Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems

Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares

Abstract—Virtualization technology starts becoming more and more widespread in the embedded space. The penalties incurred by standard software-based virtualization is pushing research towards hardware-assisted solutions. Among the existing commercial off-the-shelf technologies for secure virtualization, ARM TrustZone is attracting particular attention. However, it is often seen with some scepticism due to the dual-OS limitation of existing state-of-the-art solutions. This letter presents the implementation of a TrustZone-based hypervisor for real-time embedded systems, which allows multiple RTOS partitions on the same hardware platform. The results demonstrate that virtualization overhead is less than 2 percent for a 10 milliseconds guest-switching rate, and the system remains deterministic. This work goes beyond related work by implementing a TrustZone-assisted solution that allows the execution of an arbitrary number of guest OSes while providing the foundation to drive next generation of secure virtualization solutions for resource-constrained embedded devices.

Index Terms—Virtualization, TrustZone, monitor, real-time, embedded systems, RODOS, ARM

1 Introduction

VIRTUALIZATION technology is a mainstream tool in the server and desktop space, presenting huge benefits in terms of load balancing, power management and service consolidation [10]. In the embedded systems arena, virtualization also has the potential to be a game-changer [6]. Driven by the possibility of consolidation, coexistence and isolation of multiple and heterogeneous OS environments, virtualization has attracted automotive, aeronautics, space and medical industries to build systems with reduced Size, Weight, Power and Cost (SWaP-C) budget [6], [8].

The traditional existing embedded virtualization solutions [8], [11], [12] typically follow two different approaches: full-virtualization and paravirtualization. Between both approaches there is a trade-off between flexibility and performance: paravirtualization [8], [11] incurs a higher design cost, while full-virtualization [12] incurs a higher performance cost. More recently, taking into consideration the top-level requirements (e.g., performance, memory, power, safety, security) that drive the development of current embedded devices, academia and industry have focused on the development of hardware-assisted virtualization. The big players on the processors market have also introduced their commercial off-the-shelf (COTS) virtualization technologies, which are being exploited to implement efficient virtualization solutions [3], [7], [9].

Among existent COTS technologies, ARM Virtualization Extensions (VE) and ARM TrustZone [1] are gaining particular attention due to the ubiquitous presence of ARM-based processors into the embedded sector. Although ARM VE is the specific answer from ARM for virtualization, ARM TrustZone, a hardware security technology, is also seen as an alternative hardware-based form of system virtualization, with important differences from traditional virtualization (e.g., memory is not virtualized but partitioned) [5]. However, it is seen as the only implementable hardware-based approach on those ARM processors where VE are not available. Furthermore, with the recent ARM announcement of introducing TrustZone technology in all Cortex-M and Cortex-R processors series, this technology is gaining momentum as it is being seen as the unique path to break the barrier to the adoption of system virtualization in resource-constrained embedded devices. TrustZone provides two completely separated execution environments as well as an additional privileged execution mode. The non-secure world acts as a virtual machine (VM) under control of a hypervisor running in the secure world side. Many TrustZone-based solutions have been proposed [2], [4], [5], [9] but they fail in providing the ability to support an arbitrary number of guests. They mainly rely on a dual-OS configuration – a real-time operating system (RTOS) side by side with a general purpose operating system (GPOS) – where the number of VMs coincides exactly with the number of virtual environments directly supported by the processor.

This letter goes beyond state-of-the-art, presenting a completely new TrustZone-assisted virtualization solution that allows the execution of an arbitrary number of guest OSes. It demonstrated how multiple guests of an RTOS can efficiently co-exist, completely isolated from each other, on the same hardware platform, and still present reduced performance overhead, memory footprint and deterministic execution. To the best of authors' knowledge, this is the first attempt to run more than one RTOS instance on top of a TrustZone-assisted hypervisor targeting the real-time domain. The main contributions of this letter are: (i) a completely new TrustZone-assisted hypervisor that supports an arbitrary number of VMs; (ii) the applicability of TrustZone-assisted virtualization on real-time applications; and (iii) a real evaluation about the penalties incurred on real-time metrics.

2 ARM TrustZone

TrustZone is a hardware security technology inherent in modern ARM applications processor cores. It virtualizes a physical core as two virtual cores, providing two completely separated execution environments: the *secure* and the *non-secure* worlds. An extra bit – the NS (Non-Secure) bit – indicates in which world the processor is currently executing. To switch between the secure and the non-secure world, a special new secure processor mode, called *monitor mode*, was introduced. To enter the monitor mode, a new privileged instruction was also specified – SMC (*Secure Monitor Call*). The monitor mode can also be enabled by configuring it to handle interrupts and exceptions in the secure side. TrustZone allows system designers to add a TrustZone Address Space Controller (TZASC). This component extends isolation to the memory infrastructure, allowing the partitioning of all or just some portions of memory into distinct segments, each of which can be configured to be used in the secure or non-secure world. The TrustZone-aware Memory Management Unit (MMU) provides two distinct MMU interfaces (each world has a local set of memory address translation tables), and isolation is still available at the cache-level. The Advanced eXtensible Interface (AXI) system bus carries extra control signals to restrict access to the main system bus. Devices can be (static or dynamically) configured as secure or non-secure, which enforces isolation at the device level. The Generic Interrupt Controller (GIC) supports the coexistence of secure and non-secure interrupt sources. It allows the configuration of secure interrupts with a higher priority than the non-secure interrupts, and supports several interrupt models which allow the assignment of Fast Interrupt Requests (FIQs) and Interrupt Requests (IRQs) to secure or non-secure interrupt sources.

3 TrustZone-Assisted Hypervisor

Fig. 1 depicts the complete system architecture for a single-core configuration. The multicore extension and all related questions

- S. Pinto, J. Pereira, T. Gomes, and A. Tavares are with the Department of Centro Algoritmi, University of Minho, Braga 4704-553, Portugal. E-mail: {sandro.pinto, jorge.m.pereira, tiago.m.gomes, adriano.tavares}@algoritmi.uminho.pt.
- M. Ekpanyapong is with the Asian Institute of Technology, Pathumthani 12120, Thailand. E-mail: mongkol@ait.ac.th.

Manuscript received 9 June 2016; revised 3 Oct. 2016; accepted 6 Oct. 2016. Date of publication 12 Oct. 2016; date of current version 5 Jan. 2018.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2016.2617308

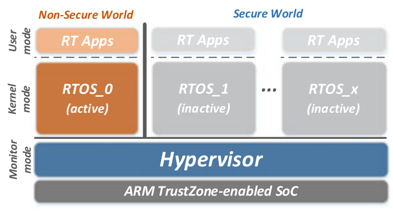


Fig. 1. TrustZone-assisted virtualization architecture.

are out of the scope of this paper. As it can be seen, the hypervisor runs in the most privileged mode of the secure world side, i.e., monitor mode, and has the highest privilege of execution; it is responsible for managing the virtual CPUs, to provide time and spatial isolation, as well as mediate interrupts assigned to inactive guests. Unmodified guest OSes can be encapsulated between the secure and non-secure world sides: the active guest partition runs in the non-secure world side, while inactive guests states are preserved in the secure world side. For the active guest the RTOS runs in the kernel mode, while real-time applications run in user mode.

3.1 Guest Switching

Existing TrustZone-based solutions implement a dual-OS approach, where each guest is running in a different world. In this particular case, the virtual CPU support is guaranteed by the hardware itself. The proposed solution is completely different, since it is able to support an arbitrary number of guest partitions, all of them executed from the non-secure side (one at a time), dictating the sharing of the same virtual processor supported by software. For that reason, the virtual CPU (vCPU data structure) of each guest should be preserved. This vCPU includes the core registers for all processor modes, the CP15 registers and some registers of the GIC, encompassing a total of 55 registers.

3.2 Memory Partition

The existence of a TZASC is a major requirement for the proposed system. The memory segmentation feature provided by the TZASC is exploited to guarantee spatial isolation between guests, by dynamically changing the security state of their memory segments. Only the guest partition currently running has its own memory segment configured as non-secure, while the remaining memory is configured as secure. If the running guest tries to access a secure memory region (e.g., belonging to an inactive guest partition), an exception is automatically triggered and redirected to the hypervisor. Since only one guest can run at a time, there is no possibility for inactive guests (belonging momentously to the secure side) to change the state of another guest. Because there is no second level memory translation, all guest OSes have to cooperate on sharing a single physical memory address space. This imposes a strong requirement to the system: all guests have to know the physical memory segment they can use, 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.

3.3 Device Partition

The proposed solution implements device virtualization adopting a pass-through policy, which means devices are managed directly by guest partitions. To achieve isolation at device level, devices assigned to guest partitions (at build time) are dynamically configured as non-secure or secure, depending on its state (active or inactive). This guarantees an active guest cannot compromise the state of a device belonging to another guest, and if an active guest partition tries to access a secure device, an exception will be automatically triggered and handled by the hypervisor. Shared device access was not taken into consideration and it is outside the scope of this letter.

3.4 Interrupt Management

The hypervisor configures secure interrupts as FIQs, and non-secure interrupts as IRQs. Secure interrupts are redirected to the hypervisor, while non-secure interrupts are redirected to the active guest. When a guest partition is under execution, all interrupts belonging to the active guest are locally and directly managed by the OS without any hypervisor interference. This guarantees no jitter is added to the interrupt latency of an active guest. A more complex problem arises when a critical interrupt for one (inactive) guest partition arrives when a different active guest is running. Different handling mechanisms, with different trade-offs on performance, latency and inter-guest interference were implemented and they are available to be statically and accordingly configured to the partition criticality level. For example, an interrupt can be deferred until the target guest is next scheduled for execution by configuring the respective interrupt sources as disabled FIQs, or immediately force a context switch to the target guest to handle the interrupt by configuring the respective interrupt sources as enabled FIQs.

3.5 Time Management

The implemented hypervisor provides a hybrid Round Robin scheduler that can be configured at build time for strong or weak time isolation. Different guests can be (statically) configured with different time slices accordingly to their level of criticality and application needs. The hypervisor manages two timers: one 32 bit timer for the hypervisor tick and one 64 bit timer to keep track of the wall-clock time. Timers dedicated to the hypervisor are configured as secure devices, i.e., they have higher privilege of execution than timers dedicated to the active guest. This means that despite of what is happening in the active guest, if an interrupt of a timer belonging to the hypervisor is triggered, the hypervisor takes control of the system. Under a weak time isolation scenario, the hypervisor can preempt the execution of an active guest if a critical asynchronous event of an inactive guest occurs. This kind of policy guarantees the timing requirements of the system can be preserved, but the system designer should tune them adequately.

At the partition level, whenever the active guest is executing, timers belonging to the guest are directly managed and updated by the guest OS. The problem lies in how to deal and handle time from inactive guests. Since the proposed solution only supports tickless guest OSes, for inactive guests the hypervisor implements a virtual tickless timekeeping mechanism based on a time-base unit that measures the passage of time (the aforementioned 64 bit timer). Therefore, when a guest is rescheduled, its internal clocks and related data structures are updated with the time elapsed since its previous execution.

4 Evaluation

The hypervisor was tested on a Xilinx ZC702 evaluation board targeting a dual ARM Cortex-A9 running at 600 MHz. As aforementioned, in spite of using a multicore hardware architecture, current implementation only supports a single-core configuration. The evaluation focused on memory footprint (Section 4.1) and performance (Section 4.2). MMU, data and instruction cache and branch prediction support for guests was enabled and disabled. In all test scenarios sotfware stacks were compiled using the ARM Xilinx toolchain with compilation optimizations (−O2). Real-time Onboard Dependable Operating

TABLE 1
Memory Footprint Results (Bytes)

	.text	*.data*	*.bss*	*Total*
Hypervisor	5,568	192	0	**5,760**

System (RODOS), a tickless RTOS already in use in several satellites, was used as guest OS.

4.1 Memory Footprint

To access memory footprint results, the size tool of ARM Xilinx Toolchain was used. Table 1 presents the collected measurements, where boot code and drivers were not taken into consideration. As it can be seen, the memory overhead introduced by the hypervisor-and in fact the trusted computing base (TCB) of the system-is around 6 KB. The main reasons behind this low memory footprint are the hardware support of TrustZone technology for virtualization and the careful design and static configuration of each hypervisor component.

4.2 Performance

The Thread-Metric Benchmark Suite consists of a set of specific benchmarks to evaluate RTOSes performance. The suite comprises seven 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.

For the first part of the experiment the hypervisor was configured with a 10 miliseconds (ms) guest-switching rate. The system was set to run one single guest partition, and the hypervisor scheduler was forced to reschedule the same guest, so that results can translate the full overhead of the complete guest-switching operation. Benchmarks were executed in the native version of RODOS and then compared to the virtualized version. Fig. 2 presents the achieved results, corresponding to the relative performance and variation (as well as the average absolute performance) of 50 collected samples for each benchmark. In both test case scenarios – Figs. 2a and 2b, it is clear that the virtualized version of RODOS only presents a very small performance overhead when compared with its native execution – < 0.5 and < 2.0 percent, respectively –, as well as a variation in the same order of magnitude as the native version. This means the virtualized system remains as deterministic as the native one. In Fig. 2a (caches disabled) the performance overhead is smaller, because the guest-switching operation does not require the execution of several operations such as cleaning and invalidating data and instruction caches.

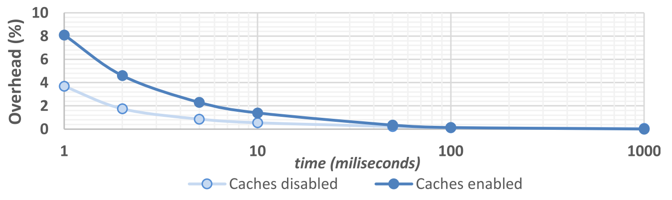


Fig. 3. Correlation between guest-switching rate and performance overhead.

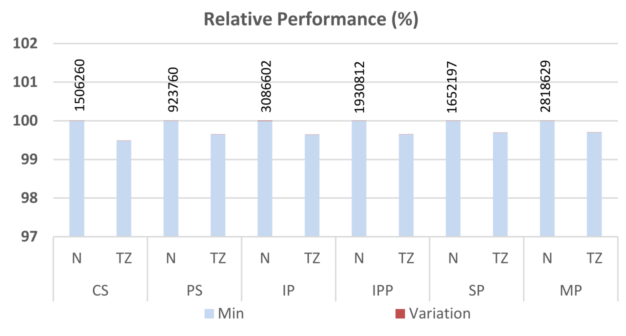
The focus of the second part of the experiment was on how the guest-switching rate correlates with guest performance. Instead of fixing the guest-switching rate in 10 ms, the same experiments were repeated for a guest-switching rate between 1 to 1,000 ms. Fig. 3 shows achieved results, where each mark corresponds to the average performance overhead of measured results for the six benchmarks. As it can be seen, the performance overhead of the virtualized RODOS ranges from 3.69 to 0.06 percent and 8.10 to 0.01 percent with caches disabled and enabled, respectively. When caches are enabled the significant rise of overhead above 5 milliseconds is mainly explained by two reasons: first, as aforementioned, when MMU and caches are enabled, the list of internal activities of the context switch operation is higher; and second, since caches have to be cleaned and invalidated each time a partition is rescheduled, partitions will not take advantage of them until they are filled. Nevertheless, guest-switching rate should be tuned accordingly to the maximum acceptable latency among each guest, otherwise the real-time characteristics of the system can be compromised.

5 Conclusion

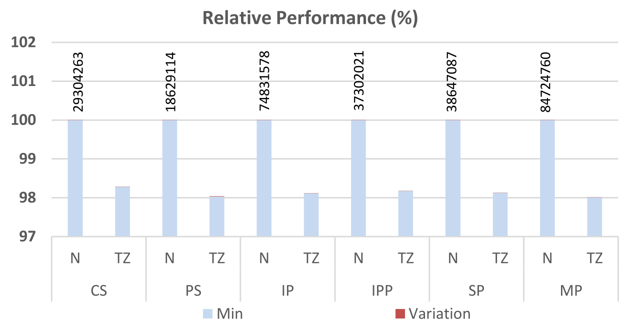
This letter describes how ARM TrustZone technology can be exploited to implement an embedded virtualization solution allowing the coexistence of multiple OSes on the same hardware platform. The hypervisor was implemented on a commercial Xilinx ZC702 board, demonstrating how it is possible to host an arbitrary number of guest OSes on the non-secure world side of TrustZone-enabled processors. Evaluations demonstrated that virtualization overhead has only a slight impact on execution performance, and the hypervisor has a reduced TCB size. This work seems very promising regarding next generation of secure virtualization solutions for resource-constrained embedded devices. Work in the near future will focus on an extensive and exhaustive evaluation of real-time aspects with short-term and long-term tests. Extension for multicore and new generation of low and middle-end ARM platforms is also under scope.

Acknowledgments

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).



(a) Caches disabled



(b) Caches enabled

Fig. 2. Thread-Metric benchmarks results.

REFERENCES

[1] J. Winter, “Trusted computing building blocks for embedded linux-based ARM trustzone platforms,” in *Proc. 3rd ACM Workshop Scalable Trusted Comput.*, ACM, 2008, pp. 21–30.

[2] M. Cereia and I. Bertolotti, “Asymmetric virtualisation for real-time systems,” in *Proc. IEEE Int. Symp. Ind. Electron.*, Jun. 2008, pp. 1680–1685.

[3] C. Dall and J. Nieh, “KVM/ARM: The design and implementation of the Linux ARM hypervisor,” *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 333–348, Feb. 2014.

[4] S. Daniel, S. Honda, and H. Takada, “Dual operating system architecture for real-time embedded systems,” in *Proc. 6th Int. Workshop Operating Syst. Platforms Embedded Real-Time Appl.*, 2010, pp. 6–15.

[5] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig, “ARM TrustZone as a virtualization technique in embedded systems,” *Proc. 12th Real-Time Linux Workshop*, 2010.

[6] G. Heiser, “Virtualizing embedded systems—why bother?” in *Proc. 48th Des. Autom. Conf.*, 2011, pp. 901–905.

[7] C.-T. Liu, K.-C. Chen, and C.-H. Chen, “CASL hypervisor and its virtualization platform,” in *Proc. Int. Symp. Circuits Syst.*, May 2013, pp. 1224–1227.

[8] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, “XtratuM: A hypervisor for safety critical embedded systems,” in *Proc. 11th Real-Time Linux Workshop*, 2009, pp. 263–272.

[9] S. Pinto, et al., “Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone,” in *Proc. IEEE Emerg. Technol. Factory Autom.*, 2014, pp. 1–4.

[10] M. Rosenblum and T. Garfinkel, “Virtual machine monitors: Current technology and future trends,” *IEEE Comput.*, vol. 38, no. 5, pp. 39–47, May 2005.

[11] U. Steinberg and B. Kauer, “NOVA: A microhypervisor-based secure virtualization architecture,” in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 209–222.

[12] A. Tavares, et al., “A customizable and ARINC 653 quasi-compliant hypervisor,” in *IEEE Int. Conf. Ind. Technol.*, Mar. 2012, pp. 140–147.