LTZVisor: TrustZone is the Key*

Sandro Pinto[1], Jorge Pereira[2], Tiago Gomes[3], Adriano Tavares[4], and Jorge Cabral[5]

1 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`sandro.pinto@dei.uminho.pt`
2 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`jorge.m.pereira@algoritmi.uminho.pt`
3 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`tgomes@dei.uminho.pt`
4 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`atavares@dei.uminho.pt`
5 **Centro Algoritmi, Universidade do Minho, Guimarães, Portugal**
`jcabral@dei.uminho.pt`

Abstract

Virtualization technology starts becoming more and more widespread in the embedded systems arena, driven by the upward trend for integrating multiple environments into the same hardware platform. The penalties incurred by standard software-based virtualization, altogether with the strict timing requirements imposed by real-time virtualization are pushing research towards hardware-assisted solutions. Among existing commercial off-the-shelf (COTS) technologies, ARM TrustZone promises to be a game-changer for virtualization, despite of this technology still being seen with a lot of obscurity and scepticism. In this paper we present a Lightweight TrustZone-assisted Hypervisor (LTZVisor) as a tool to understand, evaluate and discuss the benefits and limitations of using TrustZone hardware to assist virtualization. We demonstrate how TrustZone can be adequately exploited for meeting the real-time needs, while presenting a low performance cost on running unmodified rich operating systems. While ARM continues to spread TrustZone technology from the applications processors to the smallest of microcontrollers, it is undeniable that this technology is gaining an increasing relevance. Our intent is to encourage research and drive the next generation of TrustZone-assisted virtualization solutions.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases hypervisor, virtualization, TrustZone, space and time partitioning, real-time, embedded systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.4

1 Introduction

Platform virtualization, which enables multiple operating systems (OSes) to run on top of the same hardware platform, is gaining momentum in the embedded systems arena, driven by the growing interest in consolidating and isolating multiple and heterogeneous environments [6]. While in industrial control or automotive systems virtualization has been used to integrate real-time control functionalities with high-level or infotainment environments [20, 9], in aeronautics and aerospace virtualization provides isolation for safety-critical components

* This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT – Fundação para a Ciência e Tecnologia – (grant SFRH/BD/91530/2012 and UID/CEC/00319/2013).

[10, 26]. Despite the differences among several embedded industries, all share an upward trend for integration, due to the common interest in building systems with reduced size, weight, power and cost (SWaP-C) budget [6, 10].

Typically, solutions for embedded virtualization [10, 1, 7, 26] follow two different approaches: full-virtualization and paravirtualization. Between both approaches there is a trade-off between performance and flexibility: the traditional full-virtualization [7, 26] incurs on a higher performance cost, while the static paravirtualization approach [1, 10, 26] incurs on a higher design cost. Recently, due to penalties incurred by software-based virtualization approaches, as well as the strict timing requirements and constraints imposed by real-time virtualization [31], academia and industry have recently begun focusing their attention in providing hardware support to assist virtualization. Intel introduced Intel Virtualization Technology (VT) [24], ARM presented ARM Virtualization Extensions (VE) and ARM TrustZone [28, 4, 5, 17], and, recently, Imagination/MIPS released MIPS Virtualization and OmniShield technhology [31].

Among existent COTS technologies, ARM VE and ARM TrustZone [30] have attracted particular attention, due to the ubiquitous adoption of ARM-based processors in the embedded market. Although ARM VE is the specific technology from ARM for virtualization, ARM TrustZone is also seen as a hardware-based alternative for system virtualization [5]. This technology is gaining momentum due to the supremacy and lower cost of TrustZone-enabled processors in comparison with VE-enabled processors, and because it is seen as the only implementable hardware-based approach on ARM processors where VE are not available. Examples of such processors include the well-established ARM Cortex-A9, and the newest Cortex-A32. Furthermore, due to the recent ARM announcement of introducing TrustZone technology in the new generation of Cortex-M processors [27], this technology also promises to be a game-changer in the low-end sector, opening the possibility of breaking the barrier to the adoption of system virtualization in resource-constrained embedded devices.

TrustZone technology virtualizes a physical core as two virtual cores, providing two completely separate execution domains. The non-secure world acts as a virtual machine (VM) under the control of a hypervisor running in the secure world side. Some TrustZone-based solutions for virtualization have been proposed [30, 3, 5, 22, 13, 17]. While some of them just support a single guest execution, others present a dual-OS configuration for running an RTOS side-by-side with a GPOS. The problem is that they still lack in providing detailed information about their implementation and deployment on physical platforms, as well as in performing extensive experiments and presenting convincing results. We believe that ARM TrustZone, when adequately exploited, opens up a number of opportunities for (real-time) virtualization, despite some researchers still arguing that perceiving TrustZone as a virtualization mechanism is very limiting and ill-guided [28, 8].

To give answers to a plethora of doubts and questions we developed LTZVisor (Lightweight TrustZone-assisted Hypervisor) as a tool to clearly understand and evaluate how TrustZone hardware can be efficiently exploited to assist virtualization. We describe all the details behind the implementation, highlighting its benefits and discussing identified limitations and how they can be overcome. We conducted an extensive set of experiments which clearly demonstrate how TrustZone-assisted virtualization can effectively meet real-time needs. LTZVisor is the outcome of years of our experience in working and developing TrustZone-based solutions for a multitude of applications and domains [17, 16, 18, 19, 15]. The amount of open-source software for TrustZone systems is scarce. We plan to make LTZVisor available for the open-source community, encouraging research, whilst providing the foundation to drive the next generation of TrustZone-assisted virtualization solutions.

1.1 Contributions

In this paper, we present LTZVisor with the following contributions:

- an open-source tool to understand, evaluate, and encourage research towards TrustZone-assisted virtualization;
- an extensive evaluation over a physical hardware platform with popular benchmark suites, focusing on the penalties incurred on the real-time properties of the secure guest OS, as well as on performance of the non-secure guest OS;
- a complete discussion about the identified drawbacks and advantages of using TrustZone for (real-time) virtualization, and how we suggest to overcome those limitations based on the knowledge and expertise consolidated over the years.

2 ARM TrustZone

TrustZone technology [29] refers to the security extensions introduced with ARMv6K in all ARM Cortex-A processors. The TrustZone hardware architecture can be seen as a dual-virtual system, partitioning all system's physical resources into two isolated execution environments. Recently, ARM also decided to extend TrustZone for the Cortex-M processor family [27]. TrustZone for ARMv8-M has the same high-level features as TrustZone for applications processors, with the benefit that context switching between both worlds is done in hardware for faster transitions. In the remainder of this section, when describing TrustZone, we are focused on the specificities of this technology for Cortex-A processors. The distinctive aspects of TrustZone for ARMv8-M are out of the scope of this paper.

At the processor level, the most significant architectural change is its partition into two separate worlds: the secure and the non-secure worlds. A new 33^{rd} processor bit, the *Non-Secure* (*NS*) bit, accessible through the *Secure Configuration Register* (*SCR*) register, indicates in which world the processor is currently executing, and is propagated over the memory and peripherals buses. 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 supported modes, because when the processor runs in this 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 IRQ, FIQ, and Aborts exceptions in the secure world. To ensure a strong isolation between secure and normal states, some special registers are banked, such as a number of *System Control Coprocessor* (*CP15*) registers. Some secure critical processor core bits and *CP15* registers are either totally unavailable to non-secure world or access permissions are closely under supervision of the secure world. The TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) extend TrustZone security to the memory infrastructure. TZASC can partition the DRAM into different memory regions: this hardware controller has a programming interface, accessible only from the secure side, that can be used to configure a specific memory region as secure or non-secure. By design, secure world applications can access normal world memory but the reverse is not possible. TZMA provides similar functionality but for off-chip ROM or SRAM. 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 which signals in which state the processor accesses the memory. System devices can be dynamically configured as secure or non-secure through the TrustZone Protection Controller (TZPC). 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.

3 LTZVisor: Design

The main design idea behind LTZVisor is the use of TrustZone hardware to assist virtualization. The key towards TrustZone-assisted virtualization is to rely on hardware support as much as possible, while containing software implementation and components privileges, and promoting the secure environment with a higher privilege of execution. This leads to three fundamental principles:

- *The principle of minimal implementation:* Spaghetti code is the main source of vulnerabilities in software and provides an avenue of exploitation for hackers. Relying on the hardware support of TrustZone technology for virtualization as much as possible, as well as promoting the careful design and static configuration of each hypervisor component, will definitively help us minimize the trusted computing base of the system and, consequently, contain the attack surface.
- *The principle of least privilege:* Components must be given access only to those resources (e.g., I/O devices, system services, etc) that are absolutely required. TrustZone technology guarantees, by design, that the non-secure world is always less privileged than the secure one, despite the CPU execution mode. Furthermore, in the secure world, the monitor mode introduces a third level of privileges. Exploring these features to implement a well-layered virtualization approach will help promoting privileged execution and hardware-enforced isolation of the real-time environment from the non-real-time one.
- *The principle of asymmetric scheduling:* Virtualization of a real-time environment is very challenging, mainly due to strict timing requirements and hierarchical scheduling problems that those systems introduce. The adoption of an asymmetric scheduling policy, where the secure environment has a higher privilege of execution than the non-secure one, will ensure that timing requirements are met, even while executing real-time tasks over the RTOS running on top of a virtual CPU.

3.1 General Architecture

LTZVisor provides a virtualization solution based on the two virtual execution environments provided by the TrustZone hardware. The secure world is responsible for hosting the privileged software, while the non-secure world is responsible for hosting the non-privileged software. Figure 1 depicts the proposed virtualization architecture. In this figure, three main software components can be identified: the hypervisor, the secure VM, and the non-secure VM.

LTZVisor runs in the highest privileged processor mode, i.e., in monitor mode. When running in this mode, the processor state is considered always secure. The hypervisor has full control of all hardware and software resources, and is responsible for configuring memory, interrupts and devices assigned to each VM, as well as managing the Virtual Machine Control Block (VMCB) of each VM during a partition switch. When a virtual machine is about to be executed by the physical processor, the hypervisor transfers the VM state, saved on the respective VMCB, to the physical processor context. When the hypervisor assigns the physical processor to another virtual machine, the processor context of the active VM is saved back into the respective VMCB.

The secure VM runs in the supervisor mode of the secure world side. This VM needs to have a small footprint, because when the processor state is secure it has full view over the

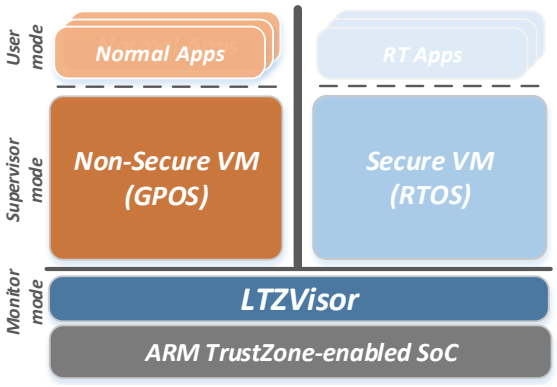


Figure 1 LTZVisor General Architecture.

non-secure world side. As such, the privileged guest code can interfere with the other virtual machine, by accessing or modifying its state or the state of its resources (memory or memory mapped devices). For this reason, the operating system hosted on the secure VM must be aware of the virtualization, and is considered part of the system's Trust Computing Base (TCB). The secure VM is ideal to run an RTOS, because the higher privilege of execution helps meeting the timing requirements of such environments. Furthermore, RTOSes typically have small memory footprints.

The non-secure VM runs in the supervisor mode of the non-secure world side. This VM is ideal to host a general purpose guest OS, useful for running human-machine interfaces as well as internet-based applications and services. The software running on the non-secure world side is completely isolated from the privileged software running on the secure world side. When the processor is operating in a privileged mode but not in the secure state, it cannot access nor modify any state information belonging to the secure world. Any attempt from the non-secure guest OS to access any resource of the secure world side immediately triggers an exception to the hypervisor.

4 LTZVisor: Implementation

LTZVisor exploits ARM TrustZone to provide time and space isolation between both partitions. The asymmetric design principle allows to preserve the real-time characteristics of the secure virtual machine (RTOS). This section provides all the details behind LTZVisor implementation, describing how CPU virtualization and memory isolation is ensured, presenting how MMU and caches are managed, describing how device partition is achieved, explaining how interrupts and time are managed for different guest OSes, and illustrating how inter-VM communication is implemented.

4.1 Virtual CPU

TrustZone technology virtualizes each physical CPU into two virtual CPUs: one for the secure world and another for the non-secure world. Between both worlds there is a list of banked registers, i.e., an individual copy of those registers exists for each world. Since each guest OS is running in a different world, in this particular case, a huge part of the virtual CPU support is guaranteed by the hardware itself, minimizing the number of registers to be saved and restored in each partition-switching operation. The VMCB of the non-secure

side is composed by 27 registers: 13 *General Purpose Registers* (*R0-R12*), the Stack Pointer (*SP*), the Linker Register (*LR*) and *Saved Program Status Register* (*SPSR*) for each of the following modes: Supervisor, System, Abort and Undef. The "high" *General Purpose Registers* (*R8-R12*), as well as the *SP*, *LR* and *SPSR* of the FIQ and IRQ modes are not included, as they are mutually exclusive for each world. Among the coprocessor registers, almost all of them are banked: only the *SCTLR* and the *ACTLR* need to be preserved. For optimization purposes, the VMCB of the secure side is composed of only 16 registers: 13 *General Purpose Registers* (*R0-R12*), the Stack Pointer (*SP*), the Linker Register (*LR*) and *SPSR* for the System mode. The Monitor mode is, by design, uniquely dedicated to the secure world side. These optimizations reduce the interrupt latency from the secure guest OS (RTOS) perspective, speeding up the transition from the non-secure to the secure world side, when a secure interrupt arises while the non-secure OS is executing.

Among the aforementioned unbanked registers, there are those which are only modifiable from the secure side: they can be read when the processor is in the non-secure state, but an attempt to modify them will be ignored. This is stated on TrustZone specification to guarantee a high degree of security in the system, which has a cost for the non-secure guest OS. For example, the *System Control Register* (*SCTLR*) and the *Auxiliary Control Register* (*ACTLR*) provide control and configuration over memory, cache, MMU, AXI accesses, etc. These registers are used to enable and disable MMU, and are only accessible in the secure state. During the non-secure guest OS boot process, an attempt to modify them will be ignored, leading the GPOS to get stuck. For that reason, the hypervisor must fill some registers of the non-secure VMCB with a specific initialization value. For example, the *SCTLR* register of the non-secure VMCB should be initialized appropriately, so that MMU and Level 1 cache of the non-secure world side are enabled before the GPOS starts booting.

4.2 Scheduler

An identified issue in virtualizing a real-time environment is the well-known hierarchical scheduling problem. Typically, a hypervisor schedules virtual CPUs while a guest RTOS running over the virtual CPU schedules its own tasks. Ensuring real-time execution of tasks over the RTOS executing on top of a virtual CPU involves a complex hierarchical scheduling analysis, requiring that both schedulers are accordingly modeled [31].

LTZVisor overcomes this problem by implementing an asymmetric or idle scheduler. This scheduling policy guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and the secure guest OS can preempt the execution of the non-secure one. In fact, the secure virtual machine (RTOS) has a higher scheduling priority than the non-secure one, and LTZVisor is not the software component that directly schedules the virtual machines, but it is scheduled itself by the secure guest OS. Although this can seem contradictory, the concept of ring protection is never jeopardized, as the LTZvisor continues executing in a more privileged mode than the secure guest OS: the hypervisor is just configured to behave in a passive way.

4.3 Memory Partition

Traditional hardware-assisted memory virtualization relies on Memory Management Unit (MMU) support for 2-level address translation, mapping guest virtual to guest physical addresses and then guest physical to host physical addresses. This MMU feature is a key enabler to run unmodified partition OSes, and also to implement isolation between partitions.

TrustZone-enabled SoCs (which are not VE-enabled) only provide MMU support for single-level address translation. Therefore, the existence of a TZASC is a major requirement

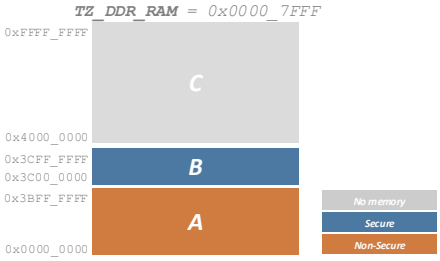


Figure 2 LTZVisor: Memory Configuration.

for the proposed solution, because this component allows partition of memory into different segments. This memory segmentation feature can be exploited to guarantee spatial isolation between the non-secure VM and the secure one, basically by adequately configuring the security state of the memory segments of respective partitions. The non-secure VM should have its own memory segment(s) configured as non-secure, and the remaining memory as secure. If the non-secure guest OS tries to access a secure memory region (either belonging to the secure partition or the hypervisor), an exception is automatically triggered and the execution control redirected to the hypervisor.

Memory segments can be configured with a specific granularity, which is implementation defined, depending on the vendor. In the hardware under which our system was deployed, Xilinx ZC702, memory regions can be configured with a granularity of 64MB. This configuration is provided via a system level control register named `TZ_DDR_RAM`. A 0 or 1 on a particular bit indicates a secure or non-secure memory region for that particular memory segment, respectively. Figure 2 depicts the memory setup and respective secure/non-secure mappings, for a virtualized system consisting of the hypervisor altogether with the secure virtual machine (B), and the non-secure virtual machine (A). In this specific configuration, the non-secure VM (GPOS) uses the first fifteen memory segments (0x00000000 – 0x3BFFFFFF), corresponding to a total of 960MB of non-secure memory. The hypervisor and the secure VM, due to their low memory footprint, use only the last available memory segment (0x3C000000 – 0x3FFFFFFF), corresponding to a 64MB of secure memory. The remainder of the 32-bit memory address space is not accessible (C), because Xilinx ZC702 is only endowed with a 1GB DDR3 memory.

4.4 MMU and Cache Management

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. This means each world has its own copy of the TTBR register set, as well as an independent MMU configuration. This reduces the list of activities to perform on each guest-switching operation, because translation lookaside buffer (TLB) entries do not need to be invalidated.

The same kind of isolation is still available at cache-level. The processor caches have been extended with an additional tag (NS bit) which records the security state of the transaction that accesses the memory. This NS bit is set by hardware and it is not directly accessible by system software. Therefore, in this cache coherence design, when the system switches between the two worlds, none of the cache lines need to be flushed. This means that this design feature at cache-level significantly improves the performance of LTZVisor, because no cache management operation needs to be performed on each guest-switching operation: cache

isolation is enforced and guaranteed by the hardware itself. On Xilinx ZC702, there are a few notes regarding the TrustZone support in L2 cache. The L2 Control register (`reg1_control`) can only be written with an access tagged as secure, which means that an attempt to enable or disable the L2 cache from the non-secure world side will be ignored. Similarly to the support that the hypervisor needs to perform in the L1 cache initialization (aforementioned in Section 4.1), LTZVisor also needs to enable the L2 cache on the secure world side before the non-secure guest OS starts booting. Once the L2 cache is enabled, maintenance operations on the non-secure entries can be performed directly from the non-secure world side.

4.5 Device Partition

TrustZone technology allows devices to be (statically or dynamically) configured as secure or non-secure. This hardware feature allows the partition of devices between both worlds while enforcing isolation at the device level.

LTZVisor implements device virtualization adopting a pass-through policy, which means devices are managed directly by guest partitions. To ensure strong isolation between them, devices are not shared between guests and are assigned to the respective partitions at design time, and then configured during boot time. The devices assigned to the RTOS (secure VM) are configured as secure devices, while devices assigned to the GPOS (non-secure VM) are configured as non-secure devices. This guarantees the GPOS cannot compromise the state of any device belonging to the RTOS, and if the non-secure guest partition tries to access a secure device then an exception will be automatically triggered and handled by hypervisor. On Xilinx ZC702, the security state of devices can be configured through a set of secure registers accesible from the secure side. This registers include, for example, the SDIO slave security registers (`security2_sdio0` and `security3_sdio1`) and the APB slave security register (`security6_apb_slaves`).

4.6 Interrupt Management

In TrustZone-enabled SoCs, the GIC supports the coexistence of secure and non-secure interrupt sources. It also allows the configuration of secure interrupts with a higher priority than the non-secure interrupts, and has several configuration models that enable the assignment of IRQs and FIQs to secure or non-secure interrupt sources.

LTZVisor configures interrupts of secure devices (i.e., secure interrupts) as FIQs, and interrupts of non-secure devices (i.e., non-secure interrupts) as IRQs. A TrustZone-enabled GIC permits all implemented interrupts to be individually defined as Secure or Non-secure, through the *Interrupt Security Registers* set (*ICDISRn*). To program secure interrupts to use the FIQ interrupt mechanism of the processor, the *FIQen* bit in the *CPU Interface Control Register* (*ICPICR*) must be set. When the secure guest OS (i.e., RTOS) is under execution, secure interrupts (i.e., FIQs) are redirected to the RTOS without hypervisor interference, guaranteeing that no overhead is added to the interrupt latency of the secure guest OS. This can be done by disabling the *FIQ* bit into the *Secure Configuration Register* (*SCR*). If an IRQ (i.e., an interrupt for the GPOS partition) arises while the RTOS is executing, it doesn't affect the expected RTOS behavior. As soon as the non-secure guest becomes active, the interrupt will be then processed. The prioritization of secure interrupts prevents a denial-of-service attack against the secure side (from the GPOS partition). From a different perspective, when the non-secure guest OS (i.e., GPOS) is executing and an FIQ (i.e., an interrupt for the RTOS partition) arises, the execution flow is immediately redirected to the hypervisor, which will be responsible for handling the interrupt directly in monitor mode.

This design decision minimizes the interrupt latency from the RTOS perspective, ensuring the interrupt is attended as soon as possible. On the other hand, if an IRQ arises, it will be directly managed by the non-secure guest. Non-secure interrupts are always signaled (by design) using the IRQ mechanism of the processor.

4.7 Time Management

Temporal isolation in virtualized systems is typically achieved using two levels of timing: at hypervisor level and at partition level. For the partition level, hypervisors typically provide timing services which allow guests to have notion of virtual or real time. In the first case, each time a partition is inactive, the time is paused, and once the guest is rescheduled, the timekeeping is resumed. In the second case, when the partition is paused, the hypervisor is responsible for keeping track of the wall-clock time, and, once resumed, update the partition timing structures.

LTZVisor provides a distinctive time management implementation. Due to its dual-OS configuration, as well as the intrinsic design principle of asymmetric scheduling, the hypervisor dedicates one independent timing unit for each guest OS. The secure VM uses the Triple Timer Counter (TTC) 0, while the non-secure VM uses the TTC1. It is fundamental that the hypervisor configures TTC1 as a non-secure device, otherwise an exception will be triggered on the first attempt to access it. This specific time management implementation ensures that each VM has its timing structures updated at all times. The RTOS does not miss any system-tick interrupt, and the GPOS, as a tickless OS, is completely aware of the real passage of time.

4.8 Inter-VM Communication

Inter-VM communication provides a transparent virtual mechanism for implementing communication between different VMs. In contrast with other solutions, which follow an non-standard approach [24, 22, 26], LTZVisor uses the standardized VirtIO [21] as a transport abstraction layer. VirtIO has been used in several implementations targeting I/O virtualization [21, 4], and has recently started being adopted to implement inter-guest [14] and inter-processor communication on multicore platforms (e.g., Texas Instrument RPMsg and Mentor Graphics MEMF) [2].

LTZVisor implements an adaptation of the RPMsg API from the Texas Instrument and OpenAMP group to a supervised single-core architecture. The implementation from Texas provides the foundation for implementing communication on top of the GPOS, while the implementation from OpenAMP provides the foundation for a bare-metal approach. The main modifications encompass: (i) the complete elimination of the Remoteproc executable loader and processor life cycle management, since it is supported by the hypervisor; (ii) the VirtIO device Remoteproc configuration implemented through a static approach (at compile-time); and (iii) RPMsg slave mode support following the VirtIO standard. Figure 3 depicts the communication architecture. As it can be seen, the data path is completely isolated by the event path, a design decision that promotes asynchronous communication, essential to guarantee the timing requirements of the secure VM. The data path is defined by a shared block of memory, configured as non-secure. The event path is defined by software generated interrupts (SGIs) routed through the hypervisor. This mechanism is based in requests from guest OSes to the hypervisor, via the SMC instruction. All requests are stored in a circular buffer, following a first-in, first-out policy. During each partition switch, LTZVisor triggers SGIs to the respective guest OSes, enabling asynchronous notifications. In spite of the

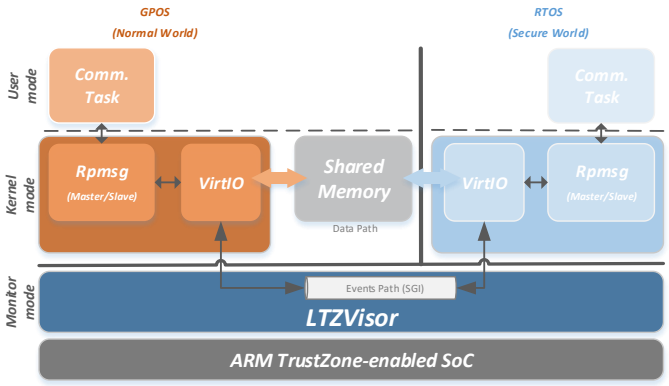


Figure 3 LTZVisor: Inter-VM Communication.

imposition of a slight increase to the partition-switching time, this trade-off guarantees the reliability of the communication as the hypervisor has control over every transaction.

5 Evaluation

LTZVisor was evaluated on a Xilinx ZC702 evaluation board targeting a dual ARM Cortex-A9 running at 667MHz. In spite of using a multicore hardware architecture, the evaluated implementation only supports a single-core configuration. Our evaluation focused on three metrics: (i) memory footprint, (ii) performance overhead and (iii) interrupt latency. LTZVisor and both OS partitions were compiled using the ARM GNU toolchain, with compilation optimizations disabled (-O0). The idea of presenting results with compilation optimizations disabled is because it represents the worst case scenario. Linaro Linux (v3.3.0) and FreeRTOS (v7.0.2) were used as non-secure and secure partitions, respectively. MMU, data and instruction cache and branch prediction were disabled on the secure world side.

5.1 Memory Footprint

In order to assess the memory footprint of each software component of the implemented architecture we used the size tool of the ARM GNU Toolchain. We evaluated LTZVisor, as well as the native, modified and virtualized version of FreeRTOS. Table 1 presents the collected measurements, where boot code, libraries and drivers were not taken into consideration. As it can be seen, the memory overhead introduced by the hypervisor is really small, i.e., 2880 bytes. The main reasons behind such a low memory footprint are related to the principle of minimal implementation followed during LTZVisor design which relies on (i) the hardware support of TrustZone technology for virtualization and (ii) the careful design and static configuration of each hypervisor component. The native version of FreeRTOS, supporting IRQ, requires 18882 bytes, the modified version, supporting FIQ, requires 18898, and the virtualized version requires 18918 bytes. From the native version to the modified one there is a slight increase of 0.8% in the memory footprint, while from the native version to the virtualized one there is an increase of 1.9%. This slight increase is completely acceptable and encompasses small modifications and adaptations for FIQ and context-switch handling (from native to modified), and in the FreeRTOS scheduler (from modified to virtualized).

Table 1 LTZVisor memory footprint (bytes).

Software	*Memory Footprint*			
	.text	*.data*	.bss	Total
LTZVisor	2368	0	512	2880
FreeRTOS IRQ (v7.0.2)	17942	20	920	18882
FreeRTOS FIQ (v7.0.2)	17954	20	924	18898
vFreeRTOS FIQ (v7.0.2)	17974	20	924	18918

5.2 Performance

The performance evaluation process was split into three different test case scenarios. First, LTZVisor was evaluated for specific micro-operations of the guest-switching operation. Then, we evaluated the virtualization overhead (using the Thread Metrics Suite) as well as the interrupt latency over the secure VM (RTOS). Finally, we assessed the virtualization overhead over the non-secure VM (GPOS) using the LMBench3 Suite.

5.2.1 Partition context switching

To evaluate the guest context switch time we used the Performance Monitor Unit (PMU) component. To measure the time consumed by each internal activity of a round-trip world switch, a PMU-specific instruction was added at the beginning and end of each code portion to be measured. Results were gathered in clock cycles and converted to microseconds accordingly to the processor's frequency (667MHz). The values represent the average and the standard deviation of 1000 collected samples.

The list of internal activities to perform a full switch between secure to non-secure and non-secure to secure worlds are:

1. *SMC handling.* The secure guest OS schedules the idle task. The idle task performs a secure call that is responsible for invoking the hypervisor (*SMC*). Time since the processor enters in the monitor's vector table until LTZVisor completes the SMC handling;
2. *Save secure guest OS context.* LTZvisor handles the SMC request and saves the context of the secure guest OS. Time to save the current state of the secure guest OS to its respective VMCB;
3. *Restore non-secure guest OS context.* LTZvisor saves the context of the secure guest OS and then restores the context of the non-secure guest OS. Time to restore the state of the non-secure guest OS from its respective VMCB;
4. *FIQ acknowledge.* The non-secure guest OS is running while a secure interrupt is triggered (e.g., RTOS timer tick). Time since the processor enters in the monitor's vector table until LTZVisor acknowledges the FIQ;
5. *Save non-secure guest OS context.* LTZvisor acknowledges the FIQ request and then saves the context of the non-secure guest OS. Time to save the current state of the non-secure guest OS to its respective VMCB;
6. *FIQ handling.* LTZvisor saves the context of the non-secure guest OS and then immediatly handles the FIQ request. Time since the hypervisor save the current state of the non-secure guest OS untill LTZVisor completes the FIQ handling;
7. *Restore secure guest OS context.* LTZvisor handles the FIQ and then restores the context of the secure guest OS. Time to restore the state of the secure guest OS from its respective VMCB.
8. *Scheduler.* LTZvisor restores the execution of the RTOS. The RTOS continues executing the idle task loop and verifies if there are real-time tasks to run. If not, the idle task performs a system call (*SMC*) that is responsible for invoking the hypervisor. Time since the processor restores the idle task execution until it enters in the monitor's vector table.

Table 2 LTZVisor performance statistics (clock cycles).

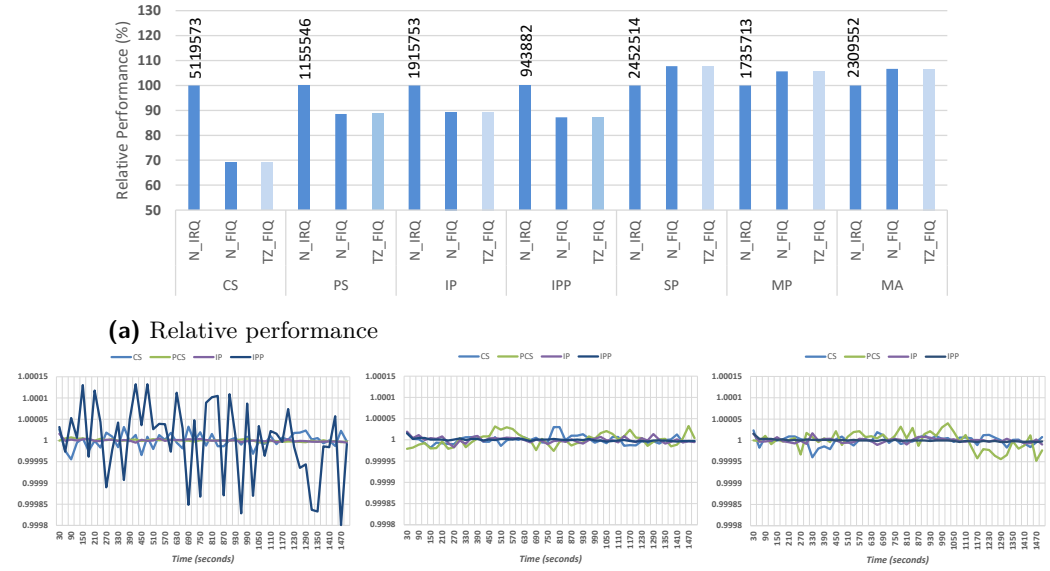
World Switch	*Operation*	*Performance*		*Time*
		$\overline{x}$	s	@667MHz
Switch to NS world	*(1) SMC handling*	571	0.943	$856ns$
	(2) Save S guest OS context	422	1.274	$633ns$
	(3) Restore NS guest OS context	949	2.324	$1423ns$
Switch to S world	*(4) FIQ acknowledge*	467	0.614	$700ns$
	(5) Save NS guest OS context	982	1.321	$1472ns$
	(6) FIQ Handling	1648	47.367	$2471ns$
	(7) Restore S guest OS context	243	0.524	$364ns$
Scheduler	*(8) Assymetric Policy*	7542	9.316	$11307ns$

Table 2 presents the collected results. As it can be seen, the complete partition-switch operation takes around 19.23 microseconds. This value assumes there are no real-time tasks ready to run once the RTOS is rescheduled. The process of verifying, from the RTOS execution, there are no real-time tasks to run and, hence, trigger the switch to the non-secure world takes around 11.31 microseconds. The process of switching from the RTOS to the GPOS takes just 2.91 microseconds, and is the most deterministic activity of the partition-switching operation. Our experiments demonstrated less than four clock cycles of deviation from the average value (for each individual activity). Once the GPOS is executing and a FIQ is triggered, the hypervisor ensures a 2.17 microseconds of interrupt latency, and then in a further 2.84 microseconds the RTOS is restored. The FIQ handling operation is the major source of non-determinism of the partition-switching operation. We strongly believe the reason is related with the required accesses to the peripheral bus when handling the interrupt request (in this specific case, the system tick timer).

5.2.2 Secure VM (RTOS)

The Thread-Metric Benchmark Suite consists of a set of benchmarks properly conceived to evaluate RTOSes performance. The suite comprises 7 benchmarks, evaluating the most common RTOS services and interrupt processing: cooperative scheduling (CS); preemptive scheduling (PS); interrupt processing (IP); interrupt preemption processing (IPP); synchronization processing (SP); message processing (MP); and memory allocation (MA). Each benchmark outputs a counter value, representing the RTOS impact on the running application: the higher the value, the smaller the impact.

Benchmarks were executed in the native version of FreeRTOS (N_IRQ), where interrupts are handled as IRQs, in a modified version of FreeRTOS, where interrupts are handled as FIQs (N_FIQ), and then compared against the virtualized version (TZ_FIQ). Figure 4a presents the achieved results, corresponding to the average relative performance (as well as the average absolute performance) of 1000 collected samples for each benchmark. Each sample reflects the benchmark score for a 30 seconds execution time, encompassing a total execution time of 500 minutes, per benchmark. In accordance with Figure 4a the execution of the modified version of FreeRTOS (N_FIQ) is very dependent from the benchmark. In some cases, the performance decreases, while in others the performance increase. The increase of performance on the modified version of FreeRTOS is completely understandable since FIQ interrupts present low hardware latency than IRQs, but the decrease is apparently strange. The reason behind this phenomenon is related with an adaption we did on the yield macro of FreeRTOS. The native version of FreeRTOS implements the yield through the use of the SVC exception. When a SVC is triggered a context-switch happens and the IRQ bit of the CPSR is set, so that there is no preemption during the execution of the critical routine



(a) Relative performance

(b) Variation: Native (IRQ) **(c)** Variation: Native (FIQ) **(d)** Variation: Virtualized (FIQ)

Figure 4 Thread-Metric benchmarks results.

(atomic execution). Thus, the modification of FreeRTOS for handling interrupts as FIQs should include the modification of the context-switch function to set the FIQ bit, instead of the IRQ bit. The problem is in the ARMv7-A specification, this bit is implementation defined. In the case of Xilinx Zynq, for security reasons, this bit is read-only, and only changes when triggered by hardware (e.g. when a FIQ happens). For this reason, we were forced to change the yield function to use an SGI (FIQ) instead of the SVC exception. The SGI has a higher latency than the SVC, which, on yield-intensive tests (i.e., the case of CS, PS, IP and IPP), this translates in a decrease of performance. It should be noted this is platform- and worlkload-specific problem that does not necessarily mean it can occur in other platforms and be noticeable in real application scenarios. In fact, the overhead introduced by LTZVisor is null, as demonstrated by the comparison of the N_FIQ and TZ_FIQ versions. This is perfectly understandable because, once FreeRTOS starts running real-time tasks, it will never be interrupted by the hypervisor. Regarding the variation, Figure 4b, Figure 4c and Figure 4d present the normalized variation of the collected results over time for the native, modified and virtualized versions of FreeRTOS, respectively. It is clear that the use of FIQ for handling interrupt sources slightly reduces the variation of results, and variation in the virtualized system is also in the same order of magnitude as the modified version, which means the virtualized system remains as deterministic as the (modified) native one. In sum, the asymmetric scheduling policy gives the RTOS a higher execution privilege, so it can preserve its real-time characteristics. Furthermore, the necessity of handling interrupts as FIQs promotes a deterministic execution, and most of the cases can either increase performance.

Interrupt latency is the measurement of system's response-time to an interrupt, which corresponds to the elapsed time between interrupt assertion and the instant when a response happens. Equation 1 expresses the system latency: τ_H is the hardware dependent time which depends on the interrupt controller, on the board, as well as the type of the interrupt; τ_{OS} is

the OS-specific induced overhead; and τ_{HYP} is the hypervisor-specific induced overhead.

$$\tau_{IL} = \tau_H + (\tau_{HYP}) + \tau_{OS} \quad (1)$$

Experiments showed that the latency in the native system (FreeRTOS) is 0.89 microseconds, which corresponds to the average interrupt handling overhead of our system, because when the secure guest OS is executing the FIQ requests are directly forward to the RTOS. The τ_{HYP} expression of Equation 1 represents the extra overhead induced by our approach, which only occurs when the RTOS has no real-time ready-to-run tasks, and consequently the hypervisor is invoked to perform a world switch. Since LTZVisor runs with all interrupt sources disabled, the worst case scenario happens when an FIQ request (e.g., RTOS tick) arrives while a context switch from the secure to the non-secure world is starting. In this case, the request is handled with a worst case interrupt latency of 5.08 microseconds. This is a very sporadic situation that can happen under rare conditions, because two asynchronous and independent events need to occur at the same time: (i) an asynchronous FIQ needs to be triggered while (ii) a world switch is happening. Nevertheless, since the overhead introduced on latency has a deterministic upper bound (5.08 microseconds), it can be taken into account when designing the real-time system.

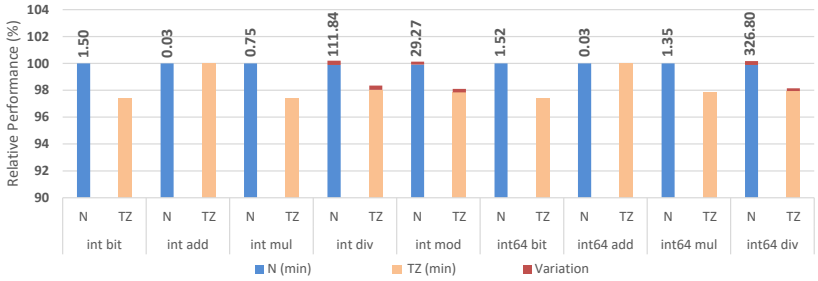
5.2.3 Non-Secure VM (GPOS)

LMBench [11] is a widely used suite of micro-benchmarks that measure a variety of important aspects of system performance, such as latency and bandwidth. The suite is written in portable ANSI-C using POSIX interfaces and targeting UNIX systems. The LMBench 3.0 suite includes more than fourthy micro-benchmarks within three different categories: *bandwith*, *latency*, and *other*. We focus our evaluation on two specific benchmarks, evaluating different architectural subsystems:

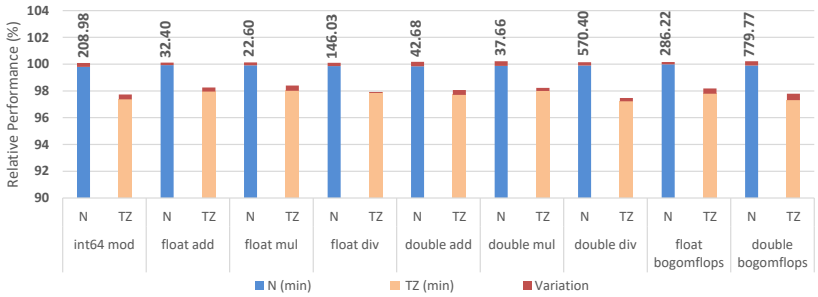
- *lat_ops:* Arithmetic operations latency, to evaluate general CPU performance (VFP and Neon are disabled);
- *bw_mem:* Memory operations bandwidth for different blocks size (2K, 128K, 4M), to evaluate the interference of the TZASC as well as Level 1 (4-way set-associative 32 KB) and Level 2 (8-way set-associative 512 KB) data caches.

For the first part of the experiment, FreeRTOS was configured with a 1 millisecond tick rate (i.e., guest-switching rate) and no real-time tasks were added to the system (i.e., the RTOS will be infinitely executing the idle task). We ran the micro-benchmarks in the native version of Linux (N) and compared them against the virtualized version (TZ). L1 and L2 caches and branch prediction were enabled for both test case scenarios. For each micro-benchmark, we performed 100 consecutive experiments. For each experiment the micro-benchmark was configured for 10 warm-ups and 1000 repetitions (–W 10 –N 1000). Presented results correspond to the average relative performance and variation (as well as the average absolute performance) of the 100 consecutive experiments, encompassing a total of 100000 samples (per bar).

Figure 5 presents the achieved results for the arithmetic operations latency benchmark. The values on top of the bars corresponds to the average latency, in nanoseconds. As it can be seen, the virtualized version of Linux only presents an average performance degradation of 2%, when compared with its native execution. This value is practically uniform among all micro-benchmarks (apart from the small variations due to the benchmark's lack of accuracy and the system's nonlinearities), except for the `int add` and `int64 add` cases. For these specific micro-benchmarks, the achieved results do not reflect the real performance penalty,



(a) *lat_ops* benchmark results (part 1)

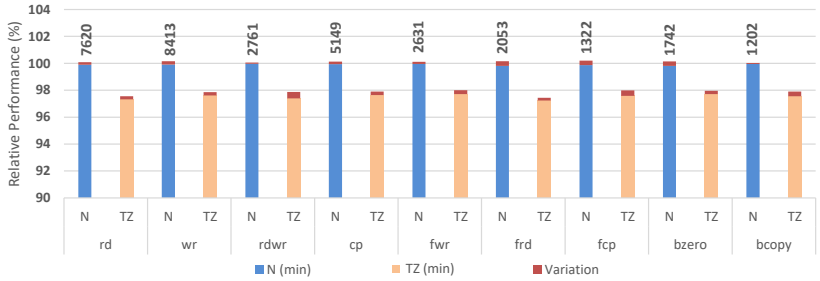


(b) *lat_ops* benchmark results (part 2)

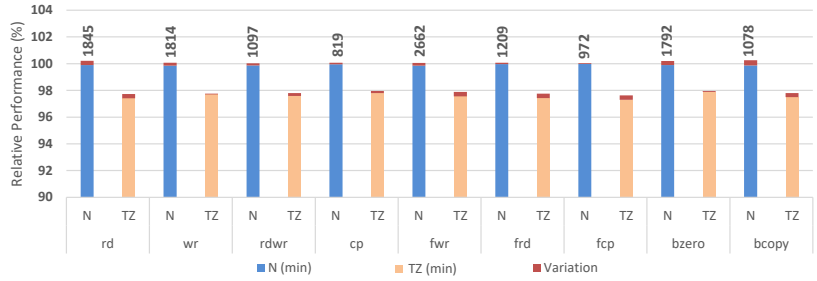
Figure 5 LMBench arithmetic operations latency (*lat_ops*) benchmark results.

due to the lack of precision. The assessed latency is 0.03 nanoseconds, and the minimal time unit is 0.01 nanoseconds. Regarding variation, it is clear the virtualized Linux presents a variation in the same order of magnitude as the native version. This means the virtualized system remains as deterministic as the native one.

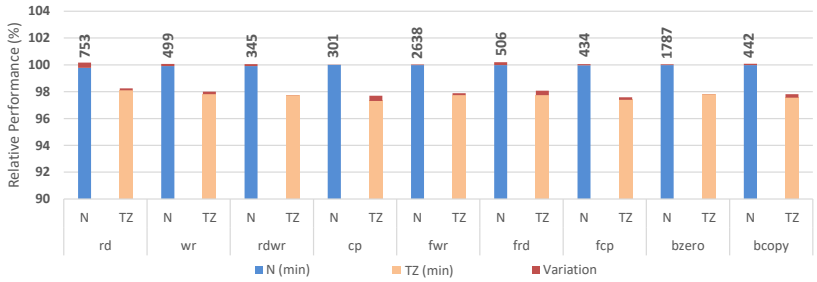
Figure 6 presents the achieved results for the memory bandwith benchmark. The values on top of the bars corresponds to the average memory bandwith, in megabytes per second (MB/s). Figure 6a, Figure 6b and Figure 6c depict the assessed results for a memory block size of 2KB, 128KB and 4MB, respectively. These memory block sizes were selected with the intention to fit and not within the L1 and L2 cache sizes. Looking at the three figures, it is clear the relative performance of the system is practically uniform among all micro-benchmarks, presenting an average performance degradation of 2% when comparing the virtualized Linux to the native one. Contrasting these values with the results presented in Figure 5, three main conclusions can be drawn: first, it is clearly noticed the effect of each cache on the accessed absolute memory bandwith results – the higher is the memory block size, the lower is the memory bandwith; second, cache isolation is in fact guaranteed by hardware, and does not introduce any extra overhead neither requires any cache maintenance operation on each guest-switch; and, finally, (memory) space isolation provided by means of the TZASC does not have associated any extra source of overhead. To corroborate the viability of our conclusions, experiments were performed without some of the hypervisor support (please refer to Sections 4.1 and 4.4) for caches and memory initialization. For example, one set of experiments were performed without the hypervisor having enabled L2 cache before booting the GPOS. The results were very straightforward: an abrupt decrease of performance, reaching almost 70% in some cases, happen for memory block sizes higher than 32KB and lower than 512KB. These experiments clearly demonstrates the effect of L2 cache in the overall system, as well as the coexistence of non-secure and secure cache entries without any cache maintenance support. Despite not being presented in Figure 6, due to



(a) *bw_mem* benchmark results (2KB)



(b) *bw_mem* benchmark results (128KB)



(c) *bw_mem* benchmark results (4MB)

Figure 6 LMBench memory bandwith (*bw_mem*) benchmark results.

shortage of space, we also performed a larger set of experiments encompassing memory block sizes of 16KB, 64KB and 1MB. The achieved results were identical to the ones presented in Figure 6, which reinforces the reliability of our conclusions.

For the second part of the experiment, instead of fixing the FreeRTOS tick with a 1 millisecond rate, the same experiments were repeated for eight different guest-switching rates within a time window between 100 microseconds to 10 milliseconds. Again, no real-time tasks were added to the system. We ran the arithmetic operations latency benchmark in the virtualized version of Linux. L1 and L2 caches and branch prediction were enabled for all test case scenarios. For each micro-benchmark we performed 100 consecutive experiments, and for each experiment the micro-benchmark was configured for 10 warm-ups and 1000 repetitions (–W 10 –N 1000). Presented results correspond to the average performance overhead of measured results for the 18 (arithmetic) micro-benchmarks, encompassing a total of 1800000 samples per test case scenario (per mark). Eight different test case scenarios were setup, corresponding to a tick rate of 100, 200, 300, 500, 1000, 2000, 5000 and 10000 microseconds, encompassing a cumulative number of 14400000 samples. Figure 7 presents the achieved results. The performance overhead ranges from 22.93% to 0.65% for a guest-switching rate of 100 and 10000 microseconds, respectively. For a system configured with a tick rate above 500 microseconds, the expected performance overhead is less than 5%.

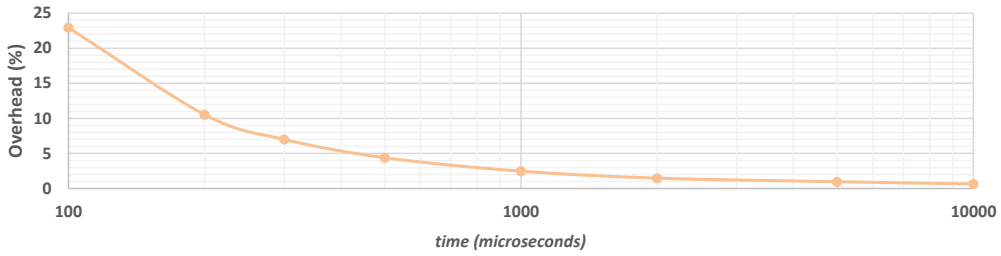


Figure 7 LTZVisor: guest-switching rate vs performance overhead.

6 Discussion

With LTZVisor we demonstrated how hardware enhancements introduced by TrustZone technology can be adequately exploited to assist virtualization, especially in the case of two virtual machines, because this number coincides exactly with the number of isolated states directly supported by the processor. We demonstrated and explained how several TrustZone features can be adequately exploited to run an RTOS side-by-side with a GPOS.

The asymmetric design principle, which dictates the secure VM has a greater scheduling priority than the non-secure one, ensures the timing requirements of the real-time environment remains nearly intact, at the cost of integrating the hypervisor with the RTOS on the secure world side. In doing so, the RTOS has full control over the system, and can access or modify the state of the non-secure VM. Recently, Ngabonziza et al. [12] presented some doubts about how our solution [17] could prevent the RTOS (secure world) from accessing the GPOS (non-secure world): in fact, it cannot; this is the price we pay to preserve the real-time demands of the system, while keeping performance acceptable for low-end and low-cost devices. Anyway, two possible solutions to guarantee a higher degree of isolation on high-end devices are: run all guest OSes in the non-secure world side, as demonstrated by our recent work [18]; or either paravirtualizing the RTOS, so that it can run in the user mode of the secure world side, and mediate each memory access through the hypervisor. Another point outlined by Ngabonziza et al. is related to guest OSes preemption and consequent starvation. They argue in our design "either OS cannot preempt the other OS". This is wrong; LTZVisor guarantees, by design, the secure guest OS (RTOS) preempts the non-secure guest OS (GPOS) as soon as a FIQ is triggered, but the reverse is not possible. So, starvation can happen, but only from the non-secure world side. However, despite this being a design decision to ensure the real-time needs, it is well-justified by the fact typical real-time applications have frequent idle times, which ensures the non-secure guest OS has enough CPU slices for execution. Ultimately, the scheduling policy can be designed accordingly to the applications needs, ensuring enough scheduling points that adequately meet the needs of both OSes, without compromising any real-time deadline; or either multicore platforms can be exploited to implement asymmetric multiprocessing (AMP) support, as we already did.

One of the main identified limitations is related to the number of supported virtual machines. LTZVisor supports the coexistence of two VMs, one running in the secure world and one running in the non-secure world. Although this is almost sufficient for a huge amount of current embedded real-time applications, some researchers still rely on this premise to consider TrustZone as an ill-guided virtualization technique. We demonstrated this is not completely true, and that is possible to overcome this limitation by multiplexing more guest OSes inside the non-secure world side [18, 19]. It requires careful handling of shared hardware resources, such as processor registers, memory, caches and MMU. Processor registers can

be easily saved and restored into/from a specific VMCB, while memory isolation can be achieved through the dynamic memory configuration feature of TZASC.

Spatial isolation is a major requirement for virtualization. LTZvisor implements memory isolation relying on the TZASC, which is an optional and implementation-specific component on TrustZone specification. The granularity of access restrictions depends on the SoC. Some outdated TrustZone-based SoCs are not equipped with this memory controller, and on many other the TZASC can only control some portions of the memory. For example, the Versatile Express platform provides no means to partition the DDR RAM into secure and non-secure areas. Nevertheless, when regarding the most modern TrustZone-based SoC, this is completely different, because they are totally equipped with fully featured TrustZone-aware memory controllers. This is the case of Xilinx Zynq SoCs and also the Freescale i.MX53 QSB. For example, Sun et al. [25] explains the use of the same functionality to create TrustICE, a framework that uses the hardware-assisted Watermark technique to dynamically protect the memory regions of the suspended secure code (ICEs).

Another identified limitation on the memory subsystem is related to non-existence of a second level memory translation. There is no way to virtualize the physical memory as used by the guest OSes. The guest-physical memory always corresponds to the host-physical memory, which means all guest OSes have to co-operate with respect to the address space being used, requiring relocation and consequent recompilation of the guest OS. This means the chance to use multiple closed-source guest OSes (only available as binary image) is very reduced, because different OS providers typically compile their software to run on the same memory address space of a specific platform. What is seen as a limitation to the system from a non-real-time perspective, is somewhat seen as an advantage from a real-time perspective. It is well-established the use of MMU and other components which introduce some non-linearities are seen with some scepticism regarding determinism and worst-case performance requirements of many real-time systems. An important argument that supports our vision is the recent decision of ARM in introducing support for virtualization in the new ARMv8-R architecture relying on a double-stage MPU [27]. In the ARMv8-R architecture, operating systems running at PL1 (IRQ, FIQ, SVC, System, etc) are able to use an MPU, as well as the hypervisor running at PL2 (Hypervisor). The MPU controlled by the hypervisor restricts access of memory regions or peripherals to an individual guest, or shared between guests. This is a similar strategy to the one we use with TrustZone, and was adopted by ARM to meet the strict requirements of real-time environments.

The existence of two distinct MMU interfaces as well as secure and non-secure cache entries is also seen as an advantage due to the performance gains achieved during the partitions-switch. From a real-time perspective, the use of these features is not always desirable, which means that in many potential embedded applications the use of MMU and cache will only be exploited by the non-secure guest OS. However, if the idea is to consolidate a soft-real-time system with a general purpose, the use of these features can be helpful in terms of context-switch time and performance. The only disadvantage that arises with the TrustZone-awareness in this components, is the need of minimal hypervisor support on their initialization, as well as their inaccessibility during runtime. In this case, one possible strategy to deal with this limitation is to implement some paravirtualization support, by statically analyzing the non-secure guest OS image file, identify the opcode of the instruction, and replace the instruction by hypercalls that request the access to those components mediated by the hypervisor.

Current device virtualization approach goes towards a pass-through model without any sharing device access support. Device isolation relies in a virtual form of IOMMU provided

by means of the TZPC. Similar to the limitation identified in the TZASC, the TZPC is also an optional and implementation-specific component on TrustZone specification. This means the number and type of devices that can be configured as non-secure vary from platform to platform and from vendor to vendor. For example, in Xilinx ZC702, the TTC0 is always secure and there is no way to configure its access directly form the non-secure guest OS. Despite the identified limitation on the TZPC, the pass-through policy without any support for shared devices is also somehow limiting. This kind of implementation makes sense in the case of the secure VM, to promote real-time characteristics, but is very limiting in a system there is a need to share devices among VMs and disregards one of our main design principles: the principle of least privilege. We plan to implement an hybrid approach in-between a pass-through and a paravirtualization strategy: the secure guest OS has direct and full control over the devices (pass-through model), but the non-secure VM requests access to devices via hypercalls, and the hypervisor mediates the access (paravirtualization). This model guarantees the timing requirements of the real-time environment, promotes the principle of less privilege by controlling the non-secure guest OS devices' access while overcoming the dependency of the TZPC for configuring devices as non-secure.

One of the main advantages of TrustZone resides on the interrupt subsystem. The direct assignment of interrupts to each world, without intervention of the hypervisor, is a plus, but, most importantly, it does not increase the interrupt latency of the secure world once the RTOS gets executed. One small disadvantage that comes with this model is that slight modifications need to be introduced in the secure guest OS, in order to use interrupt handlers as FIQs instead of IRQs. In doing so, another problem on this specific platform arises: the decrease of performance on yield-intensive workloads. However, since this problem is very specific to this platform and precise workloads, we believe it should not be generalized.

Last, but not least, another considerable advantage of the presented solution is its scalability. The recent ARM decision of introducing TrustZone technology in the new Cortex-M processors series opens up a number of opportunities for implementing cost-effective virtualization for future low-end real-time systems. We strongly believe that it will be possible to consolidate a hard real-time environment (as a secure guest OS) with a soft real-time environment (as a non-secure guest OS), at the cost of minimal engineering effort. The enhancements introduced in TrustZone specification for ARMv8-M architecture will definitively ensure better timing and performance guarantees, since the new specification implements more hardware support for world switching while guaranteeing faster transitions and greater power efficiency.

7 Related Work

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 work presented by Johannes Winter [30] was the first scientific public attempt to exploit the TrustZone technology to assist virtualization. The paper introduces 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. [3] 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 on the same processor. The evaluation process was conducted only on an emulator, and presenting limited results regarding the virtualization overhead and

the hypervisor interference in the real-time characteristics. In [5] Frenzel et al. presented a minimal adapted version of Linux-kernel (as normal world OS) on top of a hypervisor running on the secure world side. SafeG [22], from TOPPERS Project, 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. ViMoExpress [13] is a lightweight virtualization solution, proposed by Oh et al., which exploits the TrustZone technology to accelerate the execution of two guest OSes. Both works do not conducted any evaluation neither reported any experiments. Schwarz et al. [23] proposed an alternative system virtualization approach based on TrustZone which allows the switch between a virtualized and non-virtualized execution mode through soft reboots.

8 Conclusion

Embedded real-time systems are proliferating at rapid pace in our everyday life, representing a huge part of our key infrastructures. The trend nowadays goes towards the consolidation of a wide range of functions into the same hardware platform, leading real-time requirements to coexist with non-real-time characteristics. Virtualization has been used as an enabler for platform consolidation whilst guaranteeing a robust functionality isolation, but the penalties incurred by existent software-based approaches, altogether with timing requirements imposed by real-time virtualization bring forth the need of hardware-assisted virtualization solutions. Among existing COTS technologies, ARM TrustZone is attracting particular attention, due to its exclusive applicability on those ARM processors where VE are not available, while offering the best cost-benefit trade-off. The problem is that this technology is still seen with a lot of scepticism, which rised an urgent need to comprehensively examine the hype, myths, and realities of the use of this technology for virtualization, especially because ARM continues to spread TrustZone across the different processors families. LTZVisor provides a tool to understand, evaluate and discuss the benefits and limitations of using this security-oriented technology to assist virtualization. We conducted an extensive set of experiments which demonstrated that this technology can effectively satisfy the strict requirements for virtualizing a real-time environment, while offering a low performance cost on running an unmodified guest GPOS. Evaluation over the non-secure guest OS also helped us understand the expected penalties over a soft-real-time guest OS, regarding the consolidation of a soft and hard real-time environment in future Cortex-M processors. With LTZVisor we want to share the experience gained over the last years, to encourage research for next generation of TrustZone-assisted virtualization solutions.

Current research aims at multicore extension. We have already implemented support for asymmetric multiprocessing (AMP), but we also want to explore other multicore configurations. Work in the near future will focus on an extensive and exhaustive evaluation of real-time aspects with short-term and long-term tests, as well as studying the timing interferences and sources of non-determinism which arise from the multicore approach. Extension of LTZVisor for new generation Cortex-M platforms is also at the top of our goals, but we still need to wait until the release of the first ARMv8-M boards.

Acknowledgements. We would also like to thank the anonymous reviewers for their helpful comments on the first version of this paper.

References

1 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. doi:10.1145/1165389.945462.

2 F. Baum and A. Raghuraman. Making Full use of Emerging ARM-based Heterogeneous Multicore SoCs. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems*, Jan 2016.

3 M. Cereia and I. Bertolotti. Virtual Machines for Distributed Real-time Systems. *Comput. Stand. Interfaces*, 31(1):30–39, January 2009. doi:10.1016/j.csi.2007.10.010.

4 C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. *SIGPLAN Not.*, 49(4):333–348, February 2014. doi:10.1145/2644865.2541946.

5 T. Frenzel, A. Lackorzynski, A. Warg H., and Härtig. ARM TrustZone as a Virtualization Technique in Embedded Systems. *Twelfth Real-Time Linux Workshop*, 2010.

6 G. Heiser. Virtualizing Embedded Systems: Why Bother? In *Proceedings of the 48th Design Automation Conference*, DAC'11, pages 901–905. ACM, 2011.

7 H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, and H. W. Jin. Full virtualizing micro hypervisor for spacecraft flight computer. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 6C5–1–6C5–9, Oct 2012. doi:10.1109/DASC.2012.6382393.

8 Genode Labs. An Exploration of ARM TrustZone Technology. URL: https://genode.org/documentation/articles/trustzone.

9 C. Lee, S. W. Kim, and C. Yoo. VADI: GPU Virtualization for an Automotive Platform. *IEEE Transactions on Industrial Informatics*, 12(1):277–290, Feb 2016. doi:10.1109/TII.2015.2509441.

10 Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J. Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.

11 L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

12 B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. TrustZone Explained: Architectural Features and Use Cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, Nov 2016. doi:10.1109/CIC.2016.065.

13 S. Oh, K. Koh, C. Kim, K. Kim, and S. Kim. Acceleration of dual OS virtualization in embedded systems. In *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, pages 1098–1101, Dec 2012.

14 S. Patni, J. George, P. Lahoti, and J. Abraham. A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial. In *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pages 6–9, Sept 2015. doi:10.1109/NGCT.2015.7375072.

15 S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares. IIoTEED: an enhanced Trusted Execution Environment for Industrial IoT Edge Devices. *IEEE Internet Computing*, 21(1):40–47, Jan-Feb 2017. doi:10.1109/MIC.2017.17.

16 S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares. FreeTEE: When real-time and security meet. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, Sept 2015. doi:10.1109/ETFA.2015.7301571.

17 S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares. Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–4, Sept 2014. doi:10.1109/ETFA.2014.7005255.

18 S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares. Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems. *IEEE Computer Architecture Letters*, PP(99):1–1, 2016. doi:10.1109/LCA.2016.2617308.

19 S. Pinto, A. Tavares, and S. Montenegro. Space and time partitioning with hardware support for space applications. *Data Systems In Aerospace (DASIA), European Space Agency, (Special Publication) ESA SP*, 2016.

20 D. Reinhardt and G. Morgan. An embedded hypervisor for safety-relevant automotive E/E-systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 189–198, June 2014. doi:10.1109/SIES.2014.6871203.

21 Rusty Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. doi:10.1145/1400097.1400108.

22 D. Sangorrin, S. Honda, and H. Takada. 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, Brussels, Belgium*, pages 6–15, 2010.

23 O. Schwarz, C. Gehrmann, and V. Do. Affordable Separation on Embedded Platforms. In *Proceedings of the 7th International Conference on Trust and Trustworthy Computing*, volume 8564 of *LNCS*, pages 37–54. Springer-Verlag New York, Inc., 2014. doi:10.1007/978-3-319-08593-7_3.

24 Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys'10, pages 209–222. ACM, 2010. doi:10.1145/1755913.1755935.

25 H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 367–378, June 2015. doi:10.1109/DSN.2015.11.

26 A. Tavares, A. Dídimo, T. Lobo, P. Cardoso, J. Cabral, and S. Montenegro. Rodosvisor – An ARINC 653 quasi-compliant hypervisor: CPU, memory and I/O virtualization. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, pages 1–10, Sept 2012. doi:10.1109/ETFA.2012.6489588.

27 J. Taylor. Security for the next generation of safe real-time systems. In *Proceedings of Embedded World Conference, Nuremberg, Germany*, March 2016.

28 Prashant Varanasi and Gernot Heiser. Hardware-supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys'11, pages 11:1–11:5. ACM, 2011. doi:10.1145/2103799.2103813.

29 P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing Embedded Security on Dual-Virtual-CPU Systems. *IEEE Design Test of Computers*, 24(6):582–591, Nov 2007. doi:10.1109/MDT.2007.196.

30 J. Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC'08, pages 21–30. ACM, 2008. doi:10.1145/1456455.1456460.

31 S. Zampiva, C. Moratelli, and F. Hessel. A hypervisor approach with real-time support to the MIPS M5150 processor. In *Sixteenth International Symposium on Quality Electronic Design*, pages 495–501, March 2015. doi:10.1109/ISQED.2015.7085475.