electronics

MDPI

Article

μRTZVisor: A Secure and Safe Real-Time Hypervisor

José Martins †, João Alves †, Jorge Cabral, Adriano Tavares and Sandro Pinto *

Centro Algoritmi, Universidade do Minho, 4800-058 Guimarães, Portugal; jose.martins@dei.uminho.pt (J.M.); joao.alves@dei.uminho.pt (J.A.); jcabral@dei.uminho.pt (J.C.); atavares@dei.uminho.pt (A.T.)
* Correspondence: sandro.pinto@dei.uminho.pt; Tel.: +351-253-510-180
† These authors contributed equally to this work.

Received: 29 September 2017; Accepted: 24 October 2017; Published: 30 October 2017

Abstract: Virtualization has been deployed as a key enabling technology for coping with the ever growing complexity and heterogeneity of modern computing systems. However, on its own, classical virtualization is a poor match for modern endpoint embedded system requirements such as safety, security and real-time, which are our main target. Microkernel-based approaches to virtualization have been shown to bridge the gap between traditional and embedded virtualization. This notwithstanding, existent microkernel-based solutions follow a highly para-virtualized approach, which inherently requires a significant software engineering effort to adapt guest operating systems (OSes) to run as userland components. In this paper, we present μRTZVisor as a new TrustZone-assisted hypervisor that distinguishes itself from state-of-the-art TrustZone solutions by implementing a microkernel-like architecture while following an object-oriented approach. Contrarily to existing microkernel-based solutions, μRTZVisor is able to run nearly unmodified guest OSes, while, contrarily to existing TrustZone-assisted solutions, it provides a high degree of functionality and configurability, placing strong emphasis on the real-time support. Our hypervisor was deployed and evaluated on a Xilinx Zynq-based platform. Experiments demonstrate that the hypervisor presents a small trusted computing base size (approximately 60KB), and a performance overhead of less than 2% for a 10 ms guest-switching rate.

Keywords: virtualization; hypervisor; TrustZone; microkernel; security; safety; real-time; Arm

1. Introduction

Embedded systems were, for a long time, single-purpose and closed systems, characterized by hardware resource constraints and real-time requirements. Nowadays, their functionality is ever-growing, coupled with an increasing complexity that is associated with a higher number of bugs and vulnerabilities [1,2]. Moreover, the pervasive connectivity of these devices in the modern Internet of Things (IoT) era significantly increases their attack surface [3,4]. Due to their myriad of applications and domains, ranging from consumer electronics to aerospace control systems, there is an increasing reliance on embedded devices that often have access to sensitive data and perform safety-critical operations [5–7]. Thus, two of the main challenges faced by modern embedded systems are those of security and safety. Virtualization emerged as a natural solution, due to the isolation and fault-containment it provides, by encapsulating each embedded subsystem in its own virtual machine (VM). This technology also allows for different applications to be consolidated into one single hardware platform, thus reducing size, weight, power and cost (SWaP-C) budgets, at the same time providing an heterogeneous operating system (OS) environment fulfilling the need for high-level programming application programming interfaces (API) coexisting alongside real-time functionality and even legacy software [5,8,9].

Despite the differences among several embedded industries, all share an increasing interest in exploring virtualization mainly for isolation and system consolidation. For example, in consumer electronics, due to current smartphone ubiquity and IoT proliferation, virtualization is being used to isolate the network stack from other commodity applications or GUI software, as well as for segregating sensitive data (e.g., companies relying on mobile phones isolate enterprise from personal data) [5,10,11]. In modern industrial control systems (ICSs), there is an increasing trend for integrating information technology (IT) with the operation technology (OT). In this context, ICSs need to guarantee functionality isolation, while protecting their integrity against unauthorized modification and restricting access to production related data [12,13]. The aerospace and the automotive industries are also dependent on virtualization solutions due to the amount of needed control units, which, prior to virtualization application, would require a set of dedicated microcontrollers, and which are currently being consolidated into one single platform. In the case of aerospace systems, virtualization allows this consolidation to guarantee the Time and Space Partitioning (TSP) required for the reference Integrated Modular Avionics (IMA) architectures [14]. As for the automotive industry, it also allows for safe coexistence of safety-critical subsystems with real-time requirements and untrusted ones such as infotainment applications [6]. Finally, in the development of medical devices, which are becoming increasingly miniaturized, virtualization is being applied to consolidate their subsystems and isolate their critical life-supporting functionality from communication or interface software used for their control and configuration, many times operated by the patient himself. These systems are often composed of large software stacks and heavy OSes containing hidden bugs and that, therefore, cannot be trusted [5].

Virtualization software, dubbed virtual machine monitor (VMM) or hypervisor, must be carefully designed and implemented since it constitutes a single point of failure for the whole system [15]. Hence, this software, the only one with maximum level of privilege and full access to all system resources, i.e., the trusted computing base (TCB), should be as small and simple as possible [16,17]. In addition, virtualization on its own is a poor match for modern embedded systems, given that for their modular and inter-cooperative nature, the strict confinement of subsystems interferes with some fundamental requirements [1,8,9]. First, the decentralized, two-level hierarchical scheduling inherent to virtualization interferes with its real-time capabilities [18,19]. In addition, the strongly isolated virtual machine model prevents embedded subsystems to communicate in an efficient manner.

Microkernel-based approaches to virtualization have been shown to bridge the gap between traditional virtualization and current embedded system requirements. Contrarily to monolithic hypervisors, which implement the VM abstraction and other non-essential infrastructure such as virtual networks in a single, privileged address space, microkernels implement a minimal TCB by only providing essential policy-void mechanisms such as thread, memory and communication management, leaving all remaining functionality to be implemented in userland components [20,21]. By employing the principles of minimality and of the least authority, this architecture has proven to be inherently more secure [22,23] and a great foundation to manage the increasing complexity of embedded systems [24,25]. However, existent microkernel-based solutions follow a highly para-virtualized approach. OSes are hosted as user-level servers providing the original OS interfaces and functionality through remote procedure calls (RPC), and, thus, these must be heavily modified to run over the microkernel [26,27]. Microkernel-based systems also seem to be well-suited for real-time environments, and, because of their multi-server nature, the existence of low-overhead inter-partition communication (IPC) is a given [28,29].

Aware of the high overhead incurred by trap-and-emulate and para-virtualization approaches, some processor design and manufacturing companies have introduced support for hardware virtualization to their architectures such as Intel's VT-x (Intel, Santa Clara, CA, USA) [30,31], Arm's VE (ARM Holdings, Cambridge, England, United Kingdom) [32,33] or Imagination's MIPS VZ (Imagination Technologies, Kings Langley, Hertfordshire, United Kingdom) [34]. These provide a number of features to ease the virtualization effort and minimize hypervisor size such as the introduction of higher privilege modes, configuration hardware replication and multiplexing, two-level address translation or virtual interrupt support. Arm TrustZone hardware extensions, although being a security oriented technology, provide similar features to those aforementioned. However, they do not provide two-level address translation but only memory segmentation support. Hence, although guests can run nearly unmodified, they need to be compiled and cooperate to execute in the confinement of their attributed segments. Despite this drawback and given that this segmented memory model is likely to suffice for embedded use-cases (small, fixed number of VMs), these extensions have been explored to enable embedded virtualization, also driven by the fact that its deployment is widely spread in low-end and mid-range microprocessors used in embedded devices [35–40]. Existing TrustZone virtualization solutions show little functionality and design flexibility, consistently employing a monolithic structure. Despite taking advantage of security extensions, TrustZone hypervisors focus exclusively on virtualization features leaving aside important security aspects.

Under the light of the above arguments, in this work, we present μRTZVisor as a new TrustZone-assisted hypervisor that distinguishes itself from existing TrustZone-assisted virtualization solutions by implementing a microkernel-like architecture while following an object-oriented approach. The μRTZVisor's security-oriented architecture provides a high degree of functionality and configuration flexibility. It also places strong emphasis on real-time support while preserving the close to full-virtualized environment typical of TrustZone hypervisors, which minimizes the engineering effort needed to support unmodified guest OSes. With μRTZVisor, we make the following contributions:

1. A heterogeneous execution environment supporting coarse-grained partitions, destined to run guest OSes on the non-secure world, while providing user-level finer-grained partitions on the secure side, used for implementing encapsulated kernel extensions.
2. A real-time, priority and time-partition based scheduler enhanced by a timeslice donation scheme which guarantees processing bandwidth for each partition. This enables the co-existence of non real-time and real-time partitions with low interrupt latencies.
3. Secure IPC mechanisms wrapped by a capability-based access control system and tightly coupled with the real-time scheduler and memory management facilities, enabling fast and efficient partition interactions.
4. Insight on the capabilities and shortcomings of TrustZone-based virtualization, since this design orbited around the alleviation of some of the deficiencies of TrustZone support for virtualization.

The remainder of this paper is structured as follows: Section 2 provides some background information by overviewing Arm TrustZone technology and RTZVisor. Section 3 describes and explains all implementation details behind the development of μRTZVisor: architecture, secure boot, partition and memory manager, capability manager, device and interrupt manager, IPC manager, and scheduler. Section 4 evaluates the hypervisor in terms of memory footprint (TCB size), context-switch overhead, interrupt latency, and IPC overhead. Section 5 points and describes related work in the field. Finally, Section 6 concludes the paper and highlights future research directions.

2. Background

This section starts by detailing the Arm TrustZone security extensions, the cornerstone technology on which μRTZVisor relies. It also describes RTZVisor, the previous version of the hypervisor on which our implementation is built and which persists as the kernel's foundation for essential virtualization mechanisms.

2.1. Arm TrustZone

TrustZone technology is a set of hardware security extensions, which have been available on Arm Cortex-A series processors for several years [41] and has recently been extended to cover the new generation Cortex-M processor family. TrustZone for Armv8-M has the same high-level features as TrustZone for applications processors, but it is different in the sense that the design is optimized for microcontrollers and low-power applications. In the remainder of this section, when describing TrustZone, the focus will be on the specificities of this technology for Cortex-A processors (Figure 1a).

The TrustZone hardware architecture virtualizes a physical core as two virtual cores, providing two completely separated execution environments: the *secure* and the *non-secure* worlds. A new 33rd processor bit, the *Non-Secure* (*NS*) bit, indicates in which world the processor is currently executing. To preserve the processor state during the world switch, TrustZone adds an extra processor mode: the monitor mode. The monitor mode is completely different from other modes because, when the processor runs in this privileged mode, the state is always considered secure, independently of the NS bit state. Software stacks in the two worlds can be bridged via a new privileged instruction-*Secure Monitor Call* (*SMC*). The monitor mode can also be entered by configuring it to handle interrupts and exceptions in the secure side. To ensure a strong isolation between secure and non-secure states, some special registers are banked, while others are either totally unavailable for the non-secure side.

The TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) extend TrustZone security to the memory infrastructure. TZASC can partition the dynamic random-access memory (DRAM) into different secure and non-secure memory regions, by using a programming interface which is only accessible from the secure side. By design, secure world applications can access normal world memory, but the reverse is not possible. TZMA provides similar functionality but for off-chip read-only memory (ROM) or static random-access memory (SRAM). Both the TZASC and TZMA are optional and implementation-specific components on the TrustZone specification. In addition, the granularity of memory regions depends on the system on chip (SoC). The TrustZone-aware memory management unit (MMU) provides two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. The isolation is still available at the cache-level because processor's caches have been extended with an additional tag that signals in which state the processor accessed the memory.

System devices can be dynamically configured as secure or non-secure through the TrustZone Protection Controller (TZPC). The TZPC is also an optional and implementation-specific component on the TrustZone specification. To support the robust management of secure and non-secure interrupts, the Generic Interrupt Controller (GIC) provides both secure and non-secure prioritized interrupt sources. The interrupt controller supports interrupt prioritization, allowing the configuration of secure interrupts with a higher priority than the non-secure interrupts. Such configurability prevents non-secure software from performing a denial-of-service (DOS) attack against the secure side. The GIC also supports several interrupt models, allowing for the configuration of interrupt requests(IRQs) and fast interrupt requests (FIQs) as secure or non-secure interrupt sources. The suggested model by Arm proposes the use of IRQs as non-secure world interrupt sources, and FIQs as secure interrupt sources.

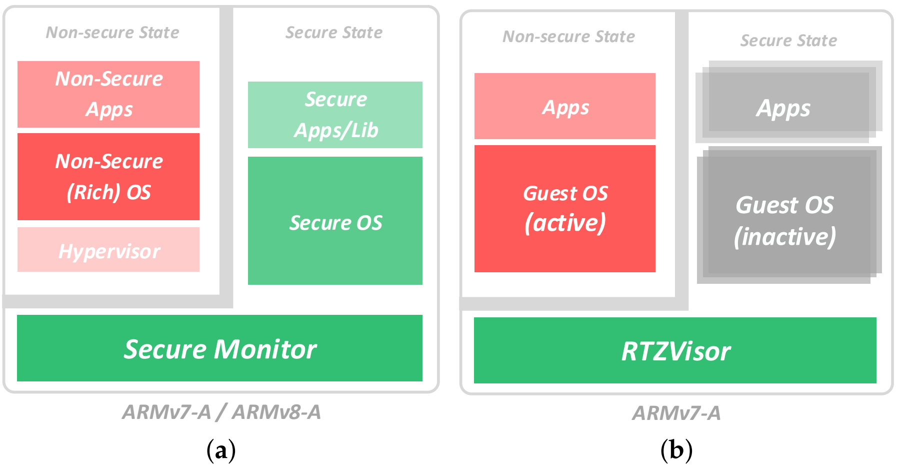

Figure 1. Arm TrustZone: generic architecture and RTZVisor system architecture. (**a**) Arm TrustZone architecture; (**b**) RTZVisor system architecture.

2.2. RTZVisor

RTZVisor [37], the Real-Time TrustZone-assisted Hypervisor, is a bare-metal embedded hypervisor that relies on TrustZone hardware to provide the foundation to implement strong spatial and temporal isolation between multiple guest OSes. RTZVisor is implemented in the C language and follows a monolithic architecture (Figure 1). All hypervisor components, drivers and other critical parts of the virtualization infrastructure run in the most privileged processor mode, i.e., the monitor mode. The hypervisor follows a simple and static implementation approach. All data structures and hardware resources are predefined and configured at design time, avoiding the use of language dynamic features.

Guest OSes are multiplexed on the non-secure world side; this requires careful handling of shared hardware resources, such as processor registers, memory, caches, MMU, devices, and interrupts. Processor registers are preserved in a specific virtual machine control block (VMCB). This virtual processor state includes the core registers for all processor modes, the CP15 registers and some registers of the GIC. RTZVisor offers as many vCPUs as the hardware provides, but only a one-to-one mapping between vCPU, guest and real CPU is supported. RTZVisor only offers the ability to create non-secure guest partitions, and no means of executing software in secure supervisor or user modes.

The strong spatial isolation is ensured through the TZASC, by dynamically changing the security state of the memory segments. Only the guest partition currently running in the non-secure side has its own memory segment configured as non-secure, while the remaining memory is configured as secure. The granularity of the memory segments, which is platform-dependent, limits the number of supported virtual machines (VMs). Moreover, since TrustZone-enabled processors only provide MMU support for single-level address translation, it means that guests have to know the physical memory segment they can use in the system, requiring relocation and consequent recompilation of the guest OS. Temporal isolation is achieved through a cyclic scheduling policy, ensuring that one guest partition cannot use the processor for longer than its defined CPU quantum. The time of each slot can be different for each guest, depending on its criticality classification, and is configured at design time. Time management is achieved by implementing two levels of timing: there are timing units for managing the hypervisor's time, as well as for managing the partitions' time. Whenever the active guest is executing, the timers belonging to the guest are directly managed and updated by the guest on each interrupt. For inactive guests, the hypervisor implements a virtual tickless timekeeping mechanism, which ensures that when a guest is rescheduled, its internal clocks and related data structures are updated with the time elapsed since its previous execution.

RTZVisor's main goal was to prove it was possible to run multiple guest OSes concurrently, completely isolated from each other, on TrustZone-enabled Arm processors without VE support. Despite achieving such a goal, RTZVisor still presented some limitations and open-issues. A list of the main identified limitations follow:

- Hypervisors are not magic bullets and they are also prone to incorrect expectations in terms of security. Guaranteeing a system is secure just by relying on a virtualization layer is not enough. These incorrect expectations probably come from the fact that a hypervisor provides separation and isolation, which positively impacts security. The problem is that security is much more than separation. Security starts from the onset, and hypervisors must be complemented with other security-oriented technologies for guaranteeing a complete chain of trust. The secure boot process is responsible for establishing a chain of trust that authenticates and validates all levels of secure software running on the device. In this sense, the integrity of the hypervisor at boot time is guaranteed.
- RTZVisor does not implement and enforce any existing coding standards. The use of coding standards is becoming imperative in modern security and safety-critical systems to reduce the number of programming errors and achieve certification.
- Although RTZVisor provides real-time support mainly by implementing efficient time services, these are still guest OS dependent and limited to a cyclic scheduling algorithm. The implementation does not allow for event-driven guests to preempt others, resulting in high interrupt latencies.

- The nature of embedded systems requires communication and interaction among the various subsystems. RTZVisor fails in this aspect by not implementing any kind of IPC facilities. All of its guests are completely isolated and encapsulated, having no mechanism to cooperate.
- Finally, and taking into account the previous point, RTZVisor provides no mechanisms for device sharing. Some kind of communication is needed for guests to synchronize, when accessing the same peripheral.

3. μRTZVisor

μRTZVisor is based on a refactoring of RTZVisor, designed to achieve a higher degree of safety and security. In this spirit, we start by anchoring our development process in a set of measures that target security from the onset. First, we made a complete refactoring of the original code from C to C++. The use of an object-oriented language promotes a higher degree of structure, modularity and clarity on the implementation itself, while leveraging separation of concerns and minimizing code entanglement. Kernel modules have bounded responsibilities and only interact through well-defined interfaces, each maintaining its internal state while sharing the control structure of each partition. However, we apply only a subset of C++ suitable to embedded systems, removing features such as multiple inheritance, exception handling or RTTI (Run-Time Type Information) which are error prone, difficult to understand and maintain, as well as unpredictable and inefficient from a memory footprint and execution perspective. In addition to the fact that C++ already provides stronger type checking and linkage, we reinforce its adoption by applying the MISRA (Motor Industry Software Reliability Association) C++ coding guidelines. Due to the various pitfalls of the C++ language, which make it ill-advised for developing critical systems, the main objective of the MISRA C++ guidelines is to define a safer subset of the C++ language suitable for use in safety related embedded systems. These guidelines were enforced by the use of a static code analyzer implementing the standard.

The main feature of μRTZVisor is its microkernel-like architecture, depicted in Figure 2. Nevertheless, we don't strive to implement traditional microkernel virtualization, which, given its para-virtualization nature, imposes heavy guest source-code modification. We aim at gathering those ideas that benefit security and design flexibility, while persevering the capability to run nearly unmodified guest OSes. Since TrustZone-enabled processors already provide two virtual CPUs, by providing a secure and non-secure view of the processor, and extends this partitioning to other resources such interrupts and devices, guests can make full use of all originally intended privileged levels, being allowed to directly configure assigned system resources, manage their own page tables and directly receive their assigned interrupts. However, the lack of a two-level address translation mechanism imposes a segmented memory model for VMs. Hence, guest OSes need to be compiled and cooperate to execute in the confinement of their assigned segments. This issue is augmented by the fact that segments provided by the TZASC are typically large (in the order of several MB) and must be consecutively allocated to guests, which leads to high levels of internal fragmentation. In addition, the maximum possible number of concurrent guests is limited by the total number of available TZASC segments, which varies according to the platform's specific implementation. Nevertheless, however small the number of segments, this segmented memory model is likely to suffice for embedded use-cases that usually require a small, fixed number of VMs according to deployed functionalities.

Besides this, guest OSes only need to be modified if they are required to use auxiliary services or shared resources that rely on the kernel's IPC facilities. For this, typically used commodity operating systems, such as Linux, may simply add kernel module drivers to expose these mechanisms to user applications. Multi-guest support is achieved by multiplexing them on the non-secure side of the processor, i.e., by dynamically configuring memory segments, devices or interrupts of the active partition as non-secure, while inactive partition resources are set as secure and by saving and restoring the context of CPU, co-processor and system control registers, which are banked between the two worlds. An active guest that attempts to access secure resources triggers an abort exception directly to the hypervisor. However, these exceptions may be imprecise as in the case of those triggered by an unallowed device access. Additionally, the existence of banked registers or their silently failing access, as is the case of secure interrupts in the GIC, coupled with the fact that the execution of privileged instructions cannot be detected by the monitor, makes it such that classical techniques such as trap-and-emulate cannot be used to further enhance a full-virtualization environment. At the moment, when one of the aforementioned exceptions is triggered, guests are halted or blindly rebooted. In addition, given that guests share cache and TLB (Translation Lookaside Buffer) infrastructures, these must be flushed when a new guest becomes active. Otherwise, the entries of the previous guest, which are marked as non-secure, could be accessed without restriction by the incoming one.

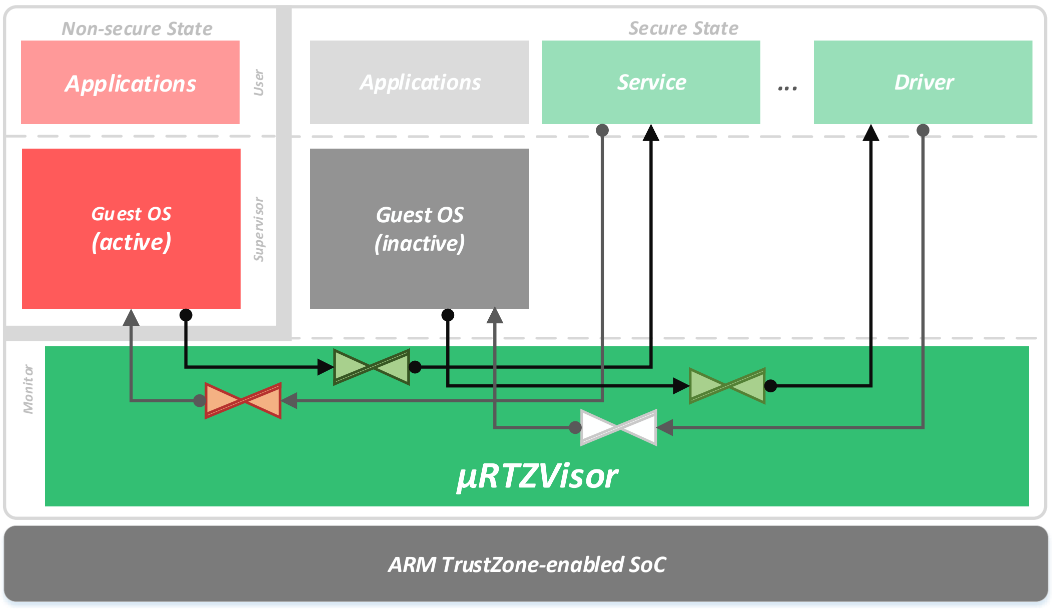

Figure 2. μRTZVisor architectural overview.

μRTZVisor privilege code runs in monitor mode, the most privileged level in TrustZone-enabled processors, having complete access and configuration rights over all system resources. This layer strives to be a minimal TCB, implementing only essential infrastructure to provide the virtual machine abstraction, spatial and temporal partitioning, and basic services such as controlled communication channels. The kernel's design aims for generality and flexibility so that new functionality can be added in a secure manner. For this, it provides a heterogeneous partition environment. As described above, coarse-grained partitions based on the memory segmentation model are used to run guest OSes. In addition, partitions running in secure user mode are implemented by managing page tables used by the MMU's secure interface, which allows for a greater degree of control over their address spaces. Secure user mode partitions are used to implement extra functionality, which would typically execute in kernel mode in a monolithic system. They act as server tasks that can be accessed through RPC operations sitting on the IPC and scheduling infrastructure. For example, shared device drivers or virtual network infrastructures can be encapsulated in these partitions. Herein lies the main inspiration from microkernel ideas. Non-essential services are encapsulated in these partitions, preventing fault-propagation to other components. Hence, they can be untrusted and developed by third-parties, incorporating only the TCB of other partitions that depend on them. Although these kind of services could be implemented in VMs running in the non-secure world, rendering worthless the extra core complexity added to the kernel, implementing them as secure world tasks provides several benefits. First, running them on a secure virtual address space eliminates the need for the relocation and recompilation and reduces the fragmentation inherent to the segmented memory model. This facilitates service addition, removal or swapping according to guests' needs and overall system requirements. At the same time, it enables finer-grained functionality fault-encapsulation. Finally, both the hypervisor and secure tasks always run with caches enabled, but, since caches are TrustZone-aware, there is no need to flush them when switching from a guest partition to a secure world task due to a service request via RPC, which significantly improves performance.

A crucial design decision relates to the fact that partitions are allocated statically, at compile-time. Given the static nature of typical embedded systems, there is no need for partitions to create other partitions or to possess parent-child relations and some kind of control over one another. This greatly simplifies the implementation of partition management, communication channel and resource distribution, which are defined and fixed according to the system design and configuration. This idea is further advanced in the next few paragraphs.

To achieve robust security, fault-encapsulation is not enough and the principle of the least authority must be thoroughly enforced. This is done at a first level by employing the aforementioned hardware mechanisms provided both by typical hardware infrastructure (virtual address translation or multiple privilege levels) and the TrustZone features that allow control over memory segments, devices and interrupts. Those are complemented by a capability-based access control mechanism. A capability represents a kernel object or a hardware resource and a set of rights over it. For example, a partition holding a capability for a peripheral can access it according to the set read/write permissions. This is essential so that secure tasks can configure and interact with their assigned hardware resources in an indirect manner, by issuing hypercalls to the kernel. Guest OSes may own capabilities but do not necessarily have to use them, unless they represent abstract objects such as communications endpoints, or the guest needs to cooperate with the kernel regarding some particular resource it owns. In this way, all the interactions with the kernel, i.e., hypercalls, become an invocation of an object operation through a capability. This makes the referencing of a resource by partitions conceptually impossible if they do not own a capability for it. Given that the use of capabilities provides fine-grained control over resource distribution, system configuration almost fully reduces to capability assignment, which shows to be a simple, yet flexible mechanism.

Built upon the capability system, this architecture provides a set of versatile inter-partition communication primitives, the crucial aspect of the microkernel philosophy. These are based on the notion of a port, constituting an endpoint to and from which partitions read and write messages. Given that these operations are performed using port capabilities, this enables system designers to accurately specify the existing communication channels. In addition, the notion of reply capabilities, i.e., port capabilities with only the send rights set, which can only be used once, and that are dynamically assigned between partitions through IPC, is leveraged to securely perform client-server type communications, since they remove the need to grant servers full-time access to client ports. Port operations can be characterized as synchronous or asynchronous, which trade-off security and performance. Asynchronous communication is safer since it doesn’t require a partition to block, but entails some performance burden. In opposition, synchronous communication is more dangerous, since partitions may block indefinitely, but allows partitions to communicate faster, by integration with scheduler functionalities for efficient RPC communication. Aiming at providing the maximum design flexibility, our architecture provides both kinds of communication.

This architecture categorically differs from classical TrustZone software architectures, which typically feature a small OS running in secure supervisor mode that manages secure tasks providing services to non-secure partitions and that only execute when triggered by non-secure requests or interrupts. This service provision by means of RPC overlaps with our approach, but it focuses only on providing a Trusted-Execution Environment (TEE) and not on a flexible virtualized real-time environment. No such OS exists following this approach, since the hypervisor directly manages these tasks, leaving the secure supervisor mode vacant. This partial flattening of the scheduling infrastructure allows for the direct switch between guest client and server partitions, reducing overhead, and for secure tasks to be scheduled in their own right to perform background computations. At the same time, given that the same API is provided to both client and server partitions, it homogenizes the semantics of communication primitives and enables simple applications that show no need for a complete OS stack or large memory requirements to execute directly as secure tasks. In addition, in some microkernel-based virtualization implementations, the VM abstraction is provided by user-level components [31], which, in our system, would be equivalent to the secure tasks. This encompasses high-levels of IPC traffic between the VM and the guest OS and a higher number of context-switches. Given the lightweight nature of the VM provided by our system, this abstraction directly provided at the kernel level, which, despite slightly increasing TCB complexity, significantly reduces such overhead.

Besides security, the architecture places strong emphasis on the real-time guarantees provided by the hypervisor. Inspired by ideas proposed in [8], the real-time scheduler structure is based on the notion of time domains that execute in a round-robin fashion and to which partitions are statically assigned. This model guarantees an execution budget for each domain which is replenished after a complete cycle of all domains. Independently of their domain, higher priority partitions may preempt the currently executing one, so that event-driven partitions can handle events such as interrupts as quickly as possible. However, the budget allocated to these partitions must be chosen with care according to the frequency of the events, to not be exhausted, delaying the handling of the event until the next cycle. We enhance this algorithm with a time-slice donation scheme [42] in which a client partition may explicitly donate its domain's bandwidth to the target server until it responds, following an RPC pattern. In doing so, we allow for the co-existence of non-real time and real-time partitions, both time and event-driven, while providing fast and efficient communication interactions between them. All related parameters such as the number of domains, their budgets, partition allocation and their priorities are assigned at design time, providing once again a highly flexible configuration mechanism.

For the kernel's internal structure, we opted for a non-preemptable, event-driven execution model. This means that we use a single kernel stack across execution contexts, which completely unwinds when leaving the kernel, and that, when inside the kernel, interrupts are always disabled. Although this design may increase interrupt and preemption latencies, which affect determinism by increasing jitter, the additional needed complexity to make the kernel fully preemptable or support preemption points or continuations would significantly increase the system's TCB. Moreover, although the great majority of hypercalls and context switch operations show a short constant execution time, others display a linear time complexity according to hypercall arguments and the current state of time-slice donation, which may aggravate the aforementioned issues. We also maintain memory mappings enabled during hypercall servicing. This precludes the need to perform manual address translation on service call parameters located in partition address spaces, but it requires further cache maintenance to guarantee memory coherency in the eventuality that guests are running with caches disabled.

Finally, it is worth mentioning that the design and implementation of the μRTZVisor was tailored for a Zynq-7000 SoC and is heavily dependent on the implementation of TrustZone features on this platform. Although the Zynq provides a dual-core Arm Cortex-A9, the hypervisor only supports a single-core configuration. Support for other TrustZone-enabled platforms and multicore configurations will be explored in the near future.

3.1. Secure Boot

Apart from the system software architecture, security starts by ensuring a complete secure boot process. The secure boot process is made of a set of steps that are responsible for establishing a complete chain of trust. This set of steps validates, at several stages, the authenticity and integrity of the software that is supposed to run on the device. For guaranteeing a complete chain of trust, hardware trust anchors must exist. Regarding our current target platform, a number of secure storage facilities were identified, which include volatile and non-volatile memories. Off-chip memories should only be used to store secure encrypted images (boot time), or non-trusted components such as guest partitions (runtime).

The complete secure boot sequence is summarized and depicted in Figure 3. After the power-on and reset sequences have been completed, the code on an on-chip ROM begins to execute. This ROM is the only component that cannot be modified, updated or even replaced by simple reprogramming attacks, acting as the root of trust of the system. It starts the whole security chain by ensuring authentication and decryption of the first-stage bootloader (FSBL) image. The decrypted FSBL is then loaded into a secure on-chip memory (OCM). Once the FSBL starts to run, it is then responsible for the authentication, decryption and loading of the complete system image. This image contains the critical code of the μRTZVisor and secure tasks, as well as the guest OSes images. The binary images of the guest OSes are individually compiled for the specific memory segments they should run from, and then attached to the final system image through the use of specific assembly directives. Initially, they are positioned in consecutive (secure) memory addresses, and, later, the hypervisor is the one responsible for copying the individual guest images to the assigned memory segment they should run from. Therefore, at this stage, the system image is attested as a whole, and not individually. In the meantime, if any of the steps on the authentication and decryption of the FSBL or the system image fails, the CPU is set into a secure lockdown state.

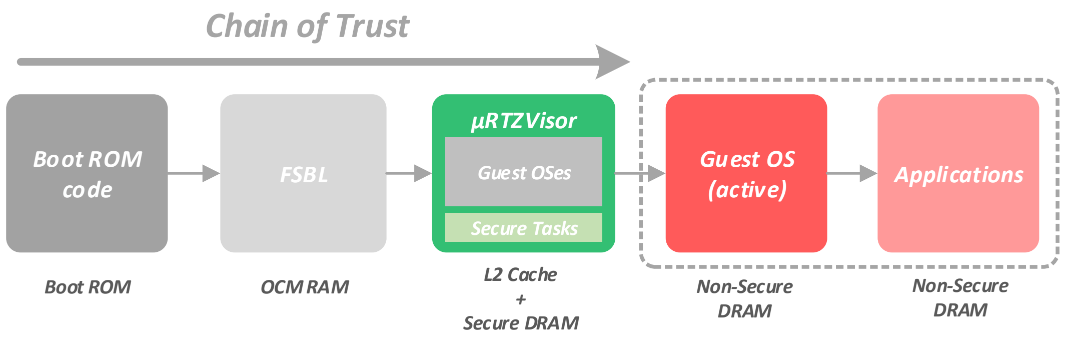

Figure 3. μRTZVisor: secure boot process.

Once the system image has been successfully loaded and authenticated, control is turned over to μRTZVisor, which resides in OCM. Upon initialization, the μRTZVisor will load and lock its complete code to the L2 cache as well as part of its critical data structures. The remaining OCM is left to be used as a shared memory area by partitions as detailed in Section 3.4. The μRTZVisor is then responsible for configuring specific hardware for the hypervisor, and for loading the guest OS images to the corresponding memory segment. Guest OSes images are not individually encrypted. As aforementioned, they are part of the overall system image. μRTZVisor does not check the integrity of the non-secure guest OSes binaries when they are loaded. This means that the chain of trust ends when the critical software is securely running. It is assumed that everything that goes outside the perimeter of the hypervisor's kernel side can be compromised. Nevertheless, the addition of another stage of verification, at the partition level, would help to achieve a supplementary level of runtime security for the entire system lifetime. By including an attestation service as a secure task, it would be possible to check and attest partition identity and integrity at boot time, as well as other key components during runtime.

3.2. Partition Manager

The main role of the partition manager is to guarantee consistency and integrity of the partitions execution context, namely their CPU state. This module also encapsulates the list of partition control blocks (PCBs), which encapsulate the state of each partition and which partition is currently active. Other kernel modules must use the partition manager interfaces to access the currently active partition and entries of the PCB for which they are responsible.

As previously explained, two types of partitions are provided: non-secure guest partitions and secure task partitions. While, for task partitions, state is limited to user mode CPU registers, for guest partitions, the state encompasses banked registers for all execution modes, non-secure world banked system registers, and co-processor state (currently, only the system configuration co-processor, CP15, is supported). The provided VM abstraction for guest OSes is complemented by the virtualization structures of the GIC's CPU interface and distributor as well as of a virtual timer. These are detailed further ahead in Sections 3.5 and 3.8.

The partition manager also acts as the dispatcher of the system. When the scheduler decides on a different partition to execute, it informs the partition manager which is responsible for performing the context-switch operation right before leaving the kernel. In addition to saving and restoring context related to the aforementioned processor state, it coordinates the context-switching process among the different core modules, by explicitly invoking their methods. These methods save and restore partition state that they supervise, such as a memory manager method to switch between address spaces.

The partition manager also implements the delivery of asynchronous notifications to task partitions, analogous to Unix-style signals. This is done by saving register state on the task's stack and manipulating the program counter and link registers to jump to a pre-agreed point in the partition's executable memory. The link register is set to a pre-defined, read-only piece of code in the task's address space that restores its register state and jumps to the preempted instruction. The IPC manager uses this mechanism to implement the event gate abstraction (Section 3.6).

3.3. Capability Manager

The Capability Manager is responsible for mediating partitions access to system resources. It implements a capability-based access control system that enables fine-grained and flexible supervising of resources. Partitions may own capabilities, which represent a single object, that directly map to an abstract kernel concept or a hardware resource. To serve its purpose, a capability is a data structure that aggregates owner identification, object reference and permissions. The permissions field identifies a set of operations that the owning partition is allowed to perform on the referenced object. In this architecture, every hypercall is an operation over a kernel object; thus, whenever invoking the kernel, a partition must always provide the corresponding capability.

Figure 4 depicts, from a high-level perspective, the overall capability-based access control system. A partition must not be able to interact with objects for which it does not possess a capability. In addition, these must neither be forgeable or adulterable, and, thus, partitions do not have direct access to their own capabilities, which are stored in kernel space. Each partition has an associated virtual capability space, i.e., an array of capabilities through which those are accessed. Whenever performing a hypercall, the partition must provide the identifier for the target capability in the capability space, which is then translated to a capability on a global and internal capability pool. This makes it conceptually impossible for a partition to directly modify their capabilities or operate on objects for which it does not possess a capability, as only the capabilities on its capability-space are indirectly accessible. In addition, for every hypercall operation, the partition must specify the operation it wants to perform along with additional operation-specific parameters. At the kernel's hypercall entry point, the capability is checked to ensure the permission for the intended operation is set. If so, the capability manager will redirect the request to the module that implements the referenced object (e.g., for an IPC port it will redirect to IPC manager), which will then identify the operation and perform it.

At system initialization, capabilities are created and distributed according to a design-time configuration. Some capabilities are fixed, meaning that they are always in the same position of the capability space of all partitions, despite having configurable permissions. These capabilities refer to objects such as the address space, which are always created for each partition at a system's initialization. For capabilities associated with additional objects defined in the system's configuration, a name must be provided such that it is unique in a given partition's capability space. During execution, partitions can use this name to fetch the associated index in their capability space.

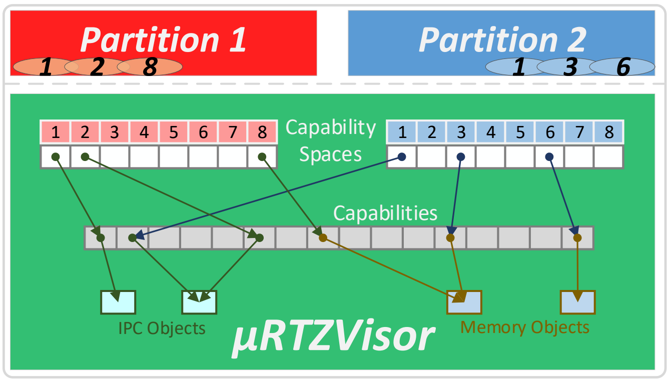

Figure 4. Capability-based access control system overview.

*µ*RTZVisor also provides mechanisms to dynamically propagate access rights by invoking `Grant` and `Revoke` operations on capabilities, which, as for any other operation, must have the respective rights set in the permissions field. The `Grant` operation consists of creating a derived capability, which is a copy of the original one, with only a subset of its permissions, and assigning it to another partition. The operation's recipient is notified about it, through an IPC message in one of its ports, which contains the index for the new capability in its capability space. To perform the `Grant` operation, the granter must specify the granting capability, the subset of permissions to grant, which must be enclosed in the original ones, and the port to which the notification will be delivered. This means that the granter must possess a capability for a port owned by the recipient. A derived capability may further be granted, giving rise to possible complex grant chains. Each derived capability is marked with a grant id, which may later be used to perform the revoke operation. In turn, the revoke operation withdraws a given capability from its owner, and can only be performed by one of the partitions in a preceding grant chain. The revocation process propagates through the donation chain. A revoked capability is maintained in a zombie state in the capability space, until it is used again. When the owning partition tries to use it, it will receive an error and the position will be freed so that it can be used again. Finally, there is a special type of capability, called a one-time capability, that can only be used once. The first time a partition uses this capability it is erased from the partition's capability space. These are also referred to as reply capabilities in the context of the IPC manager, and are leveraged to perform secure RPC communication. This is further detailed in Section 3.6.

3.4. Memory Manager

At system initialization, the memory manager starts by building the address spaces for all partitions from meta-data detailing the system image layout. For guest partitions, this encompasses figuring out which consecutive memory segments must be set as non-secure when the guest is active, i.e., those for which they were compiled to run on, and loading them to these segments. On Zynq-based devices, TrustZone memory segments have a granularity of 64 MB, which might lead to high levels of internal fragmentation. For example, for a 1 MB or 65 MB guest OS binary, 63 MB of memory is left unused. During this process, if it is detected that two guests were built to run by sharing the same memory segment, the manager will halt the system, since the spatial isolation requirement cannot be guaranteed. From the remaining free memory, the memory manager will build the virtual address spaces for secure tasks, which currently only use 1 MB pages instead of the available 4 K pages. In doing so, we simplify the manager's implementation while keeping the page table footprint low, since the latter would require second level page tables. In addition, in the current implementation, no more memory can be allocated by tasks after initialization, so partition binaries must contemplate, at compile-time, memory areas to be used as stack and heap, according to the expected system needs.

The hypervisor code and data are placed in the bottom 64 MB memory segment, which is always set as secure. Address spaces always contemplate this area as kernel memory and may extend until the

1 GB limit. Since we manage tasks' page tables, their virtual address space always starts immediately after kernel memory, making the recompilation needed for guest partitions, which always have a confined, but direct view of physical memory unnecessary. Above the 1 GB limit, the address space is fixed for all partitions, contemplating a peripheral, CPU private and system register area, and, at the top, an OCM region of TZASC 4KB segments, which we call slices and use for guest shared memory as detailed below.

This module manages two page tables used by the secure interface of the MMU. The first is a 1-to-1 mapping to physical memory and is used when a guest partition is currently active. The second is used when a task partition is active and is updated each time a new task is scheduled. Since it is expected that secure service partitions do not need to occupy a large number of pages, only the individual page table entries are saved in a list structure. The extra-overhead of updating the table at each task context restore was preferred to keeping a large and variable number of page tables and only switching the page table pointer, reducing the amount of memory used by the hypervisor. Despite the transparent view of the top physical address space, this is controlled by managing a set of three extra page tables that map the 4 KB peripheral and the TZPC and TZASC partitioning infrastructure that enables control over guest access to peripherals and shared slices. This is done in cooperation with the device manager module described in Section 3.5.

Partition memory mappings are kept active while the kernel executes, in order to avoid manual address translation for the kernel to access partition space when reading information pointed to by hypercall arguments. At the same time, the memory manager offers services to other system modules that enable them to verify the validity of the hypercall arguments (i.e., if data pointed by these arguments is indeed part of the partitions address space), read and write to and from address spaces others than the currently active one and perform address translations when needed.

Upon address space creation, a capability is inserted in the partitions' capability spaces that enables them to perform operations over it. At the moment, these operations are exclusively related to the creation and mapping of objects representing some portion of physical memory and that support shared memory mechanisms. We distinguish two different types of memory objects, page and slice objects, always represented and manipulated through capabilities and shared among partitions using grant and revoke mechanisms. Although both kinds of objects may be created by guest and task partitions, only slice objects may be mapped by guests, since guest address space control is exclusively performed through TrustZone segmentation mechanisms. For example, a guest that needs a task to process its data may create a page object representing the pages containing that data using its address space capability. It then grants the page object capability to the task. The task uses this capability to map the memory region to its own address space and processes the data. When the task is done, it signals the client partition, which revokes the capability for the page object, automatically unmapping it from the task's address space. The same can be done among tasks, and among guests, although, in the latter case, only using the slice memory region. Using this sharing mechanism, a performance boost is obtained for data processing service provision among partitions.

TrustZone-aware platforms extend secure and non-secure memory isolation to both the cache and memory translation infrastructure. Nevertheless, a cache or TLB marked as non-secure may be accessed by non-secure software, despite the current state of memory segment configuration. Hence there is a need to flush all non-secure cache lines when a new guest partition becomes active, so that they cannot access each other cached data. This is also performed for non-secure TLB entries since a translation performed by a different guest might be wrongly assumed by the MMU. This operation is not needed when switching secure tasks since TLB entries are tagged. Although it is impossible for non-secure software to access secure cache entries, the contrary is possible by marking secure page table entries as non-secure, which enables the kernel and secure tasks to access non-secure cache lines when reading hypercall arguments or using shared memory. This, however, puts forth coherency and isolation issues, which demand maintenance that negatively impacts performance. First, it becomes imperative for guest partition space to be accessed only through a memory manager's services so that it can keep track of possible guest lines pulled to the cache. When giving control back to the guest, these lines must be flushed to keep coherency in case the guest is running with caches disabled. This mechanism may be bypassed if, at configuration time, the designer can guarantee that all guest arguments are cached. Moreover, if a guest shares memory with a task, further maintenance is required, in order to guarantee that a guest with no access permission for the shared region cannot find it in the cache. This depends on the interleaved schedule of task and guests as well as the current distribution of shared memory objects.

3.5. Device Manager

The job of the device manager is similar to that of the memory manager since all peripherals are memory mapped. It encompasses managing a set of page tables and configuring TZPC registers to enable peripheral access to task and guest partitions, respectively. Each peripheral compromises a 4 KB aligned memory segment, which enables mapping and unmapping of peripherals for tasks, since this is the finest-grained page size allowed by the MMU. The peripheral page tables have a transparent and fixed mapping, but, by default, bar access to user mode. When a peripheral is assigned to a task, the entry for that device is altered to allow user mode access at each context switch. For all non attributed devices, the reverse operation is performed. An analogous operation is carried out for guests, but by setting and clearing the peripheral's secure configuration bit in a TrustZone register. If a peripheral is assigned to a partition, it can be accessed directly, in a pass-through manner, without any intervention of the hypervisor.

At initialization, the device manager also distributes device capabilities for each assigned device according to system configuration. Here, when inserting the capability in a partition's capability space, the manager automatically maps the peripheral for that partition, allowing the partition to directly interact with the peripheral without ever using the capability. However, if partitions need to share devices by granting their capabilities, the recipient must invoke a map operation on the capability before accessing the device. The original owner of the capability may later revoke it. This mechanism is analogous to the shared memory mechanism implemented by the memory manager.

There may be the need for certain devices to be shared across all guest partitions, if they are part of the provided virtual machine abstraction. These virtual devices are mapped in all guest partition spaces, being no longer considered in the aforementioned sharing and capability allocation mechanisms. A kernel module may inform the device manager that a certain device is virtual at system initialization. From there on, that module will be responsible for maintaining the state of the device by installing a callback that will be invoked at each context-switch. This is the case for the timer provided in the virtual machine, TTC1 (Triple Timer Counter 1) in the Zynq, destined to be used as a tick timer for guest OSes. We added a kernel module that maintains the illusion that the timer belongs only to the guest running on the VM by effectively freezing time when the guest is inactive. The same is true for interrupts associated with these virtual devices, and so the interrupt manager provides a similar mechanism for classifying interrupts as virtual instead of assigning them to a specific guest.

Finally, it is worth mentioning that devices themselves can be configured as secure or non-secure, which will define if they are allowed to access secure memory regions when using Direct Memory Access (DMA) facilities. We have not yet studied the intricacies of the possible interactions between secure and non-secure partitions in such scenarios, so we limit devices to make non-secure memory accesses.

3.6. IPC Manager

Communication is central to a microkernel architecture. In our approach, it is based on the notion of ports, which are kernel objects that act as endpoints through which information is read from and sent to in the form of messages. Communication mechanisms are built around the capability-system, in order to promote a secure design and enforce the principle of least authority. Thus, as explained in Section 3.3, in order to perform an IPC operation over a port, a partition must own a capability referencing that same port, with the permissions for the needed operations set. For example, a given task may own a capability for the serial port peripheral. Whenever other partitions want to interact with the device, they would need to send a message to the owning task that would interpret that message as a request, act and reply accordingly. In this scenario, a port must exist for each partition. For each port, they should possess capabilities with the minimum read and write permissions set that ensure a correct communication progression.

Port operations may work in a synchronous or asynchronous style, and are further classified as blocking or non-blocking. Synchronous communication requires that at least one partition blocks waiting for another partition to perform the complementary operation, while, in asynchronous communication, both partitions perform non-blocking operations. Synchronous communication does not require message buffering inside the kernel, improving performance since it is achieved by copying data only once, directly between partition address spaces. On the other hand, asynchronous communication requires a double data copy: first from the sender's address space to the kernel, and then from the kernel to the recipient's address space. Although this provokes performance degradation, it enforces the system's security by avoiding the asymmetric trust problem [43,44], where an untrustworthy partition may cause a server to be blocked indefinitely, preventing it from answering other partitions' requests, resulting in possible DOS attacks. This could be solved by the use of timeouts; however, there is no theory to determine reasonable timeout values in non-trivial systems [28]. In asynchronous communication, since no partition blocks waiting for another one, there is no risk of that happening. These trade-offs between performance and security must be taken into account when designing this kind of system. Despite our focus on security, we also want to offer the flexibility provided by synchronous communication, which enables fast RPC semantics for efficient service provision. As such, the μRTZVisor provides port operations for both scenarios, combining synchronous operations with the scheduling infrastructure, explained in Section 3.7.

The most elemental port operations are `Send` and `Receive`, where the former is never blocking, but the latter may be. Other operations are composed as a sequence of these elemental operations, in addition to other services provided by the scheduler, for time-slice donation, or by the capability manager for one-time capability propagation.

Table 1 summarizes all IPC primitives over available ports. As shown in the table, there are two kinds of receive operations—one blocking and the other non-blocking. In the first case, the partition will stall and wait for a complementary send operation to happen on the respective port to resume its execution. The second one will check for messages in the port's message buffer, and return one in first-in, first-out (FIFO) arrival order if available, or an error value if the port is empty.

Table 1. Port operations characterization, i.e., if it is synchronous or asynchronous and either blocking or non-blocking.

Port Operations	Synchronous	Asynchronous	Blocking	Non-Blocking
Send	x	x	-	x
RecvUnblock	-	x	-	x
RecvBlock	x	-	x	-
SendReceive	x/-	x/x	-	x
SendReceiveDonate	x/x	x/-	x	-
ReceiveDonate	x	-	x	-

If the `Send` operation follows a `Receive` that is blocking, it will happen in a synchronous style, i.e., that copy will be sent directly to the recipient's address space. Otherwise, the communication will be asynchronous, which means that a message will be pushed into the port's message buffer.

Both `SendReceive` operations will perform an elemental send followed by an elemental receive. In addition, they rely on services from the capability manager to grant a capability to the recipient partition. The capability may be granted permanently, like in a grant operation performed by the capability manager, or may be erased after being used once, dubbed reply-capabilities. When performing an operation with the -Donate suffix, the partition is donating its execution time-slice to the recipient partition, and it blocks its execution until receiving a response message from that same partition. More details about the donation process will be given in Section 3.7.

When a given partition asynchronously sends a message, the recipient partition may receive an event to notify it about the message arrival. Events are asynchronous notifications that alter the partition's execution flow. For the secure tasks' partitions, they are analogous to Unix signals and are implemented by the partition manager described in Section 3.2. For guest partitions, they resemble a normal interrupt, to not break the illusion provided by the virtual machine. In addition, it would be extremely difficult to implement them as signals, given that the hypervisor is OS-agnostic and has no detailed knowledge about the internals of the guest. Hence, services from the interrupt manager (Section 3.8) are used to inject a virtual interrupt in the virtual machine. To receive events, guests must configure and enable the specified interrupt in their virtual Generic Interrupt Controller (GIC). In this way, events are delivered in a model closer to the VM abstraction, and OS agnosticism is maintained. Partitions interact with the event infrastructure through a kernel object called event gate. To receive events, partitions must configure the event gate and associate it with ports that will trigger an event upon message arrival. To lower implementation complexity, each partition is assigned a single event gate, which will handle events for all ports in a queued fashion. In addition, a capability with static permissions is assigned to each partition for its event gate at system's initialization. The aforementioned permissions encompass only the `Configure` and `Finish` operations. The configure operation allows partitions to enable events, and also to specify the memory address of a data structure where event-related data will be written to upon event delivery, and that should be read by the owning partition to contextualize the event.

For synchronization purposes, there are also specific kernel objects called locks, whose operations may be blocking or unblocking. A partition may try to acquire a synchronization object by performing one of two versions of a lock: `LockBlocking` and `LockUnblocking`. The first changes partition state to blocked in case the object has already been acquired by other partition, scheduling the background domain. The latter will return the execution to the invoker, thus working like a spin-lock. To release the object, there is an `Unlock` operation. The existence of this kind of object will allow, for example, partitions to safely share a memory area for communication purposes. To do this, all of them must possess capabilities referencing the same lock object.

All communication objects must be specified at design time, which means that partitions are not allowed to dynamically create ports. In addition, the respective capabilities should be carefully distributed and configured, since, depending on its permissions, it may be possible for a partition to dynamically create new communication relations through the capability `Grant` operation, or by `SendReceive` operations. Therefore, all possible existing communication channels are, at least implicitly, defined at configuration time. Although partitions may grant port capabilities, if no relation for communication exists between partitions, they will never be able to spread permissions. Thus, designers must take care to not unknowingly create implicit channels, since isolation is reinforced by the impossibility of partitions to communicate with each other, when they are not intended to.

The port abstraction hides the identity of partitions in the communication process. They only read and write messages to and from the endpoints but do not know about the source or the recipient of those messages. This approach is safer since it hides information about the overall system structure that might be explored by malicious partitions. However, ports can be configured as privileged and messages read from these ports will contain the ID of the source partition. This enables servers to implement connection-oriented services, which might encompass several interactions, in order to distinguish amongst their clients and also to associate their internal objects with each one.

3.7. Scheduler

The presented approach for the μRTZVisor scheduler merges the ideas of [42,45]. It provides a scheduling mechanism that enables fast interaction between partitions, while enabling the coexistence of real-time and non-real-time applications without jeopardizing temporal isolation, by providing strong bandwidth guarantees.

The scheduler architecture is based on the notion of a time domain, which is an execution window with a constant and guaranteed bandwidth. Time domains are scheduled in a round-robin fashion. At design time, each time domain is assigned an execution budget and a single partition. The sum of all execution budgets constitutes an execution cycle. A partition executes in a time domain, consuming its budget until it is completely depleted, and the next time domain is then scheduled. Whenever a complete execution cycle ends, all time budgets are restored to their initial value. This guarantees that all partitions run for a specified amount of time in every execution cycle, providing a safe execution environment for time-driven real-time partitions.

The scheduler allows that multiple partitions may be assigned to a special-purpose time domain, called domain-0. Inside domain-0, partitions are scheduled in a priority-based, time-sliced manner. Furthermore, domain-0's partitions may preempt those running in different domains. It is necessary to mention that any partition is assigned a priority, which only has significance within the context of domain-0. At every scheduling point, the priorities of the currently active time domain's partition and the domain-0's highest priority ready partition are compared. If the latter possesses a higher priority, it preempts the former, but consuming its own domain's (i.e., domain-0's) time budget while executing. The preemption does not happen, of course, if domain-0 itself is the currently active domain. Figure 5 presents an example scenario containing two domains: time domain 1, assigned with partition X; and time domain 2, assigned with Partition Y, in addition to domain-0, assigned with partitions Z and W. 0 Time domain 1 is first in line, and since partition X has higher priority than the ones in domain-0, it will start to execute. After its time budget expires, a scheduling point is triggered. Time-domain 2 is next, but since domain-0's partition Z possesses higher priority than partition Y from time domain 2, Z is scheduled consuming domain-0's budget. At a certain point, partition Z blocks, and since no active partition in domain-0 has more priority than domain 2's Y, the latter is finally scheduled and executes for its time domain's budget. The next scheduling point makes domain-0 the active domain, and the only active partition, W, executes, depleting domain-0's budget. When this expires, a new execution cycle begins and domain 1's partition X is rescheduled.

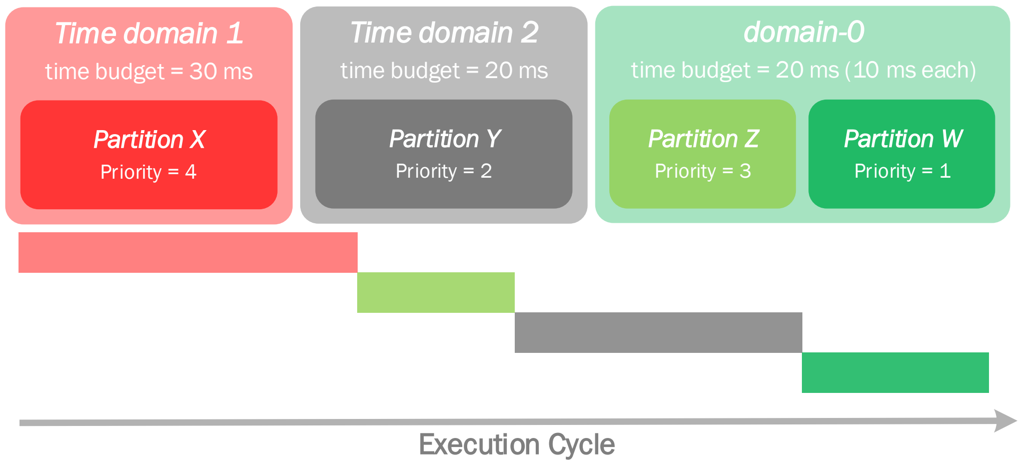

Figure 5. Example of an execution cycle, given a set of time domains with their own partitions and respective configuration.

Aiming at providing fast interaction between partitions, some IPC operations are tightly coupled with specific scheduling functionalities. Section 3.6 highlights a number of IPC operations that rely on the scheduler: the `ReceiveBlocking` , and the ones with the -donate suffix (i.e., `SendReceiveDonate` and `ReceiveDonate`). The `ReceiveBlocking` operation results in changing the partitions state to blocked, and then scheduling the next ready partition from domain-0 to perform some background work. Nevertheless, it keeps consuming the former domain's time budget, since it prevails as the active time domain. Hence, by blocking, a partition implicitly donates its execution time to domain-0. The following scheduling point will be triggered according to one of three scenarios: (a) domain-0's internal time slice expires, which results in scheduling the next highest priority partition from domain-0; (b) the active time domain's budget expires, and the next time domain becomes active; (c) the executing partition sends a message to the active time domain's partition, which would change its state to ready and result in scheduling it right away. In summary, upon blocking, a partition remains in this state until it is unblocked by receiving a message on the port it is hanging. If an execution cycle is completed without a change in the partition's state, partitions from domain-0 are scheduled once more in its place.

The -donate suffixed operations require a more intricate operation from the scheduler, and by invoking them, a partition is explicitly donating its time budget to the recipient port's owner. Hence, it will block until it has the created dependency resolved, i.e., it blocks waiting for the message's recipient to send its response. In case the donator has a higher priority than the donatee server, the latter will inherit the former's priority, augmenting the chances of it to execute and to resolve the dependency sooner. Considering a scenario where two partitions donated their time to low-priority servers running in domain-0, the server that inherits the higher priority will execute first when domain-0's becomes active, or even preempt another time domain's partition, which it previously wouldn't preempt. This enables services to be provided in a priority-based manner, i.e., maintaining the priority semantics of the requesting partitions. This priority inheritance mechanism also mitigates possible priority inversion scenarios. A partition relying on another one, and donating its time domain without any other intervener, constitutes the simplest form of a donation chain. However, a donate operation may be performed to or from a partition that is already part of a donation chain in a transitive manner, constituting a more intricate scenario. Whatever partition is at the tail of the chain, it will be the one to execute whenever one of the preceding partitions is picked by the scheduler. Nonetheless, only the one following a given partition at the donation chain is able to restore its state to ready, by sending a message to the port on which the donator is waiting for the response to its request.

This synchronous, donation-based mechanism is prone to deadlock scenarios, which in our approach is synonymous with a cycle in the donation chain. Instead of traversing the chain and looking for cycles for every donation, this problem is mitigated recurring to a bitmap for a more lightweight solution. Given that the number of partitions is fixed, each bit in the bitmap represents a partition in the system, and that, if set, means that the represented partition is in the donation chain between the respective node and the chain's tail. Every partition has at least its own bit set in its bitmap. Thus, cycles are detected whenever a match occurs by crossing both bitmaps, from the donator and the recipient for the donation.

Although not imposed by the implementation, this design was devised so that guest partitions are placed in common time-domains and secure task partitions are placed in domain-0. Since the idea of secure tasks is to encapsulate extended kernel services or shared drivers, these can be configured with lower priorities, executing according to guest needs and based on the latter's priority semantics. In addition, this models allows for the coexistence of event-driven and background partitions in domain-0, while supporting guests with real-time needs and that require a guaranteed execution bandwidth. For example, a driver with the need for a speedy reaction to interrupts could be split in two cooperating tasks: a high priority task acting as the interrupt handler, which upon interrupt-triggering would message the second lowest priority task that interfaces other partitions, executing only upon a guest request. Even though a mid-priority client guest could be interrupted by the interrupt handler, its execution time within the cycle is guaranteed. Due to possible starvation, only the tasks that act as pure servers should be configured with the lowest possible priorities in domain-0. Other partitions that may be acting as applications on their own right or may have the need to perform continuous background work should be configured with a middle range priority. It is worth mentioning that the correctness of a real-time schedule will depend on time domain budgets, partition priorities and on how partitions use communication primitives. Thus, while the hypervisor tries to provide flexible and efficient mechanisms for different real-time constraints to be met, their effectiveness will depend on the design and configuration of the system.

3.8. Interrupt Manager

Before delving into the inner workings of the interrupt manager, a more detailed explanation of TrustZone interrupt support is unavoidable. The TrustZone-aware GIC enables an interrupt to be configured as secure or non-secure. Non-secure software may access all GIC registers, but when writing bits related to secure interrupts, the operation silently fails. On the other hand, when reading a register, the bits related to secure interrupts always read as zero. Priorities are also tightly controlled. In the GIC, the priority scale is inversed, that is, low number priorities reflect the highest priorities. When secure software writes to priority registers, the hardware automatically sets the most significant bit, so non-secure interrupts are always on the least priority half of the spectrum. In addition, many of the GIC registers are banked between the two worlds. Finally, the CPU and GIC can be configured so that all secure interrupts are received in monitor mode, i.e., by the hypervisor as FIQ exceptions, and all non-secure interrupts to be directly forwarded to the non-secure world as IRQs. All of these features enable the hypervisor to have complete control over interrupts, their priority and preemption, while enabling pass-through access of guests to the GIC.

The interrupt manager works on the assumption that only one handler per interrupt exists in the system. These may reside in the kernel itself or in one of the partitions. In the first and simplest case, kernel modules register the handler with the manager at initialization time, which will be called when the interrupt occurs. At the moment, the only interrupt used by the kernel is the private timer interrupt that is used by the scheduler to time-slice time domains. If the interrupt is assigned to one partition in the configuration, the capability for the interrupt will be added to its capability space with the grant permission cleared. Partition interrupts are always initially configured as disabled and with the lowest priority possible. The details on how a interrupt is configured and handled depends on the kind of partition it is assigned to, as detailed below.

Task partitions cannot be granted access to the GIC, since, if so, by running on the secure world, they would have complete control over all interrupts. All interactions with the GIC are thus mediated by the hypervisor by invoking capability operations, such as enable or disable. These partitions receive interrupts as an IPC message. Hence, before enabling them, they must inform the interrupt manager on which port they want to receive it. The kernel will always keep task interrupts as secure, and when the interrupt takes place, it will place the message in the task's port and disable it until the task signals its completion through a hypercall. Since this process relies on IPC mechanisms, the moment in which the task takes notice of the interrupt will completely depend on what and when it uses the IPC primitives and on its scheduling priority and state. It is allowed for a task to set a priority for an interrupt, but this is truncated by a limit established during system configuration and the partitions' scheduling priority.

Guest partitions can directly interact with the GIC. For them, a virtual GIC is maintained in the virtual machine. While a guest is inactive, its interrupts are kept secure but disabled. Before running the guest, the hypervisor will restore the guest's last interrupt configurations as well as a number of other GIC registers that are banked between worlds and may be fully controlled by the guest. Active guests receive their interrupts transparently when they become pending. Otherwise, as soon as the guest becomes active, the interrupts that became pending during its inactive state are automatically triggered and are received normally through the hardware exception facilities in the non-secure world. As such, at first sight, a guest has no need to interact with the GIC or the interrupt manager through capabilities as task partitions do. However, if the capability has the right permission set, the guest can use it to signal the kernel that this interrupt is critical. If so, the interrupt is kept active while the guest is inactive, albeit with a modified priority, according to the partition's scheduling priority and a predefined configuration. Regardless of which partition is running, the kernel will receive this interrupt and manipulate the virtual GIC to set the interrupt pending as would normally happen. In addition, it will request for the scheduler to temporarily migrate the guest partition to time domain-0 (if not there already), so that it can be immediately considered for scheduling before its time domain becomes active again and handle the interrupt as fast as possible. Although the worst case interrupt latency persists as the execution cycle length minus the length of the partition's time domain, setting it as critical increases the chance of it being handled earlier, depending on the partition's priority.

Finally, in the same manner as devices, interrupts may be classified as virtual and shared among all guest partitions. These virtual interrupts are considered fixed in the virtual GIC, i.e., the currently active guest is considered the owner of the interrupt. If no guest is active, they are disabled. They are always marked as non-secure and a more intricate management of their state is needed, so as to maintain coherence to the expected behavior of the hardware. For the moment, the only virtual interrupt is the one associated with the guest's tick timer in the Zynq, TTC1.

4. Evaluation

*μ*RTZVisor was evaluated on a Xilinx Zynq-7010 (Xilinx, San Jose, CA, USA) with a dual Arm Cortex-A9 running at 650 MHz. In spite of using a multicore hardware architecture, the current implementation only supports a single-core configuration. Our evaluation focuses on the TCB size and memory footprint imposed by the hypervisor, and the impact on guest performance related to context-switch overhead and communication throughput and latency as well as interrupt latency for both guest and task partitions. On all performed measurements, hypervisor, guest and task partitions were running with caches enabled. Both the hypervisor and partition code was compiled using the Xilinx Arm GNU toolchain (Sourcery CodeBench Lite, 16 May 2015) with -O3 optimizations.

4.1. Hypervisor and TCB Size

*μ*RTZVisor's implementation encompassed the development of all needed drivers and libraries from scratch, so no third-party code is used. In this prototype, ignoring blank lines and comments, it compromises 6.5 K SLOC (source lines of code) from which 5.7 K are C++ and 800 are Arm assembly. This gives rise to a total 58 KB of *.text* in the final executable binary. This small TCB size, coupled with a small number of existent hypercalls (25, at the moment, although we expect this number to increase in the future), with a common entry point for capability access-control, results in a small attack surface to the hypervisor kernel. Nevertheless, we stress the fact that, for a given guest, its actual TCB size might be increased if it shows strong dependencies on non-secure task servers, which might further interact with other servers or guests. Hence, we consider that the effective TCB of a guest fluctuates according to its communication network. Furthermore, this small code size enables us to load and lock it in one of the ways of L2 cache at system initialization, resulting in increased performance and enhanced security, given this is an on-chip memory and cannot be tampered with from the outside.

Since partitions and communication channels are static, the number of structures used for their in-kernel representation and manipulation, including capabilities, are fixed at compile-time and, thus, the amount of memory used for data by the hypervisor for partition bookkeeping essentially depends on system configuration. For example, a scenario with seven partitions, two guests and five servers, where the guests share three of the servers, totaling the existence of 12 communication ports, the amount of fixed data is about 22 KB. The large majority of this data is related to capability representation, where the sum of the data used in all capability spaces is 15 KB. Capabilities are heavyweight structures mainly due to the needed tracking for grant and revoke mechanisms. We plan to refactor their implementation in the near future to reduce their memory footprint. For small setups such as the one presented above, we also load and lock this data in L2 cache; otherwise, it is kept in OCM as defined by the secure boot process (Section 3.1). As explained in Sections 3.4 and 3.5, translation table number and size are fixed. Nevertheless, these are also kept in OCM, given that the MMU does not access caches during page walks, and table updates would force a flush to the main memory. Other data structures with more volatile and dynamic characteristics such as those used for representing messages and memory objects are allocated using object pools for efficient allocation and eliminating fragmentation. These are kept in secure DRAM.

4.2. Context-Switch Overhead

To evaluate the overhead of the context-switch operation on guest virtualization, we started by running the Thread-Metric Benchmark Suite on an unmodified FreeRTOS, a widely used RTOS (Real-Time Operating System), guest. The Thread-Metric Benchmark Suite consists of a set of benchmarks specific to evaluate real-time operating systems (RTOSes) performance. The suite is comprised of seven benchmarks. For each benchmark, the score represents the RTOS impact on the running application, where higher scores correspond to a smaller impact. Benchmarks were executed in the native environment, and compared against the results when running on top of the hypervisor. The μRTZVisor was configured with a single time partition for which the period was varied between 1 and 20 ms, which effectively defines a tick for the hypervisor in this scenario. FreeRTOS was configured with a 1 millisecond tick and was assigned a dedicated timer so that time would not freeze during the context-switch as in the case of the virtual timer, allowing the guest to keep track of wall-clock time. Since only one guest was running, the hypervisor dispatcher was slightly modified to force a complete context-switch, so that results can translate the full overhead of the guest-switching operation. This includes the flushing of caches and TLB invalidation. Figure 6 presents the achieved results, corresponding to the normalized values of an average of 1000 collected samples for each benchmark. Each sample reflects the benchmark score for a 30 s execution time, encompassing a total execution time of 500 min for each benchmark. The results show that, for a 50 Hz switching rate (20 ms period), FreeRTOS performance degradation is less than 1% across all benchmarks. This degradation aggravates as the switching period decreases to 10, 5 and 1 millisecond, at which point the hypervisor's tick meets the guest's, averaging 1.8%, 3.6% and slightly less than 18%, respectively. Although not perceptible in the figure, the standard deviation of the measured benchmarks is null for the native version of FreeRTOS, while, for the virtualized ones, it increases as the time window period decreases. This appears to happen due to the fact that, if the FreeRTOS timer interrupt is triggered during the context-switch operation or the hypervisor's during the FreeRTOS tick handler, the interrupt latency or interrupt handling time drift. With a higher rate of switching operations, the occurrence of this interference increases.

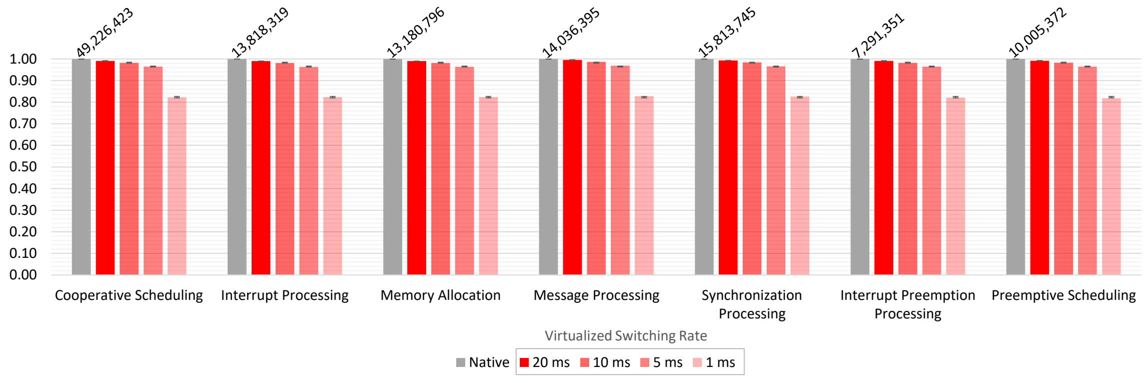

Figure 6. Thread-Metric Benchmark results, comparing the native execution of a FreeRTOS guest against its virtualized execution with different switching-rates.

Although the Thread-Metric benchmarks are useful to compare the relative performance of RTOSes by measuring common RTOS functionality performance, these are synthetic benchmarks and do not reflect the system's operation under realistic workload scenarios. We complement this assessment with the MiBench Embedded Benchmark Suite. Its 35 benchmarks are split into six suites, each targeting a specific area of the embedded market. The six categories are automotive (and industrial control), consumer devices, office automation, networking, security, and telecommunications. The benchmarks are available as standard C open-source code. We've ported the benchmarks to FreeRTOS enhanced with the FatFs file system. We focus our evaluation on the automotive suite since this is intended to assess the performance of embedded processors in embedded control systems, which we considered to be one of the target applicational areas of our system. These processors require performance in basic math abilities, bit manipulation, data input/output and simple data organization. An example of applications include air bag controllers, engine performance monitors and sensor systems. The benchmarks that emulate this scenarios include a basic math test, a bit counting test, a sorting algorithm and a shape recognition program (susan). Figure 7 shows the measured relative performance degradation for the six benchmarks in the automotive suite, each executed under a large and small input data set. The normalized column is labeled with the result for the natively executed benchmark. The benchmarks were assessed under the same conditions and hypervisor configuration as the Thread-Metric benchmarks. The results show a performance degradation of the same order of magnitude but are, nevertheless, larger than those obtained for the Thread-Metric benchmarks. A degradation of about 21.2%, 4.2%, 2% and 0.9% for the 1, 5, 10, 20 ms guest-switching rate, respectively, was obtained. We conclude that this slight increase in degradation under more realistic workloads pertains to the fact that these applications have a much larger working set than those present in the Thread-Metric Suite. Hence, the impact of a complete cache flush upon each context-switch results in a much larger cost in performance, since a much higher number of cache lines must be fetched from main memory when the guest is resumed.

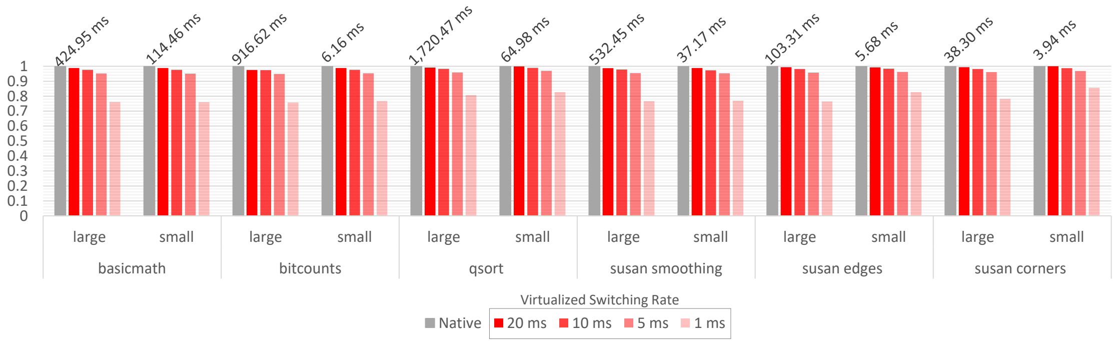

Figure 7. MiBench Automotive Benchmark results, comparing the native execution of a FreeRTOS guest against its virtualized execution with different switching-rates.

The benchmarks discussed above explain the context-switch overhead for a scenario running only guest partitions. Since guest and task partitions are represented in different kernel structures, the needed save and restore operations performed during a context-switch depend on whether the preempted partition or the incoming one are guests or tasks. All of the possible combinations give rise to four scenarios: guest–guest, task–task, guest–task and task–guest. The last scenario subdivides into two: when switching between a guest, to a task, back to the same guest, as in the case of RPC communication, there is no need for cache and TLB maintenance. However, when a guest is scheduled after a task, and the last active guest was not the same, cache flushing and TLB invalidation need to be performed. We measured the time of the context-switching operation for each scenario, running partitions in separate 10 ms time domains. Table 2 shows the average of 10,000,000 samples of each measured operation. First, we note that operations involving the invalidation of caches are an order of magnitude higher than the rest, and more than 15 times higher than the lowest one. Switching between guest and tasks or between tasks takes much less time, about 19.4 and 10.4 µs, respectively. These latter are the times involved in a brief RPC operation from a guest to a task, using the time donation facilities. This validates our initial premise that executing services in secure tasks brings great performance benefits, which is also supported by the synchronous communication results shown in Section 4.4.

Table 2. Context-switch operation execution time (μs).

Guest–Guest	166.68
Task–Task	10.38
Guest–Task	19.13
Task–Guest	19.63
Task–Different Guest	156.00

4.3. Interrupt Latency

To assess partition interrupt latency, which is defined as the time from the moment an interrupt is triggered until the final handler starts to execute, we set up a dedicated timer to trigger an interrupt every 10 ms. We measured the latency both for when the interrupt is assigned to a guest and to a task, collecting 1,000,000 samples for each case. In both cases, the scheduling configuration scenario contains two time domains in addition to domain-0, each configured with a 10 ms time window. Each time domain is assigned a different guest, and domain-0 is assigned two task partitions. The handler partition is always configured with the highest priority in the system, to translate high priority interrupt semantics when the handler partition is inactive. In addition, a domain-0 time window is enough and never completely depleted by the interrupt handler's execution, allowing the partition to react as quick as possible to interrupt events, by executing as a high priority partition in domain-0.

In the case of a guest OS partition, the interrupt is configured as critical, which means that, when the guest is inactive, the hypervisor will receive the interrupt and migrate the guest to domain-0 so that it can be immediately scheduled. Figure 8a shows the relative frequency histogram for a guest handler interrupt latencies. From the collected data, three different result regions stand out, which we identify as depending on the currently active partition at the moment the interrupt is triggered: handler guest, task or a different guest. It is clear that when the interrupt is triggered while the handler guest is executing, the measured latency is minimal since no hypervisor intervention is needed. The guest receives the interrupt transparently through normal hardware exception mechanisms and the final interrupt handler executes within an average 0.22 μs and a 0.25 μs WCET (Worst-Case Execution Time). When a different partition is active at the trigger point, the interrupt latency increases significantly since this involves the hypervisor itself receiving the interrupt, migrating the guest partition to domain-0, performing the scheduling and finally the context-switch operation. As explained in the previous section, the latter differs depending on what kind of partition is preempted, which results in an average latency of 29.03 and 180.64 μs and a WCET of 29.93 and 181.36 μ and when a task and guest are preempted, respectively. This shows that, although improving the best case scenario, placing the interrupt handler on a guest partition might result in unacceptably high latencies when a different guest needs to be preempted and caches flushed upon context-switch.

When the interrupt is handled by a task partition, the hypervisor always receives the interrupt and forwards it to the task via an IPC message. In our test scenario, the handler task is configured as the highest priority partition, executing in domain-0. Since running a high priority partition in domain-0 quickly depletes its time budget, the partition cyclically blocks on the port on which it receives the interrupt message. Figure 8b shows the relative frequency histogram with two distinct regions corresponding to the cases that result in a different task or guest preemption when the interrupt is triggered and the task unblocks by receiving the interrupt message. The results show an average 20.68 or 30.32 μs for preempted task and guest partitions, respectively. This latency arises given the extra overhead imposed by the message sending, scheduling and context-switch operations. From the presented results we conclude that, while the direct handling of an interrupt by a guest will yield a best case result, placing the interrupt handler's top half in a high priority domain-0 task (which performs the critical handling operations and notifies the bottom half executing on a different partition), will guarantee lower average and more deterministic latencies.

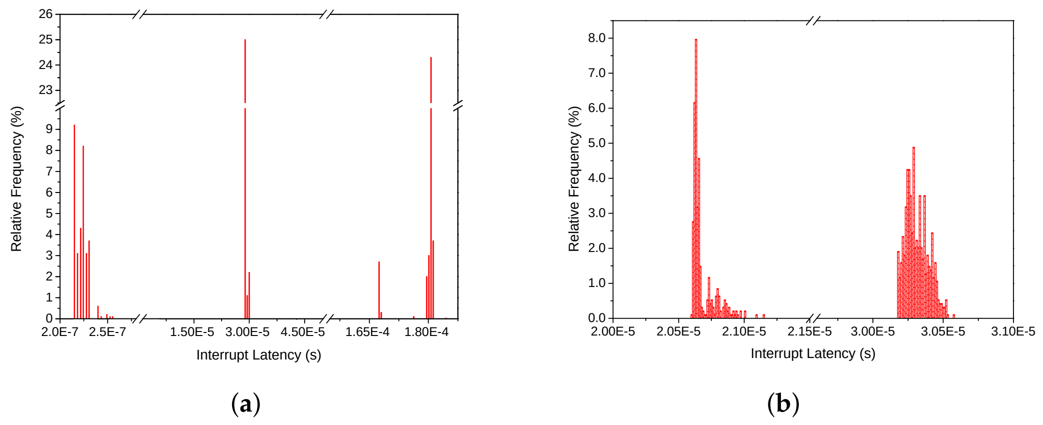

Figure 8. Relative frequency histogram of interrupt latencies (s). (**a**) guest interrupt latencies; (**b**) task interrupt latencies.

We note that, although our test scenario only contemplates tasks running in domain-0, and guests running in their own time domain, the reverse would yield similar results. Running a task in its own time domain would accrue the case on which the interrupt is triggered during the tasks execution, resulting in approximately the same latency for when a different task is preempted. A guest running in domain-0 as a high priority partition would yield the same preemption resulting latencies as those shown in Figure 8a.

4.4. IPC

To evaluate the performance of communication mechanisms, we devised scenarios for both asynchronous and synchronous communication. Table 3 shows the times needed to perform the asynchronous `Send`, `Receive` and `SendReceive` operations. Despite the performance varying slightly depending on whether the running partition is a guest or a secure task, this variation is not considered to be significant in case of asynchronous communication. As such, the performed measurements only reflect the time that it takes to perform the respective hypercalls from a guest partition. For a 64 byte message size, the hypercall execution time is of 4.36, 4.17 and 5.49 µs for each operation, respectively. These times increased by about 1 µs for each additional 64 bytes in the message. In all cases, there is one copy to be made. In the `Send` and `SendReceive` hypercalls, from the guest's address space into the port's message buffer, and the opposite for the `Receive` hypercall. There is a slight difference between the `Send` and `SendReceive` execution times, depending on the recipients' state when the operations are performed. In our implementation, this dictates whether the communication is synchronous or asynchronous. The `SendReceive` operation behaves similarly to the `Send` operation, with the addition of granting a one-time capability to the recipient guest, which only degrades performance by around 1 µs.

Table 3. Asynchronous IPC primitives latency (µs).

Message Size (bytes)	Send	Receive	Send Receive
64	4.36	4,17	5,49
128	5.17	4.75	6.31
192	6.00	5.16	7.13
256	6.82	5.72	7.98
320	7.69	6.21	8.80

In Table 4, the time measurements for synchronous IPC are presented. To infer about synchronous communication performance, we have prepared three tests, performed under three different scenarios. The first test consists on measuring an elemental `Send` operation, ensuring that the recipient is blocked, resulting in a synchronous operation. The second (One Way test) and the third (Two Way test) tests encompass time-slice donations procedures. In the latter, a full RPC communication cycle is measured, where the client partition waits for the server's response, while the former is used to measure only half of that cycle, i.e., the time that takes for a request to get the a blocked server. Each test scenario was performed between two guests (a); two tasks (b); and between one guest and one task (c). In each scenario, we kept the scheduling configurations simple, with only two partitions running concurrently, in this way reducing scheduling operations to the bare minimum. We note that the performance of synchronous IPC, as mentioned in Section 3.6, heavily relies on the performance of time-slice donation services provided by Scheduler and data transfer services provided by the Memory Manager, which is reflected in the achieved results. The first test evaluates the time that it takes to deliver a message to a blocked recipient, without scheduling involved. In this way, we can measure the latency introduced only by the data transfer. By comparing this value with the other two, we can measure the overhead introduced by our approach to build the donation chain and consequent scheduling operation. Whenever donating, there's a context-switch involved, and as such, the overhead imposed by this operation, as shown in Section 4.2, also applies in this context. Thus, in scenario (a); in which there are two guests communicating, results in the largest overhead in every performed test. Consequently, donation scenarios (b) and (c) involve tasks that are far more efficient. Regarding message size variation, we see that with the increase of 64 bytes in message size, the latency increases in just 1 to 2 µs, which is not too significant.

Table 4. Synchronous IPC communication latency (µs).

Message Size (bytes)	**(a) Guest–Guest**			**(b) Task–Task**			**(c) Guest–Task**		
	Send	**One Way**	**Two Way**	**Send**	**One Way**	**Two Way**	**Send**	**One Way**	**Two Way**
64	15.21	195.14	385.73	5.46	20.17	42.90	6.01	28.97	63.50
128	16.24	197.23	389.45	6.18	20.96	44.35	7.15	30.30	66.60
192	16.78	199.76	394.18	6.88	21.74	45.87	8.37	30.89	68.63
256	18.58	202.65	398.10	7.57	22.50	47.31	9.48	32.32	71.49
320	18.88	204.66	402.39	8.28	23.42	48.83	10.68	33.25	73.77

Synchronous communication encompasses an increased overhead, regarding, donation and scheduling operations and resulting context-switches. On the other hand, it reduces latency in service provision. As for asynchronous communication, the time involved to perform each elemental operation is smaller. Nevertheless, it should be taken into account that in client–server scenarios, the respective response could take more time than with synchronous communication, since partitions other than the server might be scheduled in the meantime. As such, the more partitions there are in a system, the bigger the latency for the response would be.

5. Related Work

Over the last few years, several embedded hypervisors have been developed in the embedded systems domain. This plethora of embedded virtualization solutions were naturally conceived following different design principles, adopting different system architectures (e.g., monolithic, microkernel, exokernel), relying on different hardware technologies (e.g., Arm VE, Arm TrustZone, Intel VT, Imagination MIPS VZ), and targeting different application domains (e.g., aerospace, automotive, industrial) [14,32–40,46–49]. Due to the extensive list of existing works under this scope, we will focus our description exclusively on TrustZone-assisted hypervisors (Section 5.1) and microkernel-based virtualization solutions (Section 5.2).

5.1. TrustZone-Assisted Virtualization

The idea of using TrustZone technology to assist virtualization in embedded systems is not new, and the first works exploiting the intrinsic virtualization capabilities of TrustZone were proposed some years ago. The majority of existing solutions just implement dual-OS support, due to the perfect match between the number of guests and the number of virtual states supported by the processors.

Winter pioneered research around the use of TrustZone technology for virtualization. In [38], Winter introduced a virtualization framework for handling non-secure world guests, and presented a prototype based on a secure version of the Linux-kernel that was able to boot only an adapted Linux kernel as non-secure world guest. Later, Cereia et al. [39] described an asymmetric virtualization layer implemented on top of the TrustZone technology in order to support the concurrent execution of both an RTOS and a GPOS (General-Purpose Operating System) on the same processor. The system was deployed and evaluated on an emulator, and never reached the light of a real hardware platform. In [35], Frenzel et al. proposes the use of TrustZone technology to implement the Nizza secure architecture. The system consists of a minimal adapted version of Linux-kernel (as normal world OS) on top of a hypervisor running on the secure world side. SafeG [40], from the TOPPERS Project [50], is a dual-OS open-source solution that takes advantage of Arm TrustZone extensions to concurrently execute an RTOS and a GPOS on the same hardware platform. Secure Automotive Software Platform (SASP) [51] is a lightweight virtualization platform based on TrustZone that consolidates a control system with a in-vehicle infotainment (IVI) system, while simultaneously guaranteeing secure device access for the consolidated automotive software. LTZVisor [36], from the TZVisor Project [52] is an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticality systems. The hypervisor supports the coexistence of an RTOS side by side with a GPOS.

Recently, in [37], Pinto et al. presented RTZVisor. The distinct aspect of RTZVisor is the implementation of the multi-OS support. To the best of our knowledge, RTZVisor has proven to be the unique TrustZone-assisted hypervisor allowing the coexistence of multiple OSes on the same hardware platform (without VE support). Finally, VOSYSmonitor [46], developed by Virtual Open Systems, enables concurrent execution of two operating systems, such as a safety critical RTOS and a GPOS. VOSYSmonitor distinguishes from other existing works because it is implemented for Armv8-A processors, which has the option to use Arm VE for running an hypervisor such as KVM [32] , a Linux Kernel-based Virtual Machine, on the non-secure world side.

5.2. Microkernels-Based Virtualization

Microkernels were not always a viable solution, due to the architecture's reliance on IPC, which constituted a bottleneck on system performance. The L4 microkernel, developed by Jochen Liedtke, appeared to break the stigma surrounding microkernels, proving their utility when providing efficient IPC mechanisms. L4 is the root of a family tree of microkernels that have a proven record of efficient performance and reliability, by following the core idea of kernel minimality and policy-void mechanisms [28]. In this section, we briefly survey some members of this family that served as the main source of inspiration for the ideas implemented in μRTZVisor, emphasizing those which aim to support virtualization.

Fiasco is an open-source descendant of L4 implemented in C++ aimed at real-time systems. It implements protected address spaces, synchronous IPC and a scheduler with multiple fixed-priority levels, whereby kernel executes a round-robing algorithm on threads characterized with the same priority [53,54]. The latest version Fiasco.OC also includes capabilities for access-control, which are propagated through IPC [55]. Fiasco has been applied to virtualization through a para-virtualization oriented technique named OS rehosting, which aligns the kernel interface with the CPU model that is assumed by operating system kernels [56].

The NOVA microhypervisor proposes a solution that deallocates virtualization to user space, which will inherently incur performance overhead and augmented engineering effort due to the highly para-virtualized approach, although augmenting security by significantly reducing TCB's size [31]. As such, the kernel solely provides services for spacial and temporal isolation, in addition to message passing and synchronization mechanisms. In addition, kernel operations require capabilities to access the kernel objects. Capabilities are immutable, and inaccessible in user-space, thus the static system configuration prevails. Ref. [57] presents Mini-NOVA, a simplified version of NOVA ported the Arm cortex-A9 architecture from the original x86 implementation. It aims at achieving lower overhead, smaller TCB size and higher security, thus making it more flexible and portable for embedded-systems. It is worth mentioning that it has the ability to dispatch hardware tasks to virtual machines through the dynamic partial reconfiguration technology.

PikeOS is an early spin-off of the L4 microkernel, whose purpose is to address requirements of safety-critical real-time embedded systems. It features spacial and temporal isolation, favoring minimum code size, in some cases to the detriment of flexibility [8]. Its scheduling algorithm was a huge inspiration for us as explained in Section 3.7, and it aims at providing a system that enables the coexistence of time-driven and event-driven partitions. The result is not the perfect fit for this kind of system, although by properly configuring each partition, it is possible to achieve a considerably good compromise [45].

OKL4 adopts a microkernel approach completely directed at virtualization and, thus, is dubbed a microvisor [21]. It meets the efficiency of the best hypervisor and the generality and minimality of the microkernel. It provides the abstraction of a virtual machine by providing virtual CPUs, virtual MMUs and TLB, and virtual interrupts. Nevertheless, these are manipulated by guests in a para-virtualized manner, incurring performance costs. It features a fast and reliable IPC, which is abstracted by channels and virtual interrupts for synchronization purposes. It implements only asynchronous IPC, which maps better to the VM model, and is less susceptible to DOS attacks. By the heritage of its seL4 predecessor, it provides access-control facilities based on capabilities, since any kernel operation requires one. OKL4 has been augmented to take advantage of the Arm virtualization extensions and support unmodified guest OSes [33].

6. Conclusions

Modern day embedded systems are becoming increasingly complex and network-oriented. At the same time, they are supporting a number of safety and security-critical infrastructures. As such, the need for highly reliable and dependable systems consequently arises. Microkernel-based virtualization has proven to be a valid solution to guarantee functionality encapsulation and fault-containment, providing an adequate environment for mixed-critically systems, while relying on a minimal TCB. However, these systems often follow a para-virtualized approach, requiring a high engineering effort for the porting and hosting of guest OSes. Hardware-supported virtualization technologies have been widely used to mitigate these obstacles. Arm's TrustZone technology stands out given its wide presence in low to mid-range processors used in embedded systems.

This work describes μRTZVisor, a hypervisor that leverages Arm TrustZone security extensions to provide a close to fully-virtualized environment on the non-secure side of the processor. Drawing inspiration from microkernel ideas, it also allows for the execution of tasks on secure user mode, towards encapsulating extra functionality such as drivers or other services and which are accessed via IPC facilities tightly coupled with scheduling mechanisms that support fast RPC and real-time behavior. All resource accesses are wrapped by a capability-based access control mechanism, which follows a design-time configuration and statically defines partition privileges and possible communication channels. Results show that μRTZVisor presents a small TCB (approximately 60 KB) and imposes small performance overhead for isolated, unmodified guest OSes (less than 2% for a 20 ms guest-switching rate). Nevertheless, the largest hindrance of the TrustZone-based virtualization approach continues to be the need for cache flushing and TLB invalidation when supporting concurrent guests on the secure side of the processor. This leads to unacceptably high context-switching times, interrupt and communication latencies when multiple guests are involved. However, the use of secure world tasks seems to mitigate these shortcomings. Placing interrupt top halves on high priority secure tasks results in lower average and more deterministic interrupt latency. In addition, communication between guests and tasks presents much better performance especially for RPC communication taking advantage of time-slice donations. This encourages our microkernel-like approach of placing extra kernel functionality in secure tasks to guarantee a small TCB, while enabling the easy plugging of extra functionality in TrustZone-base virtualization.

Future Work

In the future, we will further study cache behavior in TrustZone-enabled platforms, to explore ideas on how to achieve better performance for multiple guest support and guest-to-guest communication. Additionally, we plan to explore available DMA facilities present on the target platforms for efficient IPC data transfer. Still regarding DMA, we also plan to study TrustZone features on compatible DMA controllers to allow for guests and tasks to use and share these facilities in a secure and isolated manner. Secure tasks are, by definition, secure kernel extensions for the provision of some shared service. The already provided isolation is desired for fault-containment, although it could incur performance degradation due to high-traffic IPC, when these services are extensively used. In the future, we would like for tasks to be configured to either be a security extension of the kernel (a secure task) or to be in the kernel itself, allowing the system's designer to decide between performance and security. In addition, we would like to maintain the IPC-like interface for both configuration scenarios, so that the service provision happens transparently to guests in the sense that, regardless of where the service is placed, accessing it happens in the same manner. The assessment of application benchmarks running on guests dependent on these services is also imperative, in order to quantify the overhead introduced by RPC in realistic workload scenarios. We would also like to port the current implementation to other Arm architectures and platforms supporting TrustZone, as well as to port other OSes to run over μRTZVisor. Analyzing the system's functioning on different contexts would allow us to identify some possible enhancements. Until now, μRTZVisor targets single-core architectures. We plan to augment our kernel to a multi-core implementation, which could result in improved real-time support and increased performance. It is also in our best interest to explore virtual machine migration, a technique widely used in server's virtualization with proven benefits [58]. To the best of our knowledge, this is a topic with little to no research in embedded systems' virtualization solutions. We think it could bring more efficient energy management through load balancing among application clusters, which we think may have applicability, for example, in industrial IoT applications. Finally, for a more thorough evaluation of our solution, we intend to compare micro and application benchmarks against other available embedded virtualization solutions (ideally open-source), under the same execution environment (i.e., same platform, same guest OS, same workload, etc.).

Acknowledgments: This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT-Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013.

Author Contributions: Sandro Pinto conceived, designed and implemented RTZVisor. Jose Martins, Joao Alves, Adriano Tavares and Sandro Pinto conceived and designed μRTZVisor. Both Jose Martins and Joao Alves implemented μRTZVisor. Jose Martins, Joao Alves and Sandro Pinto designed and carried out the experiments. Jorge Cabral and Adriano Tavares contributed to data analysis. All authors contributed to the writing of the paper.

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

References

1. Heiser, G. The role of virtualization in embedded systems. In Proceedings of the ACM 1st Workshop on Isolation and Integration in Embedded Systems, Scotland, UK, 1–4 April 2008; pp. 11–16, doi:10.1145/1435458.1435461.
2. Herder, J.N.; Bos, H.; Tanenbaum, A.S. *A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers*; Technical Report IR-CS-018; Vrije Universiteit: Amsterdam, The Netherlands, 2006.
3. Moratelli, C.; Johann, S.; Neves, M.; Hessel, F. Embedded virtualization for the design of secure IoT applications. In Proceedings of the IEEE 2016 International Symposium on Rapid System Prototyping (RSP), Pittsburg, PA, USA, 6–7 October 2016; pp. 1–5, doi:10.1145/2990299.2990301.

4. Pinto, S.; Gomes, T.; Pereira, J.; Cabral, J.; Tavares, A. IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices. *IEEE Internet Comput.* **2017**, *21*, 40–47, doi:10.1109/MIC.2017.17.
5. Heiser, G. Virtualizing embedded systems: Why bother? In Proceedings of the ACM 48th Design Automation Conference, San Diego, CA, USA, 5–10 June 2011; pp. 901–905, doi:10.1145/2024724.2024925.
6. Reinhardt, D.; Morgan, G. An embedded hypervisor for safety-relevant automotive E/E-systems. In Proceedings of the 2014 9th IEEE International Symposium on Industrial Embedded Systems (SIES), Pisa, Italy, 18–20 June 2014; pp. 189–198, doi:10.1109/SIES.2014.6871203.
7. Kleidermacher, D.; Kleidermacher, M. *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*; Elsevier: Amsterdam, The Netherlands, 2012.
8. Kaiser, R. Complex embedded systems-A case for virtualization. In Proceedings of the IEEE 2009 Seventh Workshop on Intelligent solutions in Embedded Systems, Ancona, Italy, 25–26 June 2009; pp. 135–140.
9. Aguiar, A.; Hessel, F. Embedded systems' virtualization: The next challenge? In Proceedings of the 2010 21st IEEE International Symposium on Rapid System Prototyping (RSP), Fairfax, VA, USA, 8–11 June 2010; pp. 1–7, doi:10.1109/RSP.2010.5656430.
10. Acharya, A.; Buford, J.; Krishnaswamy, V. Phone virtualization using a microkernel hypervisor. In Proceedings of the 2009 IEEE International Conference on Internet Multimedia Services Architecture and Applications (IMSAA), Bangalore, India, 9–11 December 2009; pp. 1–6, doi:10.1109/IMSAA.2009.5439460.
11. Rudolph, L. A virtualization infrastructure that supports pervasive computing. *IEEE Perv. Comput.* **2009**, *8*, doi:10.1109/MPRV.2009.66.
12. Da Xu, L.; He, W.; Li, S. Internet of things in industries: A survey. *IEEE Trans. Ind. Inform.* **2014**, *10*, 2233–2243, doi:10.1109/TII.2014.2300753.
13. Sadeghi, A.R.; Wachsmann, C.; Waidner, M. Security and privacy challenges in industrial internet of things. In Proceedings of the 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 8–12 June 2015; pp. 1–6, doi:10.1145/2744769.2747942.
14. Joe, H.; Jeong, H.; Yoon, Y.; Kim, H.; Han, S.; Jin, H.W. Full virtualizing micro hypervisor for spacecraft flight computer. In Proceedings of the 2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC), Williamsburg, VA, USA, 14–18 October 2012; doi:10.1109/DASC.2012.6382393.
15. Shuja, J.; Gani, A.; Bilal, K.; Khan, A.U.R.; Madani, S.A.; Khan, S.U.; Zomaya, A.Y. A Survey of Mobile Device Virtualization: Taxonomy and State of the Art. *ACM Comput. Surv.* (*CSUR*) **2016**, *49*, 1, doi:10.1145/2897164.
16. Hohmuth, M.; Peter, M.; Hartig, H.; Shapiro, J.S. Reducing TCB size by using untrusted components: Small kernels versus virtual-machine monitors. In Proceedings of the 11th Workshop on ACM SIGOPS European Workshop ACM, Leuven, Belgium, 19–22 September 2004; p. 22, doi:10.1145/1133572.1133615.
17. Murray, D.G.; Milos, G.; Hand, S. Improving Xen security through disaggregation. In Proceedings of the ACM Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Seattle, WA, USA, 5–7 March 2008; pp. 151–160, doi:10.1145/1346256.1346278.
18. Lackorzynski, A.; Warg, A.; Volp, M.; Hartig, H. Flattening hierarchical scheduling. In Proceedings of the ACM Tenth ACM International Conference on Embedded Software, New York, NY, USA, 7–12 October 2012; pp. 93–102, doi:10.1145/2380356.2380376.
19. Gu, Z.; Zhao, Q. A state-of-the-art survey on real-time issues in embedded systems virtualization. *J. Softw. Eng. Appl.* **2012**, *5*, 277, doi:10.4236/jsea.2012.54033.
20. Armand, F.; Gien, M. A practical look at micro-kernels and virtual machine monitors. In Proceedings of the 6th IEEE Consumer Communications and Networking Conference (CCNC 2009), Las Vegas, NV, USA, 10–13 January 2009; pp. 1–7, doi:10.1109/CCNC.2009.4784874.
21. Heiser, G.; Leslie, B. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In Proceedings of the ACM First ACM Asia-Pacific Workshop on Workshop on Systems, New Delhi, India, 30 August 2010; pp. 19–24, doi:10.1145/1851276.1851282.
22. Heiser, G. Secure embedded systems need microkernels. *USENIX Login* **2005**, *30*, 9–13.
23. Tanenbaum, A.S.; Herder, J.N.; Bos, H. Can we make operating systems reliable and secure? *Computer* **2006**, *39*, 44–51, doi:10.1109/MC.2006.156.
24. Kuz, I.; Liu, Y.; Gorton, I.; Heiser, G. CAmkES: A component model for secure microkernel-based embedded systems. *J. Syst. Softw.* **2007**, *80*, 687–699, doi:10.1016/j.jss.2006.08.039.
25. Herder, J.N.; Bos, H.; Gras, B.; Homburg, P.; Tanenbaum, A.S. Modular system programming in MINIX 3. *USENIX Login* **2006**, *31*, 19–28.

26. Hartig, H.; Hohmuth, M.; Liedtke, J.; Wolter, J.; Schonberg, S. The performance of μ-kernel-based systems. In Proceedings of the ACM SIGOPS Operating Systems Review, Saint Malo, France, 5–8 October 1997; Volume 31, pp. 66–77, doi:10.1145/268998.266660.
27. Leslie, B.; Van Schaik, C.; Heiser, G. Wombat: A portable user-mode Linux for embedded systems. In Proceedings of the 6th Linux.Conf.Au, Canberra, Australia, 18–23 April 2005; Volume 20.
28. Elphinstone, K.; Heiser, G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles ACM, New York, NY, USA, 3–6 November 2013; pp. 133–150, doi:10.1145/2517349.2522720.
29. Ruocco, S. A real-time programmer's tour of general-purpose L4 microkernels. *EURASIP J. Embed. Syst.* **2007**, *2008*, 234710, doi:10.1155/2008/234710.
30. Uhlig, R.; Neiger, G.; Rodgers, D.; Santoni, A.L.; Martins, F.C.; Anderson, A.V.; Bennett, S.M.; Kagi, A.; Leung, F.H.; Smith, L. Intel virtualization technology. *Computer* **2005**, *38*, 48–56, doi:10.1109/MC.2005.163.
31. Steinberg, U.; Kauer, B. NOVA: A microhypervisor-based secure virtualization architecture. In Proceedings of the ACM 5th European Conference on Computer Systems, New York, NY, USA, 13–16 April 2010; pp. 209–222, doi:10.1145/1755913.1755935.
32. Dall, C.; Nieh, J. KVM/ARM: The design and implementation of the linux ARM hypervisor. In *ACM Sigplan Notices*; ACM: New York, NY, USA, 2014; Volume 49, pp. 333–348, doi:10.1145/2541940.2541946.
33. Varanasi, P.; Heiser, G. Hardware-supported virtualization on ARM. In Proceedings of the ACM Second Asia-Pacific Workshop on Systems, Shanghai, China, 11–12 July 2011; p. 11, doi:10.1145/2103799.2103813.
34. Zampiva, S.; Moratelli, C.; Hessel, F. A hypervisor approach with real-time support to the mips m5150 processor. In Proceedings of the IEEE 2015 16th International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 2–4 March 2015; pp. 495–501, doi:10.1109/ISQED.2015.7085475.
35. Frenzel, T.; Lackorzynski, A.; Warg, A.; Härtig, H. Arm trustzone as a virtualization technique in embedded systems. In Proceedings of the Twelfth Real-Time Linux Workshop, Nairobi, Kenya, 25–27 October 2010.
36. Pinto, S.; Pereira, J.; Gomes, T.; Tavares, A.; Cabral, J. LTZVisor: TrustZone is the Key. In Proceedings of the 29th Euromicro Conference on Real-Time Systems (Leibniz International Proceedings in Informatics), Dubrovnik, Croatia, 28–30 June 2017; Bertogna, M., Ed.; Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany; Volume 76, doi:10.4230/LIPIcs.ECRTS.2017.4.
37. Pinto, S.; Pereira, J.; Gomes, T.; Ekpanyapong, M.; Tavares, A. Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems. *IEEE Comput. Archit. Lett.* **2016**, doi:10.1109/LCA.2016.2617308.
38. Winter, J. Trusted computing building blocks for embedded linux-based ARM Trustzone platforms. In Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, Alexandria, VA, USA, 31 October 2008; pp. 21–30, doi:10.1145/1456455.1456460.
39. Cereia, M.; Bertolotti, I.C. Virtual machines for distributed real-time systems. *Comput. Stand. Interfaces* **2009**, *31*, 30–39, doi:10.1016/j.csi.2007.10.010.
40. Sangorrin, D.; Honda, S.; Takada, H. Dual operating system architecture for real-time embedded systems. In Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium, 7–9 July 2010; pp. 6–15.
41. Alves, T.; Felton, D. TrustZone: Integrated Hardware and Software Security. *Technol. Depth* **2004**, *3*, 18–24.
42. Steinberg, U.; Wolter, J.; Hartig, H. Fast component interaction for real-time systems. In Proceedings of the 17th Euromicro Conference on Real-Time Systems, (ECRTS 2005), Washington, DC, USA, 6–8 July 2005; pp. 89–97, doi:10.1109/ECRTS.2005.16.
43. Herder, J.N.; Bos, H.; Gras, B.; Homburg, P.; Tanenbaum, A.S. Countering ipc threats in multiserver operating systems (a fundamental requirement for dependability). In Proceedings of the 2008 14th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'08), Taipei, Taiwan, 15–17 December 2008; pp. 112–121, doi:10.1109/PRDC.2008.25.
44. Shapiro, J.S. Vulnerabilities in synchronous IPC designs. In Proceedings of the IEEE 2003 Symposium on Security and Privacy, Berkeley, CA, USA, 11–14 May 2003; pp. 251–262, doi:10.1109/SECPRI.2003.1199341.
45. Kaiser, R.; Wagner, S. Evolution of the PikeOS microkernel. In Proceedings of the First International Workshop on Microkernels for Embedded Systems, Sydney, Australia, 16 January 2007; p. 50.

46. Lucas, P.; Chappuis, K.; Paolino, M.; Dagieu, N.; Raho, D. VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A. In Proceedings of the 29th Euromicro Conference on Real-Time Systems (Leibniz International Proceedings in Informatics), Dubrovnik, Croatia, 28–30 June 2017; Marko, B., Ed.; Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2017; Volume 76; doi:10.4230/LIPIcs.ECRTS.2017.6.
47. Masmano, M.; Ripoll, I.; Crespo, A.; Metge, J. Xtratum: A hypervisor for safety critical embedded systems. In Proceedings of the 11th Real-Time Linux Workshop, Dresden, Germany, 28–30 September 2009; pp. 263–272.
48. Ramsauer, R.; Kiszka, J.; Lohmann, D.; Mauerer, W. Look Mum, no VM Exits! (Almost). In Proceedings of the 13th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Dubrovnik, Croatia, 14 April 2017.
49. Pak, E.; Lim, D.; Ha, Y.M.; Kim, T. Shared Resource Partitioning in an RTOS. In Proceedings of the 13th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Dubrovnik, Croatia, 14 April 2017.
50. Toppers.jp. Introduction to the SafeG. 2017. Available online: http://www.toppers.jp/en/safeg.html (accessed on 29 September 2017).
51. Kim, S.W.; Lee, C.; Jeon, M.; Kwon, H.; Lee, H.W.; Yoo, C. Secure device access for automotive software. In Proceedings of the IEEE 2013 International Conference on Connected Vehicles and Expo (ICCVE), Las Vegas, NV, USA, 2–6 December 2013; pp. 177–181, doi:10.1109/ICCVE.2013.6799789.
52. Tzvisor.org. TZvisor—TrustZone-assisted Hypervisor. 2007. Available online: http://www.tzvisor.org (accessed on 29 September 2017).
53. Schierboom, E.G.H. Verification of Fiasco's IPC Implementation. Master's Thesis, Computing Science Department, Radboud University, Nijmegen, The Netherlands, 2007.
54. Steinberg, U. Quality-Assuring Scheduling in the Fiasco Microkernel. Master's Thesis, Dresden University of Technology, Dresden, Germany, 2004.
55. Smejkal, T.; Lackorzynski, A.; Engel, B.; Völp, M. Transactional IPC in Fiasco.OC. In Proceedings of the 11th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Lund, Sweden, 7–10 July 2015; pp. 19–24.
56. Lackorzynski, A.; Warg, A.; Peter, M. Virtual processors as kernel interface. In Proceedings of the Twelfth Real-Time Linux Workshop, Nairobi, Kenya, 25–27 October 2010.
57. Xia, T.; Prévotet, J.C.; Nouvel, F. Mini-nova: A lightweight arm-based virtualization microkernel supporting dynamic partial reconfiguration. In Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), Hyderabad, India, 25–29 May 2015; pp. 71–80, doi:10.1109/IPDPSW.2015.72.
58. Voorsluys, W.; Broberg, J.; Venugopal, S.; Buyya, R. Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. *Cloud Com* **2009**, *9*, 254–265.