe & secure future of avionics. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 6B5–1. IEEE, 2015.

74 Richard West, Ye Li, Eric Missimer, and Matthew Danish. A virtualized separation kernel for mixed-criticality systems. *ACM Transactions on Computer Systems (TOCS)*, 34(3):1–41, 2016.

75 Tian Xia, Ye Tian, Jean-Christophe Prévotet, and Fabienne Nouvel. Ker-one: A new hypervisor managing fpga reconfigurable accelerators. *Journal of Systems Architecture*, 98:453–467, 2019.

76 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].

77 Xilinx. Versal Device Technical Reference Manual. `https://docs.xilinx.com/r/en-US/am011-versal-acap-trm`. [Accessed 07-02-2024].

78 Xilinx. Zynq Ultrascale+ Device Technical Reference Manual. `https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm`. [Accessed 07-02-2024].

79 YosysHQ. picorv32. `https://github.com/YosysHQ/picorv32`. [Accessed 27-02-2024].

80 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, 2014.

81 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui 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)*, pages 55–64. IEEE, 2013.

82 Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. MemPol: Policing core memory bandwidth from outside of the cores. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–248. IEEE, 2023.

83 Alexander Zuepke, Marc Bommert, and Daniel Lohmann. Autobest: a united autosar-os and arinc 653 kernel. In *21st IEEE real-time and embedded technology and applications symposium*, pages 133–144. IEEE, 2015.