2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)

Virtualization on TrustZone-enabled Microcontrollers? Voilà!

Sandro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, Adriano Tavares
Centro ALGORITMI - Universidade do Minho
{sandro.pinto, hugo.araujo, daniel.oliveira, jose.martins, atavares}@dei.uminho.pt

***Abstract*—With predictions pointing to more than 20 billion Internet-enabled 'things' by 2020 and much more to come, smart sensor nodes are expected to be predominant in the Internet of Things (IoT) era. As these systems are connected to the Internet and tend to implement an ever-growing number of mixed-criticality features, there is huge pressure for strong isolation to guarantee a reliable, secure, and predictable infrastructure. While virtualization has been a game-changer for consolidation and isolation in mid- to high-end embedded applications, for low-end and low-cost systems it is still in its infancy, and only a limited number of solutions have been proposed so far. This work aims at developing a lightweight hypervisor which provides strong isolation on resource-constrained devices. Our approach leverages TrustZone technology available on modern Arm microcontrollers (TrustZone-M) to implement a predictable virtualization infrastructure for low-end and low-cost systems. Experiments conducted on an Arm Musca-A multi-core platform demonstrate our solution achieves low memory footprint, high efficiency, and strict timing predictability.**

***Index Terms*—Virtualization, TrustZone, Mixed-criticality, Isolation, Microcontrollers, Multi-core, Real-time, Arm.**

I. Introduction

With the advent of the Internet of Things (IoT), we are witnessing a massive adoption of Internet-enabled 'things' [1], [2]. Gartner predicts that by 2020 there will be over 20 billion connected devices [3], and Arm expects that a trillion new IoT devices will be produced between 2017 and 2035 [2]. This massive global network infrastructure represents a collection of billions of smart, connected devices [4]. As these systems are connected to the Internet, they are inherently exposed to an endless number of security threats. Furthermore, as they tend to consolidate multiple applications/environments originating from different developers and aim at implementing an ever-growing number of mixed-criticality features, there is huge pressure for strong isolation to guarantee a reliable, secure, and predictable infrastructure [4], [5].

In resource-constrained devices, the partitioning of mixed-criticality systems has been mainly implemented through federated architectures [6]. This means there is a physical separation of several subsystems spanned across different microcontrollers (MCUs). However, since the number of functions is increasing at a rapid pace and this trend is expected to continue growing in the near future, federated architectures are becoming impractical, due to size, weight, power, and cost (SWaP-C) requirements [6], [7]. In such a scenario, widespread virtualization technologies already used in high-end embedded computing systems [8]–[10] also become paramount in low-end and low-cost systems. Use-cases for MCU-based virtualization range from the isolation of security- and safety-critical functions from those that require less stringent control (e.g., small Industrial IoT controllers), to the consolidation of several applications onto fewer electronic control units (ECUs) to manage complexity and reduce cost. So, the increasing demand for MCU-based virtualization is leading academia [6], [11], [12] and industry [13] to put significant effort into the development of lightweight virtualization solutions.

Arm TrustZone is a hardware security-oriented technology introduced into Cortex-A processors back in 2004 [14]. This technology is centered around the concept of separating the system execution into the secure and normal worlds. The ubiquitous adoption of Arm-based processors in the embedded market has made TrustZone as one of the most used key-enabling technologies for enforcing trusted execution environments (TEE) in mobile devices [14], [15]. The research community has also been extremely active in exploring new ways to leverage TrustZone for isolation, including the use of TrustZone to implement an alternative and lightweight form of system virtualization [16], [17].

TrustZone for MCUs (a.k.a. TrustZone-M) is a new endeavor [18]. At a high-level, this variant is similar to the variant implemented in Cortex-A processors; however, at a low-level, there are significant architectural differences since the underlying mechanisms were re-designed from the ground up and optimized for low-power applications. Although TrustZone-M is mainly targeting use-cases for root of trust implementation, security management, and firmware protection, it is our belief that since TrustZone has enabled an alternative form of system virtualization in high-end devices, TrustZone-M will be a game-changer for low-end virtualization. However, as of this writing, existing TrustZone-assisted hypervisors [16], [17], [19] have no support for Armv8-M.

TrustZone-M is available in the new generation of Cortex-M MCUs, i.e. the Cortex-M23 and the Cortex-M33. Among existing TrustZone-M platforms, Arm Musca-A board is particularly interesting, due to the implementation of multi-core technology in MCUs. Computing systems are progressively moving towards multi-core platforms due to the proven advantages in terms of computing power and power consumption. This trend is expected to continue in the near future, and multi-core platforms are expected to become mainstream even on low-end and low-cost systems. Although multi-core technology brings several advantages, it also raises several

978-1-7281-0678-6/19/$31.00 ©2019 IEEE
DOI 10.1109/RTAS.2019.00032

challenges and difficulties deriving from the reciprocal interference caused by hardware resource sharing (e.g., memory controllers, caches, buses) [5], [20]. This issue is particularly relevant when the platform is deploying applications with different criticalities. The real-time community has been proposing several approaches to minimize contention and improve predictability at the OS [21], [22] and hypervisor level [7], [23], [24]. Nevertheless, to the best of our knowledge, existing approaches focus on high-end platforms and depend on hardware features (e.g., two-stage MMU and performance counters) which are not typically available on MCUs.

Using TrustZone-M to implement virtualization is not straightforward; there are a set of unsolved key challenges which arise from the fact that the technology was redesigned from the ground up, requiring a considerable effort in software architectural redesign and implementation. Moreover, correct partitioning of micro-architectural shared resources is neglected by existing TrustZone-assisted hypervisor solutions. In this paper, we begin by discussing the background on TrustZone technology and TrustZone-assisted virtualization, and then we focus on the paper's contributions:

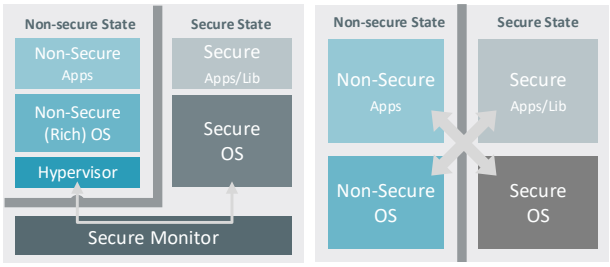
- The implementation of a TrustZone-M-assisted virtualization infrastructure that provides strong isolation for lightweight mixed-criticality systems (Section III). To the best of our knowledge, this is the first hypervisor supporting TrustZone-M technhology (Armv8-M).
- An analysis of a modern TrustZone-M-enabled multi-core platform considering the main sources of unpredictability and contention and a set of guidelines and design decisions to improve determinism and predictability (Section IV). To the best of our knowledge, none existing TrustZone system has addressed it so far.
- Finally, an evaluation of the virtualization infrastructure according to different dimensions and metrics, and focusing on determinism and predictability (Section V).

II. Background

In this section, we start by overviewing TrustZone and TrustZone-M technologies (Section II-A). Then, we explain the concept of TrustZone-assisted virtualization, and, finally, we highlight the challenges for implementing a TrustZone-M-assisted hypervisor (Section II-B).

A. Arm TrustZone

Arm TrustZone provides a system-wide hardware approach to security. TrustZone was firstly introduced into Arm application processors (Cortex-A) in 2004, and, recently, Arm released TrustZone-M for the new generation of Arm MCUs (Cortex-M) [14], [18] (Fig. 1). This technology is centered around the concept of two hardware-enforced protection domains. Each world is granted uneven privileges, with non-secure software prevented from directly accessing secure world resources. On Cortex-A processors, the current world in which the processor runs is determined by the Non-Secure (NS) bit. Furthermore, an extra privileged mode, named secure monitor, is added to implement mechanisms to perform



(a) TrustZone for Cortex-A (b) TrustZone for Cortex-M

Fig. 1: TrustZone technology

a secure context switch between worlds. To enter monitor mode there is a specific privileged instruction named Secure Monitor Call (SMC). TrustZone allows system designers to add a TrustZone Address Space Controller (TZASC) and a TrustZone Protection Controller (TZPC), which allow memory and devices to be configured as either secure or non-secure. Isolation is also implemented at the cache-level. The Generic Interrupt Controller (GIC) also supports the coexistence of secure and non-secure interrupt sources.

TrustZone for MCUs (TrustZone-M). From a high-level perspective, TrustZone for Armv8-M is similar to the variant in Cortex-A processors. In both designs, the processor can execute in either a secure or non-secure state. There are, however, important differences as Cortex-M has been optimized for faster context switch and deterministic execution. As a result, the underlying mechanisms of TrustZone-M are significantly different from the original TrustZone specification. In TrustZone-M the execution state is memory map based and world crossings take place automatically in exception handling code and special branching instructions (see Fig. 1). These security states are orthogonal to the existing processor modes, i.e., there is a Thread and Handler mode in both secure and non-secure states. TrustZone-M excludes the monitor mode and the need for any secure monitor software. This reduces the world switch latency, resulting in more efficient transitions. For bridging software between both worlds, TrustZone-M supports multiple secure function entry points. For this purpose, the instruction set architecture (ISA) was extended with three new instructions, including the secure gateway (SG). With the exception of stack pointers and a few special registers, in the Armv8-M architecture, most of the registers are shared between secure and non-secure states. Regarding the memory infrastructure, the physical address space is partitioned into secure and non-secure sections. Additionally, in TrustZone-M, the secure memory space is further divided into two types: secure and non-secure callable (NSC). NSC is a special secure memory location which is used to hold SG instructions; this is the entry point of every explicit transition between non-secure and secure states. The security state of memory can be configured using the Secure Attribution Unit (SAU) or the Implementation Defined Attribution Unit (IDAU). System designers can use the IDAU to define a fixed memory map

and the SAU to override the security attributes for some parts of the memory. The memory partitioning is also used to configure peripherals as secure or non-secure. The TrustZone-aware Memory Protection Unit (MPU) enables each world to have a local set of memory access permissions by providing a different MPU interface per world. The Nested Vectored Interrupt Controller (NVIC) allows interrupts to be configured as secure or non-secure. There are no restrictions regarding whether a non-secure or secure interrupt can take place when the processor is running non-secure or secure code. If the arriving interrupt's state is equal to the current execution state, the exception sequence is similar to the previous M-series processors. The main difference occurs when a non-secure interrupt takes place and is handled by the processor during the execution of secure code. In this case, the processor automatically pushes all secure information onto the secure stack and erases the contents from the register banks.

B. TrustZone-assisted virtualization

TrustZone technology enables a specialized, hardware-assisted, form of system virtualization. With a virtual hardware support for dual world execution, an extra processor mode (i.e., the monitor mode), and other TrustZone features like memory segmentation, it is possible to provide time and spatial isolation between execution environments [16], [17], [25]. Basically, the non-secure software runs inside a virtual machine (VM) whose resources are completely managed and controlled by a hypervisor running in the secure world. TrustZone-assisted virtualization is not particularly considered full-virtualization neither paravirtualization, because, although guest OSes can run without modifications on the non-secure world side, they need to interoperate to manage the usage of memory map and address space. TrustZone-assisted virtualization solutions support mainly two types of system configurations: dual-guest [16], [17] and multi-guest [26]. For instance, in a dual-guest configuration, the hypervisor runs in the monitor mode, and the secure and non-secure guest OSes run in supervisor mode of the secure and non-secure states, respectively. The VM running in the secure world is considered privileged because in this world there is no isolation between both supervisor and monitor modes. Among existing system configurations, the majority of TrustZone-assisted hypervisors follow a dual-OS approach, due to the perfect match between the number of VMs and the number of protection domains directly supported by the processor.

Challenges shifting to TrustZone-M-assisted virtualization. Given the previously detailed differences between TrustZone on Cortex-A and Cortex-M platforms, the existing TrustZone-assisted hypervisors are not directly amenable to modern Cortex-M processors. To make TrustZone-M-assisted virtualization a reality, several key challenges need to be addressed:

- TrustZone technology for Armv8-M excludes the NS bit and the privileged monitor mode; this requires a significant re-design in the overall architecture since the monitor mode is the CPU mode used for running the hypervisor component of existing TrustZone-assisted virtualization infrastructures.
- The ISA of TrustZone-M-enabled MCUs excludes the SMC instruction; this requires the implementation of a different mechanism to explicitly trigger transitions between VMs.
- The TrustZone-M specification does not include a TZASC nor a TZPC: this requires the implementation of a different memory and device manager for correctly partitioning memory and peripheral resources according to the new security controllers (SAU and IDAU).
- The TrustZone-enabled NVIC does not provide FIQ interrupts: this requires the implementation of a different interrupt management mechanism for managing and handling secure and non-secure interrupts.

III. TrustZone-M hypervisor

The TrustZone-M hypervisor was implemented targeting a dual-OS configuration. Comparing to existing classic TrustZone-assisted solutions (e.g., SafeG [16] and LTZVisor [17]), the main architectural difference is the adoption of a co-allocated virtualization approach due to the non-existence of the secure monitor mode. Even in this co-allocated approach, the hypervisor is decoupled from the secure OS since we strive to keep our virtualization infrastructure as much OS-agnostic as possible. Furthermore, we decided to implement a from-scratch solution because (i) TrustZone-assisted hypervisors are characterized by a very small TCB which is very dependent on the TrustZone specification and due to (ii) envisioned roadmap and future activities which will be completely focused on low-end devices. Fig. 2 depicts the implemented single-core architecture and subsections III-A to III-D goes through the implementation details (CPU virtualization, memory and device partitioning, and interrupt and time management) for such configuration. In addition, given that TrustZone-M platforms are also shifting towards multi-core, we have also extended our implementation for an asymmetric multi-core (AMP) configuration - subsection III-E discusses the implementation details. Comparing the single-core to the AMP configuration, while the former assume guests are multiplexed in one core, the latter assumes guests run in parallel in different cores - both were implemented and the system can be tuned at design time according to the target platform.

A. CPU Virtualization

In essence, TrustZone technology virtualizes a physical core as two virtual cores. Between both virtual cores, there is a list of banked registers; this list encompasses the Stack Pointer (SP), as well as the Control (CONTROL) and Exception/Interrupt Masking (PRIMASK, FAULTMASK, BASEPRI) registers. The remaining core registers are shared among the secure and non-secure states. So, while implementing the Virtual Machine Control Block (VMCB) we make sure to include all general-purpose registers (R0-R12), along with the Link Register (LR), Program Counter (PC), and Application, Interrupt and Execution Program Status Registers. In the

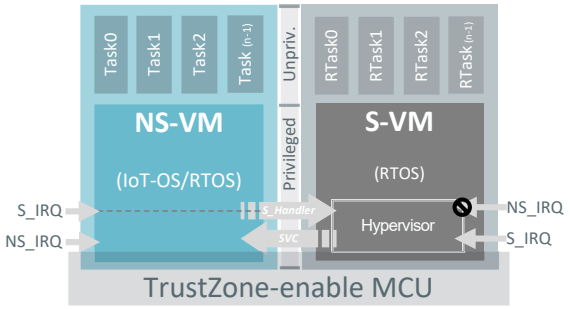


Fig. 2: TrustZone-M-assisted hypervisor (single-core)

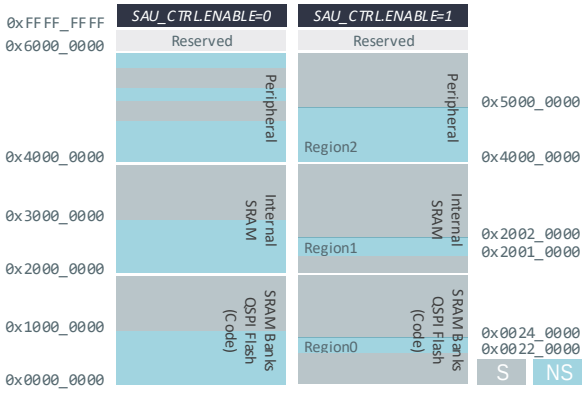


Fig. 3: Memory map with a) SAU disable and b) SAU enable.

single-core configuration, VMCBs are managed at every world switch, and it is the sole responsibility of the secure software to sanitize any sensitive information held in these registers.

We borrow the asymmetric scheduling policy from SafeG [16] and LTZVisor [17]. So, the implemented scheduler ensures that the non-secure VM (NS-VM) is only scheduled during the idle periods of the secure VM (S-VM), and that the S-VM resumes its execution as soon as a secure IRQ is triggered (Fig. 2). A transition from the S-VM to the NS-VM is triggered by explicitly using an SVC exception. In addition, we have fixed the layout of the VMCB according to the stack frame defined by the Procedure Call Standard for Arm Architecture (AAPCS). This ensures that the majority of non-banked core registers are automatically (un)stacked by the hardware itself, which significantly reduces the world switch overhead.

B. Memory and Device Partition

In existing TrustZone-assisted virtualization solutions memory cannot be virtualized; instead, memory is partitioned (i.e., through the TZASC). As TrustZone controllers do not provide virtualization of addresses, all VMs must be non-overlapping, which means they need to cooperate on sharing a single physical memory address space.

In the TrustZone-M architecture, memory virtualization is already non-existent. Memory regions can be configured and partitioned as secure or non-secure (or NSC) using a programmable SAU. So, similarly to the TZASC, which is a major requirement for TrustZone-assisted virtualization, the SAU is a major requirement for TrustZone-M-assisted virtualization. The difference is that while the TZASC is an optional component on TrustZone specification, the SAU is mandatory. Moreover, while TZASCs available on commercial Armv7-A platforms have limited flexibility and granularity, we noted that the SAU provides a configurable number of memory regions (0, 4 or 8), which can be programmed at unlimited granularity. So, to configure an SAU region, we needed to configure the base (SAU_RBAR) and limit (SAU_LBAR) addresses, as well as its security state. For example, on the Arm Musca-A platform [27], the implemented IDAU defines (when SAU is disabled) a memory map with consecutive and alternated 256 MB secure and non-secure memory regions (Fig. 3a). In contrast, Fig. 3b depicts how we have configured the SAU to set three non-secure memory regions (common memory layout - see Table II): Region0 in the QSPI Flash (code), Region1 in an internal SRAM (data), and Region2 in the Peripheral area. NSC entry points are not explored in the current implementation since the transition from the NS-VM to the S-VM goes through a single entry point: a secure interrupt. NSC regions may be explored in the near future to implement inter-VM communication.

While in Cortex-A processors two independent TrustZone controllers exist to configure memory and devices, in TrustZone-enabled MCUs the security state of memory and devices is configured exclusively through the SAU (and complementary platform-specific TrustZone protection controllers). Our current implementation supports pass-through device virtualization. This means devices are statically assigned and directly managed by guest OSes. So, memory-mapped devices assigned to the S-VM are configured as secure, while devices assigned to the NS-VM are configured as non-secure.

C. Interrupt Management

In Cortex-M MCUs interrupt management is particularly different than in Cortex-A processors because the TrustZone-enabled NVIC just supports classic interrupt requests (IRQs). Notwithstanding, it is possible to configure IRQs as secure or non-secure, by adequately configuring the NVIC_ITNS register. There are no restrictions regarding whether a non-secure or secure interrupt can occur when the processor is in the secure or non-secure state. This means, in TrustZone-M, it is possible for a non-secure interrupt to preempt the execution of the S-VM (Fig. 1b). However, to keep with the asymmetric design principle, while avoiding any kind of denial-of-service (DoS) attack from the NS-VM to the S-VM, non-secure IRQs are configured to have a lower priority than secure interrupts and are disabled (through the PRIMASK_NS register) while the S-VM is running. Moreover, while the NS-VM is running, it has not granted rights to change the security state of interrupts. Any attempt, from the NS-VM, to change any secure NVIC register will have no effect, and any attempt from the NS-VM to redirect an interrupt source to a secure interrupt handler will be trapped to the hypervisor.

Although the TrustZone-enabled NVIC has limited configuration flexibility when compared to the TrustZone-enabled GIC, the Vector Table Relocation (VTR) feature opens the opportunity to modify the location of the exception vector at run-time. The Vector Table Offset Register (VTOR) is a banked register that indicates the offset of the vector table base address. Since a different register exists per world, VTOR enables the existence of a vector table per VM. So, each VM is able to independently handle its own interrupts, without any hypervisor interference (Fig. 2). Hypervisor indirection just occurs when the NS-VM is running and a secure IRQ is triggered. For the S-VM, we have implemented two secure vector tables: (i) an active secure vector table and (ii) a passive secure vector table. The former is used when the S-VM is active, and all interrupts and exceptions are directly handled by the S-VM - except the Supervisor Call (SVC), which is handled by the hypervisor. The latter is used when the S-VM is idle, and all secure interrupts and exceptions which might happen need to be mediated through the hypervisor (Fig. 2). The secure VTOR register is updated between the active and passive vector table at every world switch.

D. Time Management

Temporal isolation is a mandatory requirement for virtualization, which is typically achieved using a hierarchical scheduling and timing strategy: both at the hypervisor and guest level [28], [29]. Our hypervisor implements a very specific time management strategy. It leverages the timing facilities available on TrustZone-enabled MCUs to allow VMs direct access to timekeeping mechanisms without any hypervisor interference. In all TrustZone-M-enabled MCUs, a 24-bit system timer (SysTick) exists per world. So, for a dual-OS configuration, it is possible to directly assign an independent timing unit per VM. This, altogether with the asymmetric design principle, ensures the SysTick dedicated to the S-VM has higher privilege than the timing unit dedicated to the NS-VM. While this approach ensures that the S-VM keeps track of the real passing of time (not missing any interrupt) independently on the time management strategy at the OS level (tick-driven or tickless), this is not necessarily true (in a single-core configuration) for the NS-VM. If we assume that the NS-VM, as a tick-driven OS, has a tick rate small enough that cannot be handled under the idle periods of the S-VM, then we will definitely lose track of the real passing of time. Unfortunately, the Armv8-M architecture does not provide any mechanism for directly injecting missing interrupts. So, in this case, we have to make one of two assumptions: (i) the guest OS running in the NS-VM implements a tickless strategy based on wall-clock time, or (ii) the workload of the secure real-time environment is known at design time and the system designer ensures idle periods are enough to handle the non-secure SysTick interrupt. In any case, when transitioning from a single-core to a multi-core approach (Section III-E) the majority of timing limitations are intrinsically overcome.

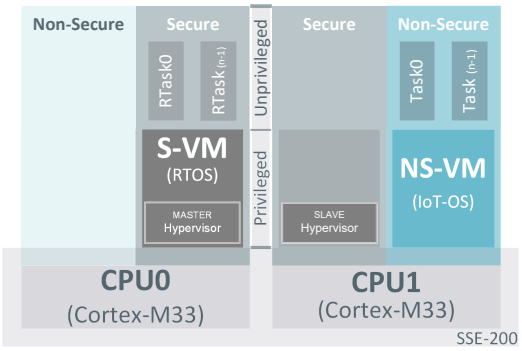


Fig. 4: Asymmetric multi-core system architecture (AMP).

E. AMP Configuration

Fig. 4 depicts the implemented AMP architecture featuring the Arm Musca-A platform (see Appendix A). As presented in Fig. 4, in the AMP configuration each VM is pinned to a specific core: the S-VM is assigned to the main processor (CPU0), while the NS-VM is assigned to the secondary core (CPU1). The hypervisor is split into two parts: the master runs in the CPU0 while the slave runs in the CPU1. In the AMP configuration, the role of the hypervisor during runtime is almost null (unless inter-VM communication is implemented). This is due to the fact there is a one-to-one mapping between the number of guests, the number of cores, and the number of virtual states supported by the cores. The majority of hypervisor-related operations are performed at boot time. Since each Cortex-M33 runs its own VM OS instance, no scheduling points and world switch operations are performed. The master hypervisor is only responsible for partitioning the system at start-up by adequately configuring the security state of the several resources. It is responsible for configuring the SAU and the NVIC to partition memory and interrupts as secure and non-secure, respectively, as well as for creating both secure and non-secure VMs. It is then responsible for kicking the secondary CPU and starting the slave hypervisor. While in TrustZone-enabled Cortex-A processors the TZASC is shared among all cores in a SoC, in TrustZone-enabled MCUs there is an SAU per core. So, the slave hypervisor is also responsible for configuring the SAU and the NVIC of the secondary core according to the same security model of the master. After initializations, the slave hypervisor starts the execution of the NS-VM, by explicitly triggering an SVC.

F. Scalability

Although our current implementation does not provide support for multiple guests, according to our previous experience [26], [30], existing TrustZone hardware resources available on Armv8-M MCUs make it also possible to implement multi-guest support. The implementation of such support will overcome the main limitation of the classic TrustZone model (split all code bases into only two worlds), and bring several benefits in terms of isolation and flexibility for the current and upcoming needs of embedded and IoT devices. We plan

to implement an architecture where the hypervisor executes standalone in the secure privileged mode, while the multiple guests can be split between running on the normal world and preserved and isolated on the secure world. As a result, shared resources need to be managed and re-configured by the hypervisor. Notwithstanding, the SAU enables the re-configuration of the security state of memory and devices at runtime (with higher flexibility than the TZASC), as well as the NVIC enables the re-configuration of the security state of interrupts also at runtime. Reconfiguration of the security state of memory, devices, and interrupts is the main requirement for implementing multi-guest support in TrustZone-assisted virtualization infrastructures. Moreover, by adopting such an approach, we will also be able to solve the lack of isolation existing between the secure co-allocated components and shrink the TCB of the system to the size of the hypervisor. While scalability in terms of the number of guests seems to perfectly fit in a single-core approach, this is not necessarily true for a multi-core configuration. For multi-core platforms, we believe only a true synergy between TrustZone and paravirtualization will bring real scalability.

IV. Predictable Shared Resources Management

Embedded virtualization has several proven benefits but still faces serious challenges, specially when real-time is a concern. On one hand, it already provides a reasonably high degree of time and space encapsulation and isolation of VMs by time-multiplexing resources such as the CPU, partitioning memory, and assigning or emulating devices. On the other hand, partitioning and multiplexing of micro-architectural shared system resources were, until recently, neglected by most hypervisors. This led to contention and lack of truly temporal isolation, hurting determinism by increasing jitter [7], [31]. Also, this can be explored by a malicious VM to implement DoS attacks by increasing their consumption of a shared resource. Moreover, it allows for the existence of timing side-channels compromising data confidentiality, which might be exploited to access private or sensitive data of either a VM or the hypervisor [31], [32]. Although AMP hypervisors with VMs pinned to dedicated cores already remove part of this contention when compared to single-core or symmetric multiprocessing (SMP) implementations, system-wide resources such as last-level caches (LLCs), memory controllers, and interconnects still remain shared and subject to contention. This is further aggravated as mechanisms such as cache replacement, cache coherency, hardware prefetching or memory controller scheduling focus mainly on performance and bandwidth maximization.

Many approaches, from which we highlight cache coloring [23], [33], memory bandwidth reservations [24], or both [7] have already been applied to mitigate these issues with promising results. However, these techniques depend on the existence of memory virtualization infrastructure or performance monitoring features which are not available on MCUs. On the positive side, many of these contention points, such as data caches or TLBs, are seldom, if ever, featured in low-end platforms, and the absence of memory translation mechanisms or deep cache hierarchies further reduces the sources of indeterminism. From another perspective, and as detailed in [34], commercial MPSoCs exhibit a high degree of heterogeneity regarding their memory subsystem which is comprised of a rich set of different types of memory (e.g. DRAM, SRAM, QSPI Flash, etc.), each accessed through different bus paths and memory controllers, providing varying degrees of latency and bandwidth guarantees. Although this study focused on a high-end Cortex-A platform, this heterogeneity is also true and even more pronounced in modern MCU-based platforms.

Taking these ideas and insights in mind, we believe that it is possible to achieve a high degree of determinism on MCU AMP virtualization, through an informed and thoughtful layout of VM memory. This is accomplished by distributing data and code segments from different VMs through different memory elements, each with dedicated controllers accessed via bus paths which enable fully concurrent accesses or assigned in such a way that minimizes contention.

A. The Arm Musca-A Memory Subsystem

Arm Musca-A test chip [27] is a TrustZone-enabled platform targeting secure IoT designs and intended as a reference implementation for other SoCs using the same core IP. Therefore, despite the fact that, at the time of writing of this paper, the number of TrustZone-M-enabled platforms is relatively small, we believe that the analysis provided throughout this section will be valid and span across many other platforms. According to its block diagram (Fig. 9), the Musca-A encompasses the CoreLink SSE-200 subsystem featuring an asymmetric dual-core Cortex-M33, each with a private 2KB instruction cache and no data caches. The asymmetry of the design has a direct relation to the performance of the CPU, i.e., when running both CPUs at the same frequency, CPU1 is inherently slower than CPU0 (see Appendix C). The cores are connected to the main bus matrix, a multi-layer AHB5 interconnect, that enables parallel access paths between multiple masters and slaves in the system. A set of four 32KB internal SRAM (iSRAM) elements is also accessible via this main bus. Each of them features a dedicated controller and are therefore considered separate slaves on the bus. Consequently, when each of these memory elements is assigned exclusively to each CPU, it results in no contention. Although both cores can access any of these memory elements, Musca's documentation details that SRAM element 3 is a tightly coupled memory (TCM) to CPU1's data port. In contact with Arm support, we unveiled that the remaining SRAM elements are also TCMs coupled to CPU0. This uneven coupling further increases the asymmetry of the design. Still, as part of the SSE-200, the main bus connects two slave AHB2APB bus bridges which allow access to system control registers and peripherals.

Finally, in Musca-A, two expansion ports extend the SSE-200 AHB bus matrix: one connecting a set of APB slaves encompassing I/O functionality while the other connects two memory elements targeted only to code storage and execution. The first is a 2MB code external SRAM (eSRAM) clocked at

the same frequency as CPU0. Using this code eSRAM for storing data is possible but impractical, as it does not support unaligned accesses. The second is a QPSI controller connected to an external QSPI 8MB boot flash memory, clocked at a much lower frequency than both CPUs. The instruction caches only cache addresses where these two memories are mapped. Although each element is accessed through distinct controllers, as explained before, they are connected to the main bus through a single expansion port for the code memory region, which prevents full concurrency when each of them is assigned to a different CPU.

B. Contention-Aware Memory Layout

The previous Musca-A chip's memory subsystem and interconnect analysis, supported by a set of empirical observations (see Section V-C) allows us to come up with a memory layout, which is intended to minimize contention of shared resources. Targeting the AMP configuration (Section III-E) and prioritizing the S-VM (RTOS) running on CPU0, we start with an idealistic layout scenario and iteratively rearrange it until all VMs' memory is allocated.

For very low memory footprint systems, all code and data for both VMs would completely fit in iSRAM, resulting in no contention (see Fig. 10a in Appendix B). In this case, we start by assigning iSRAM0 to S-VM and iSRAM3 to NS-VM, given their tightly coupled nature, and distributing the remaining SRAM elements according to each VM memory needs, maintaining the invariant that each one is exclusively assigned to a single VM. As a FreeRTOS image compiled with only a small toy application amounts to about 25KB, this would only be feasible when only minimal functionality was included in each VM. In a more realistic setting, there is the need to offload code segments to the external memories. If possible, we maintain one of the VMs full images in the iSRAM and migrate the other's code to the eSRAM. If not, we migrate both. This layout, where VMs share eSRAM for code segments, and each has dedicated iSRAM elements for data, constitutes the best option for real use cases (see Fig. 10b). At first sight, assigning one of the VMs to QSPI flash would minimize contention as there would be no sharing of a controller (see Fig. 10c). However, sharing eSRAM results in better performance and less contention, given that QSPI is clocked at a much lower frequency. Our observations show that when a core is accessing this memory, on a concurrent request, the latter will be stalled for a longer period until the former is served, as the bus expansion port is also shared. We believe that the eSRAM's 2MB will suffice for hosting both VMs' code. If not, we shall place the NS-VM, or even partially the S-VM, in QSPI. This results in the worst scenario regarding both performance and contention (see Fig. 10d). Note that the impact of moving both code segments to the external memories, despite sharing the same bus expansion master port or the same controller, will not greatly increase contention when good code locality is present, as we assume that instruction caches will always be enabled in both cores. This is not guaranteed, however, since a compromised OS running on NS-VM (CPU1) could disable caches and increase contention on the bus expansion connected to the code memories. Although in the Musca-A platform, the non-secure software cannot access the cache control register, this continues to be true, as it can execute in such a way that continuously thrashes cache lines.

We are aware that this is an *ad hoc* approach that requires burdensome analysis of the platform memory subsystem and error-prone manual modifications to each VM linker script. Therefore, is not easily applied to a different target. We envision a tool that from available platform models and a description of the VMs requirements automates the whole process. Also, this allocation scheme does not contemplate peripheral assignment, which might result in contention if different peripherals on the same APB are assigned to each VM. This is out of the scope of this paper and it will be explored in future works.

V. Evaluation

The evaluation was conducted on an Arm Musca-A Test Chip Board running both cores at 50 MHz. We evaluate the system for both single- and multi-core (AMP) configurations. FreeRTOS (version 9.0.0) was used as the guest OS for both secure and non-secure VMs. The hypervisor and both VMs were compiled using the GNU Arm Embedded Toolchain (version 7-2017-q4-major), with -Os optimizations (in Section V-C -O0 were used as well). Our evaluation focused on performance, interrupt latency, and contention. Section V-A focuses on performance; we aim at assessing the runtime overhead imposed by the hypervisor on the VM execution. Section V-B evaluates the interrupt latency in order to understand the additional jitter, at the VM level, caused by the underlying infrastructure. Finally, Section V-C evaluates contention; we aim at understanding how the system memory layout can lead the NS-VM to create contention on shared buses and consequently affect the timing predictability of the S-VM.

A. Performance Overhead

To assess the performance overhead introduced by the hypervisor, we ran the Thread-Metric benchmarks. The Thread-Metric Benchmark Suite [35] is an open-source, vendor-neutral, free benchmark suite that measures RTOS performance. The suite comprises 7 benchmarks, evaluating the most common RTOS services and interrupt processing; each benchmark outputs a counter value, representing the RTOS impact on the running application - higher scores correspond to a smaller impact.

We ran the seven benchmarks for three different system configurations, presented in Table I. First, we evaluated the system in a native environment, i.e., by running FreeRTOS directly on top of CPU0. Then, we evaluated the virtualized single- and multi-core configurations; for both configurations, we evaluated the system from two different perspectives: the S-VM and the NS-VM. For the single-core configuration, this means both VMs share the same CPU (i.e., CPU0); for the multi-core configuration, both VMs run in parallel on different

System Configuration		CPU0		CPU1	
		Secure	Non-Secure	Secure	Non-Secure
Native	N	●	○	×	×
Single-core	S-VM	●	○	×	×
	NS-VM	i	●		
Multi-core	AMP S-VM	●	○	○	i
	AMP NS-VM	i	○	○	●

TABLE I: System configurations under evaluation. Symbols indicate whether the system runs (●) or does not run (○) in a specific security state of a particular CPU, or the CPU is non-existent (×); the (i) symbol indicates that the twin VM (which runs in a different security state) is idle.

CPUs: the AMP S-VM runs in CPU0 while the AMP NS-VM runs in CPU1. When running the benchmark in one VM, the other VM is idle but the SysTick is kept active, i.e. the FreeRTOS scheduler will still be triggered periodically, and the idle task will execute minimal memory management services before yielding execution. For the first part of the experiments, the SysTick of all VM OS instances was configured with a period of 1ms. Finally, we allocated the memory map so that the code of both VMs are loaded into the eSRAM and the respective data into different iSRAMs (common memory layout - see Table II).

Fig. 5 presents the achieved results, where each bar shows the average relative performance of 10000 collected samples (each sample reflects 30 seconds of execution). The values on top of the bars show the average absolute performance, i.e. the average value of the output counter for the benchmark. As it can be seen, for the single-core configuration, the S-VM has no performance penalty (asymmetric design principle), while the NS-VM presents, on average, a performance degradation of about 0.6%. This performance degradation is the result of the periodic preemption imposed by the S-VM, i.e. is the overhead for getting S-VM in context every millisecond. Finally, regarding the AMP configuration, there are two notes worth mentioning. First, the AMP S-VM, although running without any hypervisor interference, presents a small performance degradation when compared to native execution, which varies across the different benchmarks. This degradation is related to contention on shared resources, which will be discussed in detail in Section V-C. Second, the AMP NS-VM performance is significantly reduced. This is not related to the virtualization overhead, but mainly due to the execution of the NS-VM on the secondary core (CPU1) which is inherently slower when running at the base frequency (i.e., 50 MHz). A comparison between a native execution of FreeRTOS on CPU0 and CPU1 demonstrated a decrease of performance on the same order of magnitude (see Appendix C).

S-VM Systick overhead. In the second part of the experiments, we focus on evaluating the correlation between the SysTick rate of one VM and the performance overhead of the other VM. We have repeated the previous experiments for three different SysTick rates ranging from 1ms to 100μs. Each point corresponds to the geometric mean of the collected samples for the seven benchmarks, encompassing a total of 70000

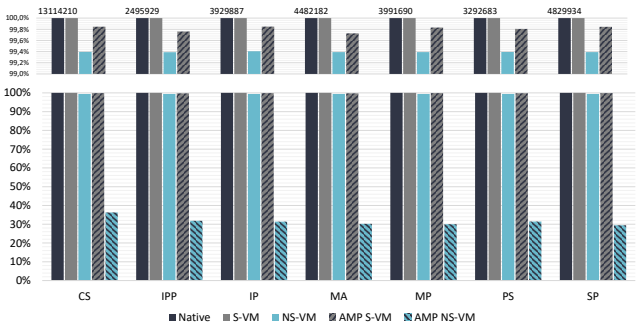


Fig. 5: Performance for Thread-Metric benchmarks: cooperative context switching (CS), interrupt processing with preemption (IPP), interrupt processing (IP), memory allocation and deallocation (MA), message passing (MP), preemptive context switching (PS), and semaphore processing (SP).

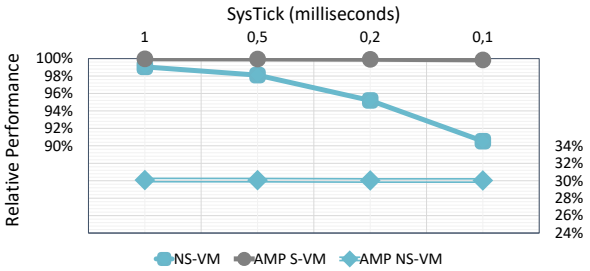


Fig. 6: Relative performance for different SysTick rates

samples per point. From Fig. 6 we can conclude that (i) in the single-core configuration the performance of the NS-VM decreases as the SysTick rate of the S-VM increases while (ii) in the AMP configuration the performance of each VM slightly decreases as the SysTick rate of the other VM increases. Although this phenomenon is not noticeable in Fig. 6, we observed a performance degradation of about 0.15%. Further analysis must be carried out to fully justify that the decrease of performance is not directly related to SysTick itself, but it is instead a consequence of concurrency while stressing buses when the FreeRTOS goes through the scheduler. The scheduler needs to go through several critical internal data structures (e.g., task and synchronization control blocks) while following different execution paths (e.g., affecting code locality).

Starvation. The asymmetric design principal (single-core configuration), borrowed from SafeG and LTZVisor, ensures that the S-VM has a greater scheduling priority than the NS-VM. While this ensures the timing requirements of the (secure) real-time environment remain nearly intact, it also gives rise to starvation of the NS-VM. We have repeated the aforementioned experiments, but using different workloads. We added one real-time task to the FreeRTOS instance running as S-VM, as a way of emulating different workloads on the S-VM. Four different workloads were emulated with utilizations of 0, 25, 50 and 75%. FreeRTOS running as S-VM has the SysTick configured to trigger every millisecond. This means the real-time task will

be consuming the CPU for 0, 250, 500 and 750 microseconds, respectively. According to our experiments, we concluded that in the single-core configuration the performance of the NS-VM decreases linearly as the workload increases. In the worst case, it will lead to a complete starvation of the NS-VM, in case the S-VM never releases the CPU. For the AMP configuration, starvation is completely overcome, and there is only a slight performance decrease which is related to contention (see Section V-C).

B. Interrupt latency

Interrupt latency, which can be defined as the time from the moment an interrupt is triggered until the moment the handler starts to execute, is a critical metric for real-time systems. To assess the interrupt latency, we set up a dedicated timer to trigger an interrupt every 10 ms, while guaranteeing this is enough time for the interrupt to be serviced and resume previous execution. As the timer is configured in incrementing, auto-reload mode, latency is obtained directly by reading the counter register at the beginning of the interrupt handler. We measured the interrupt latency for the system configurations presented in Table I. For the single-core configuration, we took into consideration the best and the worst case scenarios, i.e. when an interrupt is directly handled by the VM, as well as when it is mediated by the hypervisor. For example, for the S-VM, we measured the interrupt latency when an interrupt is triggered while the S-VM is running, as well as while the NS-VM is running (i.e., a world switch needs to be performed). All measurements were repeated 10000 times. Fig. 7 shows the assessed results, which expresses the best- and the worst-case execution time (WCET). According to our experiments and results, it is clear the additional overhead introduced in a single-core configuration. This is perfectly understandable as both VMs necessarily need to share the same CPU, which requires an additional world switch. Notwithstanding, while for the S-VM the WCET has a well-defined upper bound, for NS-VM this is not necessarily true. The lemniscate symbol on top of the bar means the NS-VM interrupt latency, on the WCET, has no specific upper bound, due to the possible starvation imposed by the S-VM. Notwithstanding, we are perfectly aware of this limitation, and this is why we only envision the usage of a soft real-time or IoT-enabled OS as NS-VM. Regarding the test cases for the AMP configuration, each VM always directly handles its own interrupts without any hypervisor interference. However, we highlight two observations that deserve an explanation: first, the additional jitter on both cases is mainly explained by concurrency on memory and buses; and second, the increased value for the AMP NS-VM is related to the fact the VM is executed on the CPU1, which is inherently slower than CPU0.

C. Contention

Finally, in the last part of our experiments, we focused on evaluating contention. As such, we focused on the AMP configuration and consequently on observing how the S-VM timing predictability may be hampered by NS-VM execution.

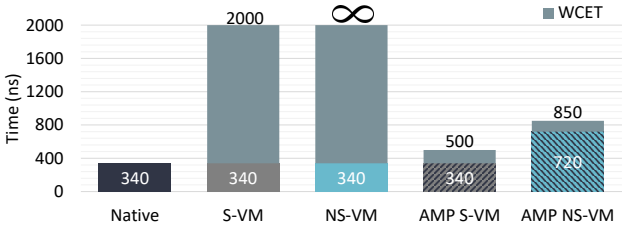


Fig. 7: Interrupt Latency

System Configuration			iSRAM0-2	iSRAM3	eSRAM	QSPI
Pessimistic	S-VM	data	●	-	-	-
		code	-	-	-	●
	NS-VM	data	●	-	-	-
		code	-	-	-	●
Random	S-VM	data	●	-	-	-
		code	-	-	●	-
	NS-VM	data	-	●	-	-
		code	-	-	-	●
Common	S-VM	data	●	-	-	-
		code	-	-	●	-
	NS-VM	data	-	●	-	-
		code	-	-	●	-
Optimistic	S-VM	data	●	-	-	-
		code	●	-	-	-
	NS-VM	data	-	●	-	-
		code	-	-	●	-

TABLE II: System configurations used to evaluate contention. Symbols indicate whether a memory is used (●) or not (−).

To evaluate such interference we set four system configurations with different memory layouts. System configurations are summarized in TABLE II. In a pessimistic memory layout (P) the system is configured to stress concurrency, by running the code of both VMs from the QSPI, and data from the same iSRAM element. In a random memory layout (R), the system designer has no concerns regarding the memory map; so, the code related to the S-VM and NS-VM runs from the eSRAM and QSPI, respectively, while the data is placed in different iSRAMs. In a common memory layout (C) both VMs are loaded into the eSRAM and respective data into different iSRAMs (memory layout used in all aforementioned experiments). Finally, in an optimistic scenario (O) code and data of the S-VM are small enough to fit within a single tightly coupled iSRAM, and the code and data of the NS-VM are placed on the eSRAM and a tightly coupled iSRAM, respectively. This scenario is similar to the ideal one mentioned in Section IV-B (and illustrated Fig. 10a as well) since the S-VM is still completely isolated; however, it is a bit more realistic, as non-trusted VM code does not also need to completely fit in the single 32KB iSRAM element tightly coupled to CPU1. In all test cases scenarios, the hypervisor runs from the same memory as the S-VM. The S-VM is running the Thread-Metric basic processing benchmark while the NS-VM runs the FreeRTOS configured without any task. Contention may result from the interference of NS-VM in accessing memory while running the idle task and internal scheduling-related services. Moreover, on CPU1 (which is running the NS-VM) the instruction cache is disabled as a way to maximize contention.

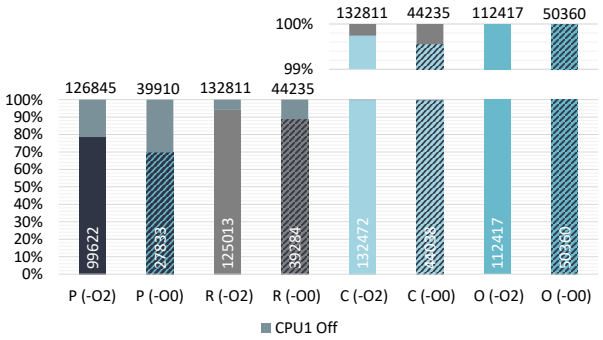


Fig. 8: S-VM performance (Thread-Metric basic processing benchmark) with and without NS-VM interference for different memory layouts. Values on top and inside the bars shows the maximum and minimum output score of the benchmark.

Fig. 8 presents achieved results. Each bar shows the best and WCET, i.e. when the CPU1 (NS-VM) is enabled and disabled. We have collected 10000 samples per test per bar. From Fig. 8 we can conclude that for the pessimistic and random memory layout there is significant interference from the NS-VM on the S-VM. The NS-VM is able to add a considerable interference, which results in a lack of determinism and timing predictability. In contrast, when the memory layout is distributed according to the guidelines proposed in Section IV-B, the S-VM suffers none or really small interference (common and optimistic system configuration, respectively) from the NS-VM. Furthermore, we can conclude that the higher are the compilation optimizations, the smaller is the contention. This is related to the reduction in the number of memory access instructions. Finally, it is worth mentioning we have repeated the experiments related to the interrupt latency for these four system configurations, and we have observed the interrupt latency follows a similar pattern as the measured performance. For example, for an optimistic memory layout, the jitter on the S-VM interrupt latency is also non-existent.

VI. Related Work

There is a rich body of hypervisor solutions, mainly due to the large spectrum of use cases for virtualization [8], [36]. While classical virtualization was mainly implemented through trap-and-emulation, paravirtualization, and binary translation techniques [37], [38], the advances on hardware support for virtualization brought to light a set of efficient hardware-assisted solutions [9], [10], [39], [40]. Due to the extensive list of works on the virtualization landscape, we will focus on the two following classes of existing solutions.

TrustZone-assisted hypervisors. The idea of using TrustZone technology to implement hardware-assisted virtualization solutions for (real-time) embedded systems applications is not new. Frenzel et al. [25] pioneered research in this domain by proposing the use of TrustZone for implementing the Nizza secure architecture [41]. SafeG [16], SASP [19], and LTZvisor [17] are dual-OS solutions which take advantage of TrustZone extensions for virtualization. SafeG [16] is an open-source solution which allows the consolidation of two different execution environments: an RTOS such as TOPPERS ASP kernel, and a GPOS such as Linux or Android. SASP [19] implements a lightweight virtualization approach which explores TrustZone technology to provide isolation between a control system and an in-vehicle infotainment (IVI) system. LTZVisor [17] is an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticality systems. VOSYSmonitor [42] is a closed-source product developed and maintained by Virtual Open Systems. While the lack of scalability was the main reason that led several researchers to perceive TrustZone as an ill-guided virtualization technique for many years, RTZVisor [26] and μRTZVisor [30] have recently demonstrated how multiple OS instances are able to coexist, completely isolated from each other, on TrustZone-enabled platforms. To the best of our knowledge, all aforementioned works do no support Armv8-M - they are implemented in Cortex-A processors and targeting mid- to high-end applications.

Low-end hardware-enforced separation. While classical approaches which provide isolation in resource-constrained systems rely on constructive (language/compiler-based) memory protection [43], [44], several embedded OSes focus instead on hardware-tailored protection mechanisms by taking advantage of MPU facilities [43], [45]. These systems provide increased reliability, but not necessarily ensure strong isolation for mixed-criticality. Virtualization for low-end and low-cost devices is in its infancy, and only a few solutions have been proposed so far. F. Paci et al. [11] proposed a lightweight I/O virtualization solution for MCUs by integrating MPU support on FreeRTOS and implementing a specific task which mediates all other tasks I/O accesses. Additionally, both F. Bruns et al. [6] and R. Pan et al. [12] have proposed interesting virtualization infrastructures on MPU-based MCUs which are able to support strong isolation along the CPU, memory, and I/O dimensions. Our work, in contrast, focuses on exploring TrustZone-M technology for strong hardware-enforced separation while avoiding the paravirtualization effort associated with existing solutions [6], [12]. The Arm Mbed uVisor (deprecated as of Mbed OS 5.10), although supporting Armv8-M, just implements a supervisory kernel on Mbed OS and is not able to host two OSes instances. To the best of our knowledge, this paper presents the first virtualization infrastructure for TrustZone-enabled MCUs.

VII. Conclusion

In this paper, we have introduced a lightweight virtualization infrastructure for low-end and low-cost systems. We have shown how the TrustZone-M technology can effectively be exploited to provide isolation on mixed-criticality systems which are expected to be deployed on billions of tomorrow's Arm MCUs. The implemented hypervisor was evaluated on a reference low-end Arm multi-core platform, and the assessed results demonstrate reduced memory footprint and high efficiency and determinism.

VIII. Acknowledgements

We would like to thank the anonymous reviewers for their helpful feedback, and Arm Ltd. for the loan of the Arm Musca-A Test Chip Board. This work has been supported by FCT - Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2019.

References

[1] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton, "Smart objects as building blocks for the Internet of things," *IEEE Internet Computing*, vol. 14, no. 1, pp. 44–51, Jan 2010.

[2] P. Sparks, "The route to a trillion devices," White Paper, ARM, 2017.

[3] Gartner. (2017) Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016. [Online]. Available: https://www.gartner.com/newsroom/id/3598917

[4] A. Sadeghi, C. Wachsmann, and M. Waidner, "Security and privacy challenges in industrial Internet of Things," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[5] R. Ernst and M. D. Natale, "Mixed Criticality Systems - A History of Misconceptions?" *IEEE Design Test*, vol. 33, no. 5, pp. 65–74, Oct 2016.

[6] F. Bruns, D. Kuschnerus, and A. Bilgic, "Virtualization for Safety-critical, Deeply-embedded Devices," in *ACM Symposium on Applied Computing (SAC)*, 2013, pp. 1485–1492.

[7] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms," in *IEEE Int. Conf. on Industrial Technology*, Feb 2018, pp. 1651–1657.

[8] G. Heiser, "The Role of Virtualization in Embedded Systems," in *Workshop on Isolation and Integration in Embedded Systems*, 2008.

[9] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 333–348, Feb. 2014.

[10] R. West, Y. Li, E. Missimer, and M. Danish, "A virtualized separation kernel for mixed-criticality systems," *ACM Transactions on Computer Systems*, vol. 34, no. 3, pp. 8:1–8:41, Jun. 2016.

[11] F. Paci, D. Brunelli, and L. Benini, "Lightweight IO virtualization on MPU enabled microcontrollers," *ACM SIGBED Review*, vol. 15, no. 1, pp. 50–56, 2018.

[12] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable Virtualization on Memory Protection Unit-Based Microcontrollers," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018, pp. 62–74.

[13] Arm, "Automotive safety hypervisor announced for ARM Cortex-R52," January 2017, [Online:] https://www.arm.com/about/newsroom/automotive-safety-hypervisor-announced-for-arm-cortex-r52.php.

[14] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 51, no. 6, Jan 2019.

[15] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications," *SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 67–80, Feb. 2014.

[16] D. Sangorrin, S. Honda, and H. Takada, "Dual operating system architecture for real-time embedded systems," in *Workshop on Operat. Syst. Platforms for Embedded Real-Time Applications*, 2010, pp. 6–15.

[17] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the Key," in *29th Euromicro Conference on Real-Time Systems (ECRTS)*, 2017, pp. 4:1–4:22.

[18] Arm, "TrustZone technology for ARMv8-M Architecture," Version 2.0 (100690_0200_00_en), Arm Ltd., March 2017.

[19] S. W. Kim, C. Lee, M. Jeon, H. Y. Kwon, H. W. Lee, and C. Yoo, "Secure device access for automotive software," in *International Conference on Connected Vehicles and Expo (ICCVE)*, Dec 2013, pp. 177–181.

[20] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms," *IEEE Trans. on Paral. and Distri. Systems*, vol. 20, no. 4, pp. 553–566, 2009.

[21] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 45–54.

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

[23] M. Xu, L. Thi, X. Phan, H. Choi, and I. Lee, "vCAT: Dynamic Cache Management Using CAT Virtualization," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2017, pp. 211–222.

[24] A. Crespo, A. Soriano, P. Balbastre, J. Coronel, D. Gracia, and P. Bonnot, "Hypervisor Feedback Control of Mixed Critical Systems: the XtratuM Approach," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

[25] T. Frenzel, A. Lackorzynski, A. W. H., and Härtig, "ARM TrustZone as a Virtualization Technique in Embedded Systems," *Twelfth Real-Time Linux Workshop*, 2010.

[26] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems," *IEEE Comp. Arch. Letters*, vol. 16, no. 2, pp. 158–161, 2017.

[27] Arm, "Arm Musca-A Test Chip and Board Technical Reference Manual," Arm Ltd., January 2018.

[28] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening Hierarchical Scheduling," in *EMSOFT*, 2012, pp. 93–102.

[29] M. Drescher, V. Legout, A. Barbalace, and B. Ravindran, "A Flattened Hierarchical Scheduler for Real-time Virtualization," in *International Conference on Embedded Software (EMSOFT)*, 2016, pp. 12:1–12:10.

[30] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, "uRTZVisor: A Secure and Safe Real-Time Hypervisor," *Electronics*, vol. 6, no. 4, 2017.

[31] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018, pp. 98:1–98:6.

[32] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr 2018.

[33] H. Kim and R. Rajkumar, "Predictable Shared Cache Management for Multi-Core Real-Time Virtualization," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 1, pp. 22:1–22:27, Dec. 2017.

[34] A. Bansal, R. Tabish, G. Gracioli, R. Mancuso, R. Pellizzoni, and M. Caccamo, "Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2018, p. 55.

[35] W. Lamie and J. Carbone, "Measure your RTOS's real-time performance," *Embedded Systems Design*, 2007.

[36] J. Shuja, A. Gani, K. Bilal, A. Khan, S. Madani, S. Khan, and A. Zomaya, "A Survey of Mobile Device Virtualization: Taxonomy and State of the Art," *ACM Computing Surveys*, vol. 49, no. 1, Apr. 2016.

[37] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Virtualization Architecture," in *European Conference on Computer Systems (EuroSys)*, 2010, pp. 209–222.

[38] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim, "Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones," in *IEEE Consumer Communications and Networking Conference*, 2008, pp. 257–261.

[39] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look Mum, no VM Exits!(Almost)," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2017.

[40] Z. Jiang, N. C. Audsley, and P. Dong, "BlueVisor: A Scalable Real-Time Hardware Hypervisor for Many-Core Embedded Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018, pp. 75–84.

[41] H. Hartig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter, "The Nizza secure-system architecture," in *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[42] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, and D. Raho, "VOSYS-monitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A," in *29th Euromicro Conference on Real-Time Systems (ECRTS)*, 2017, pp. 6:1–6:18.

[43] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient Memory Safety for TinyOS," in *International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007, pp. 205–218.

[44] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kB Computer Safely and Efficiently," in *Symp. on Operating Systems Principles (SOSP)*, 2017, pp. 234–251.

[45] D. Danner, R. Mller, W. Schrder-Preikschat, W. Hofer, and D. Lohmann, "SAFER SLOTH: Efficient, hardware-tailored memory protection," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 37–48.

Appendix A
The Arm Musca-A Test Chip

The Musca-A Test Chip Board is a reference implementation of an Arm TrustZone-M system that helps system designers get IoT security right from the outset. The Musca-A board provides access to the Musca-A test chip (Fig. 9) which implements the Arm CoreLink SSE-200 Subsystem. According to Fig. 9, the SSE-200 subsystem features an asymmetric dual-core Cortex-M33, each with a private 2KB instruction cache. The asymmetry of the design is first due to fact that CPU1 is equipped with an FPU and DSP, and can run from 50 MHz up to 170 MHz, while CPU0 has no associated coprocessor and is only running at 50 MHz. The cores are connected to the main bus matrix, a multi-layer AHB5 interconnect, that enables parallel access paths between multiple masters and slaves in the system. Four 32KB internal SRAM elements are also connected to the main bus through an AHB5 Fabric and have a dedicated controller. According to the Musca-A Technical Reference Manual [27], the internal SRAM3 memory is tightly coupled to CPU1's data port; in contact with Arm, we unveiled that the remaining SRAM elements are also tightly coupled to CPU0. Still, as part of the SSE-200, the main bus connects two slave AHB2APB bus bridges which allow access to system control registers and peripherals. Additionally, the SSE-200 AHB bus matrix connects another two memory elements: one is a 2MB code external SRAM clocked at the same frequency as CPU0; the other is a QPSI controller connected to an external QSPI 8MB boot flash memory, clocked at a much lower frequency than both CPUs.

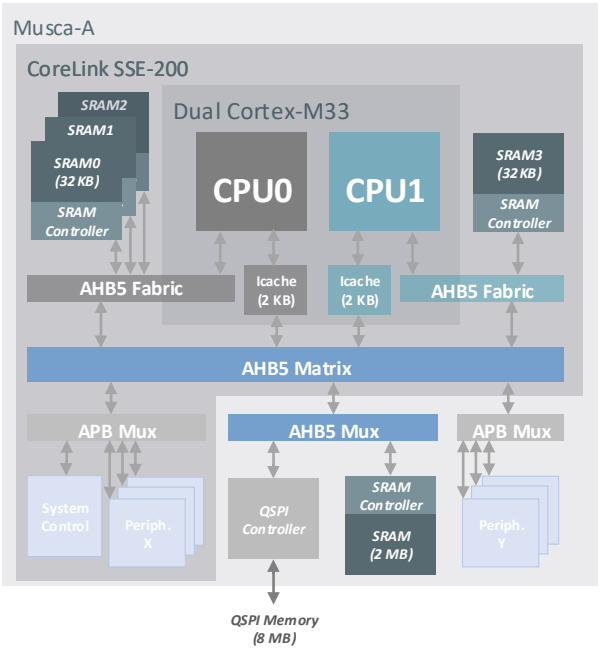
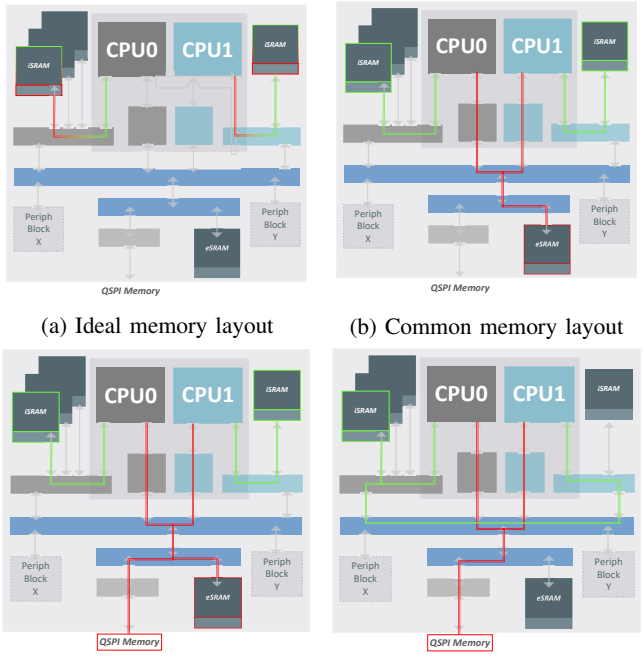


Fig. 9: Musca-A chip memory and interconnect block diagram. Adapted from [27].

Appendix B
Path of Memory Configurations in Musca-A

Fig. 10 shows the path for different memory configurations in the Arm Musca-A Test Chip Board.



(a) Ideal memory layout (b) Common memory layout

(c) Random memory layout (d) Pessimistic memory layout

Fig. 10: Path of memory configurations in Musca-A. Green line for data and red line for code.

Appendix C
Performance Asymmetry in Musca-A

Aiming at understanding the significant decrease of performance for the AMP NS-VM, we have conducted a set of experiments to evaluate the performance in the Musca-A Test Chip. We run the native version of FreeRTOS in different CPUs and in different security states. According to Fig. 11, when running at the same frequency (50 MHz), CPU1 can just reach near 30% of the performance of CPU0. So, the significant decrease of performance of the AMP NS-VM is not related to underlying virtualization infrastructure, but due to the architectural asymmetries in the design of the platform.

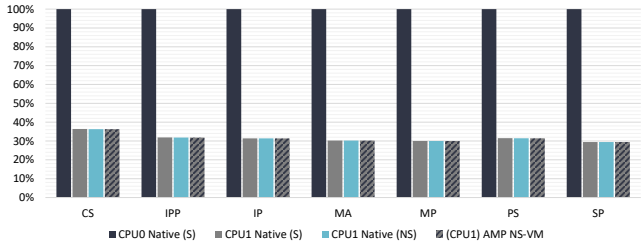


Fig. 11: Performance asymmetry in Musca-A: CPU0 vs CPU1