Lightweight Multicore Virtualization Architecture exploiting ARM TrustZone

S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, A. Tavares
Centro Algoritmi - University of Minho
{sandro.pinto, andre.oliveira, jorge.pereira, jcabral, joao.monteiro, atavares}@dei.uminho.pt

***Abstract*—Virtualization technology is well established in the server and desktop spaces, and has been spreading across embedded system market. This technology allows for the co-existence and execution of multiples operating systems on top of the same hardware platform, with proven technological and economic benefits. Hardware extensions for easing virtualization have been added into several commercial off-the-shelf processors. Among existing technologies, ARM TrustZone is gaining particular attention due to its broadly availability into ARM processors. However, existent TrustZone-assisted virtualization solutions are limited to a dual-guest and single-core configuration, which can lead to the starvation of the non-secure side when the secure world does not yield the processor.**

This work presents the extension of a TrustZone-assisted hypervisor to an asymmetric multi-processing configuration. We describe and demonstrate how to run a general-purpose operating system side-by-side with an real-time operating system in a Xilinx Zynq-based platform, enhanced with a dual ARM Cortex-A9. The achieved results demonstrate that the implemented multicore approach not only completely eliminates starvation, but also increases the general-purpose operating system's performance, especially when the real-time workload is demanding.

***Index Terms*—TrustZone, Virtualization, Multicore, Monitor, Security, Embedded Systems, ARM.**

I. Introduction

Platform virtualization, which enables multiple operating systems (OSes) to run on top of the same hardware platform, is gaining momentum in the embedded systems domain [1]. Nevertheless, embedded virtualization has inherent characteristics which substantially differentiate it from general-purpose or server virtualization, since embedded devices are typically resource and power constrained systems that must respond to events within strict deadlines [2]. While in the server and desktop space virtualization is used for load balancing, service consolidation and power management, in the embedded systems domain virtualization has been primarily employed in order to partition functionality (real-time and non-real-time characteristics), as well as minimizing the design and certification efforts [2], [3].

Over the last few years several embedded virtualization solutions have been proposed in the aerospace [4], [5], [6], automotive [7], [8] and industrial [9] domains. Some of them follow a full-virtualization approach, while others implement paravirtualization [10]. Between both approaches there is a trade-off regarding flexibility and performance: full-virtualization incurs on a higher performance cost, while paravirtualization incurs on a higher design cost. Taking in mind the penalties incurred by traditional virtualization, research and industry have been focused their attention in providing hardware support to assist virtualization. Intel, ARM and Imagination/MIPS introduced their own commercial off-the-shell (COTS) technologies [11], [12], [13], [14]. Among existent COTS technologies, ARM TrustZone [15] is gaining particular attention due to the ubiquitous presence of ARM-based devices in the embedded sector, as well as the supremacy of TrustZone-enabled processors when compared with virtualization-enabled processors. Furthermore, this technology is seen as the only implementable hardware-based approach on those ARM processors, where Virtualization Extensions (VE) are not available.

TrustZone technology virtualizes a physical core as two virtual cores, providing two completely separate execution domains [16]. The non-secure world acts as a virtual machine (VM) under the control of a hypervisor running in the secure world side. Some TrustZone-based solutions for virtualization have been proposed [13], [17], [18], [19], [20], [21], however all of them are limited to a dual-guest and single-core configuration. The lack of scalability, in terms of the number of guests and in terms of the number of supported cores, is the main reason why several researchers still perceive TrustZone as a limited and ill-guided virtualization mechanism [22]. In [23], we demonstrate why this is not necessarily true, and that is possible to run more than two VMs, by multiplexing several guest OSes inside the non-secure world side. In [24], Ngabonziza et al. outline the problem of starvation which can occur in single-core platforms when the real-time operating system (RTOS) does not yield its control of central processing unit (CPU). We strongly believe this problem can be solved by extending existent solutions to a multicore configuration.

This paper presents the extension of lightweight TrustZone-assisted hypervisor (LTZVisor) [21] to an asymmetric multi-processing (AMP) configuration. We demonstrate how to run a general-purpose operating system (GPOS) side-by-side with a real-time operating system (RTOS) on a Xilinx Zynq platform, endowed with a dual-core ARM Cortex-A9. The GPOS runs in one core over the non-secure world side, while the RTOS runs in another core over the completely isolated secure world side. We conducted an extensive set of experiments which demonstrate the multicore approach completely extinguishes starvation, while at the same time presenting several performance advantages when the RTOS has a demanding workload.

II. ARM TrustZone

TrustZone technology is a set of hardware security extensions, introduced with ARMv6K, in all ARM Cortex-A processors [16]. Recently, ARM extended TrustZone also to the Cortex-M processor family, but with slight differences. In the remainder of this section, when describing TrustZone, the focus will be on the specificities of TrustZone for the application processors.

The TrustZone hardware architecture virtualizes a physical core as two virtual cores, providing two completely separated execution environments: the *secure* and the *non-secure* worlds. A new 33rd processor bit, the *Non-Secure* (*NS*) bit, indicates in which world the processor is currently executing. To 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. The TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) extend TrustZone security to the memory infrastructure. TZASC can partition the DRAM into different memory regions: this hardware controller has a programming interface, accessible only from the secure side, that can be used to configure a specific memory region as secure or non-secure. By default, secure world applications can access normal world memory but the reverse is not possible. TZMA provides similar functionality but for off-chip ROM or SRAM. The TrustZone-aware Memory Management Unit (MMU) provides two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. The isolation is still available at the cache-level, because processor's caches have been extended with an additional tag which signals in which state the processor accesses the memory. The AXI (Advanced eXtensible Interface) system bus carries extra control signals to restrict access to the main system bus. System devices can be dynamically configured as secure or non-secure through the TrustZone Protection Controller (TZPC). To support the robust management of secure and non-secure interrupts, the Generic Interrupt Controller (GIC) provides both secure and non-secure prioritized interrupt sources.

III. LTZVisor-AMP: Design

LTZVisor-AMP implements a lightweight TrustZone-assisted hypervisor with asymmetric multi-processing support. One core runs in the secure world and is responsible for hosting the privileged software, while the other core runs in the non-secure world and is responsible for hosting the non-privileged software. This means a one-to-one mapping between guest OSes, cores, and the virtual states supported by the processor exists. Fig. 1 depicts the proposed virtualization architecture based on three main software components: the hypervisor, the real-time guest OS, and the general-purpose guest OS.

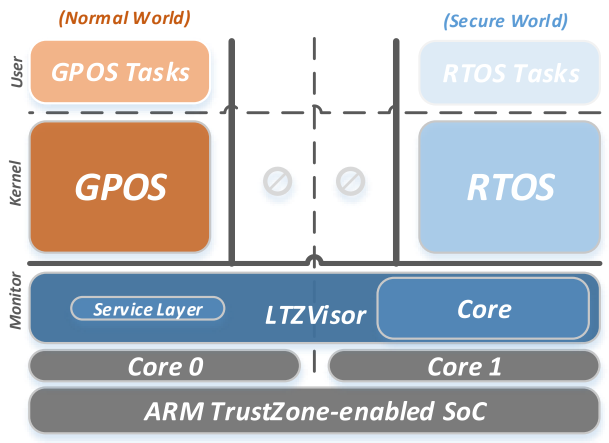

Fig. 1: LTZVisor-AMP Architecture

LTZVisor-AMP runs in the highest privileged processor mode, i.e., in the monitor mode. The hypervisor, following an AMP schema, is split into two parts. The core of the hypervisor is the master of the system and is responsible for the main tasks: configuring memory, interrupts and devices assigned to each guest OS, as well as guaranteeing runtime support for inter-partition communication. Whereas, the service layer is responsible for guaranteeing minimal privileged support for exception handling and forwarding, as well as for inter-partition communication. The real-time guest OS kernel runs in the supervisor mode of the secure world side and has assigned a dedicated core. This VM must have a small trusted computing base (TCB), because when the processor state is secure it has full view over the non-secure world side. As such, the privileged guest code can interfere with the other guest OS, by accessing or modifying its state or the state of its resources (memory or memory mapped devices). The general-purpose guest OS kernel runs in the supervisor mode of the non-secure world side, and has also assigned a dedicated core. The software running on the secure world side is completely isolated from the privileged software running on the non-secure world side. When the processor is operating in a privileged mode but not in the secure state, it cannot access or modify any state information belonging to the secure world. Any attempt from the non-secure guest OS to access any resource of the secure world side immediately triggers an exception to the service layer, which will be responsible for forwarding it to the core of the hypervisor.

IV. LTZVisor-AMP: Implementation

LTZVisor-AMP is a lightweight TrustZone-assisted hypervisor which implements support for a supervised asymmetric multi-processing configuration. This section describes the details behind its implementation.

A. Guest Management

LTZVisor-AMP is responsible for managing guest OSes in an asymmetric fashion, by assigning them to run over individual cores at boot time. Since the hardware in which our system was deployed (Zedboard) is endowed with a dual-core Cortex-A9, an one-to-one mapping between guests, worlds and cores was implemented. This means the general-purpose guest OS, running on the non-secure world side, is assigned to one core, while the real-time guest OS, running on the secure side, is assigned to another core. For easing the development and to avoid modifications to the Linux kernel, the general-purpose guest OS runs over the primary core (core0), because by default the supported version of Xilinx Linux is not ready for executing on the secondary core. The flexibility to swap guests across cores will be addressed in the future.

B. Memory Partition

TrustZone-enabled SoCs (which are not VE-enabled) only provide MMU support for single-level address translation. However, the TZASC provides mechanisms for partitioning memory into different segments. This memory segmentation feature can, therefore, be used to ensure spatial isolation between guest OSes. It is basically done by adequately configuring the security state of the memory segments of respective partitions. The general-purpose guest OS must have its own memory segment(s) configured as non-secure, while the real-time guest OS, as well as the hypervisor, as secure. If the non-secure guest OS tries to access a secure memory region, an exception is automatically triggered and redirected to the hypervisor. Memory segments can be configured with a specific granularity, which can be different from platform to platform. In the hardware platform under which our system was deployed (Zynq), memory regions can be configured with a granularity of 64MB, and both cores share the same memory model. LTZVisor-AMP configures the security state of the memory according to the following map: the general-purpose guest OS uses the first seven memory segments, corresponding to a total of 448MB of non-secure memory; the hypervisor and the real-time guest OS use only the last available memory segment; finally the remainder of the 32-bit memory address space is not accessible because Zedboard is only endowed with a 512MB DDR3 memory.

C. MMU and Cache Management

TrustZone-enabled SoCs provide two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. The same kind of isolation is still available at cache-level. The MMU and Level 1 (L1) cache exist for each individual core, which mean each guest OS, running in an AMP schema, has a dedicated set of hardware. Level 2 (L2) cache, in turn, is shared among both cores. However, the existence of secure and non-secure cache entries alleviates L2 cache management, because the cache coherence between both guest OSes is guaranteed by the TrustZone-hardware itself.

D. Device Partition

LTZVisor-AMP implements device virtualization by adopting a pass-through policy, which means devices are managed directly by guest partitions. To achieve isolation at device level, devices assigned to the GPOS and the RTOS partitions (at build time) are statically configured as non-secure or secure, respectively. Isolation provided by means of the TZPC guarantees the non-secure guest (GPOS) cannot compromise the state of a device belonging to the secure guest (RTOS), and if the non-secure GPOS tries to access a secure device, an exception will be automatically triggered and handled by the hypervisor service layer. The granularity of the isolation is per-device, but it also depends from platform to platform. In Zynq-based devices, which is the case of the ZedBoard platform, both processing system (PS) peripherals and programmable logic (PL) custom peripherals can be configured as secure or non-secure. Shared device access was not taken into consideration and it is out of the scope of this paper.

E. Interrupt Management

The TrustZone-aware GIC supports the coexistence of secure and non-secure interrupt sources. The GIC supports several interrupt models, allowing for the configuration of IRQs and FIQs as secure or non-secure interrupt sources. LTZVisor-AMP configures secure interrupts as FIQs, and non-secure interrupts as IRQs. This is the suggested model by ARM, which makes sense in our system configuration, since FIQs (assigned to the real-time environment) have a smaller interrupt latency than IRQs. The core of the hypervisor (running on core1) is responsible for configuring the GIC distributor as well as its individual interface. The service layer of the hypervisor is responsible for configuring just the individual interface of core0. Secure interrupts (i.e., FIQs) arriving to core1 are redirected to the RTOS without any hypervisor interference. Secure interrupts arriving to core0 are redirected to the service layer, which will forward them to the core running the hypervisor. According to our design, only secure (software generated) interrupts are supposed to be triggered on core0 when inter-partition communication is needed. If an IRQ arises (on core1) while the RTOS is executing, it does not affect the expected RTOS behaviour. This prevents any kind of denial-of-service attack over the real-time environment.

F. Time Management

Temporal isolation in virtualized systems is typically achieved using two levels of timing: at hypervisor level and at partition level. For the partition level, hypervisors typically provide timing services which allow for guests to have notion of virtual or real time. The implementation of virtual or real notion of the passage of the time introduces different levels of complexity, and when virtualization targets real-time environments, timekeeping issues still prevail as an open question. LTZVisor-AMP provides a distinctive time management implementation. Due to its dual-OS nature, the hypervisor dedicates one independent timing unit for each guest OS. The RTOS uses the Triple Timer Counter (TTC) 0, while the GPOS uses the ARM global timer. It is fundamental that the hypervisor configures the global timer as a non-secure device, otherwise an exception will be triggered on the first attempt to access it. This specific time management implementation

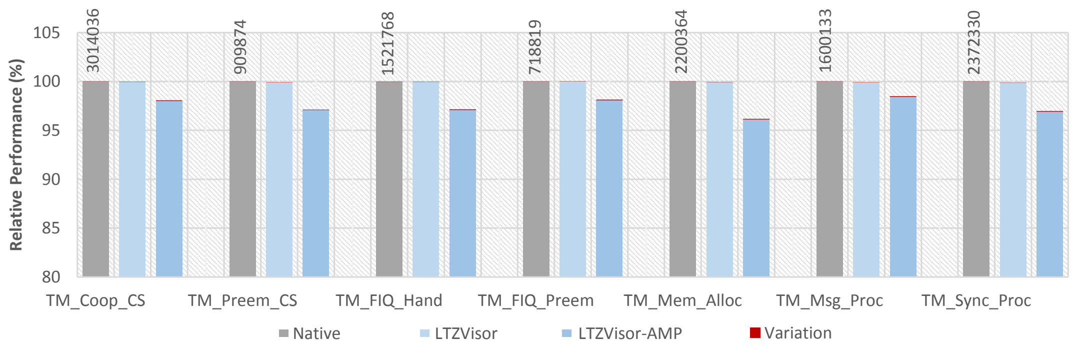

Fig. 2: Thread-Metric benchmarks results

ensures that each VM has its timing structures updated at all times. The RTOS does not miss any system-tick interrupt, and the GPOS is completely aware of the real passage of time.

G. *Inter-Partition Communication*

LTZVisor-AMP uses the standardized VirtIO interface [25] to provide a transparent virtual mechanism for implementing communication between the two different guest OSes. It implements an adaptation of the Remote Processor Messaging (RPMsg) API from the Texas Instrument and OpenAMP group to a supervised multicore architecture. The implementation from Texas provides the foundation for implementing communication on top of a GPOS, while the implementation from OpenAMP provides the foundation for a bare-metal approach. The adopted architecture implements different data and event paths, which promotes asynchronous communication. The data path is defined by a shared block of memory, configured as non-secure. The event path is defined by software generated interrupts (SGIs) routed through the hypervisor. This mechanism is based on requests from guest OSes to the hypervisor, via the SMC instruction. All requests are stored in a circular buffer. During each partition switch, LTZVisor triggers SGIs to the respective guest OSes, enabling asynchronous notifications. More details regarding the inter-partition communication mechanism are out of the scope of this paper, and it will be individually addressed in a future publication.

V. Evaluation

LTZVisor-AMP was evaluated on a Zedboard targeting a dual ARM Cortex-A9 running at 667MHz. Our evaluation was split into three different test case scenarios: (i) RTOS performance (Section V-A), (ii) GPOS performance when the RTOS is idle, and (iii) GPOS performance when the RTOS has different workloads (Section V-B). LTZVisor and both OS partitions were compiled using the ARM Xilinx toolchain, with compilation optimizations disabled (-O0). The idea of presenting results with compilation optimizations disabled is because this configuration outputs the worst case scenario. Xilinx Linux (v4.0.0) and FreeRTOS (v7.0.2) were used as non-secure and secure partitions, respectively.

A. *RTOS performance*

The Thread-Metric Benchmark Suite consists of a set of benchmarks properly conceived to evaluate RTOSes performance. The suite comprises 7 benchmarks, evaluating the most common RTOS services and interrupt processing. Each benchmark outputs a counter value, representing the RTOS impact on the running application: the higher the value, the smaller the impact.

Benchmarks were executed in three different system configurations: the native FreeRTOS; the virtualized FreeRTOS running on top of the hypervisor in a single-core configuration (LTZVisor); and the virtualized FreeRTOS running on top of the hypervisor in a multicore configuration (LTZVisor-AMP). The native version of FreeRTOS runs on top of the core0, likewise the virtualized FreeRTOS partition running on top of LTZVisor. The FreeRTOS running on top of LTZVisor-AMP distinguishes from the other system configurations, because it executes in core1. Fig. 2 presents the achieved results, corresponding to the average relative performance (as well as the average absolute performance) of 1000 collected samples for each benchmark. Each sample reflects the benchmark score for a 30 seconds execution time, encompassing a total execution time of 500 minutes for each benchmark. In accordance with Fig. 2 the overhead introduced by the virtualization layer in single-core configuration is null, while in the multicore configuration it presents an average performance degradation of 3%. For the single-core configuration the null overhead is perfectly understandable because, despite both OSes are sharing the same core, once FreeRTOS starts running real-time tasks, it will never be interrupted by the GPOS or even by the hypervisor (asymmetric scheduling [21]). For the multicore configuration, we strongly believe the reason behind the performance degradation is related with some architectural or micro-architectural implication of running the real-time guest OS over the secondary core. Notwithstanding, an in-depth investigation about the motivational reasons behind this phenomenon will be carried out in the near future.

B. *GPOS performance*

LMBench [26] is a widely used suite of micro-benchmarks that measure a variety of important aspects of system per-

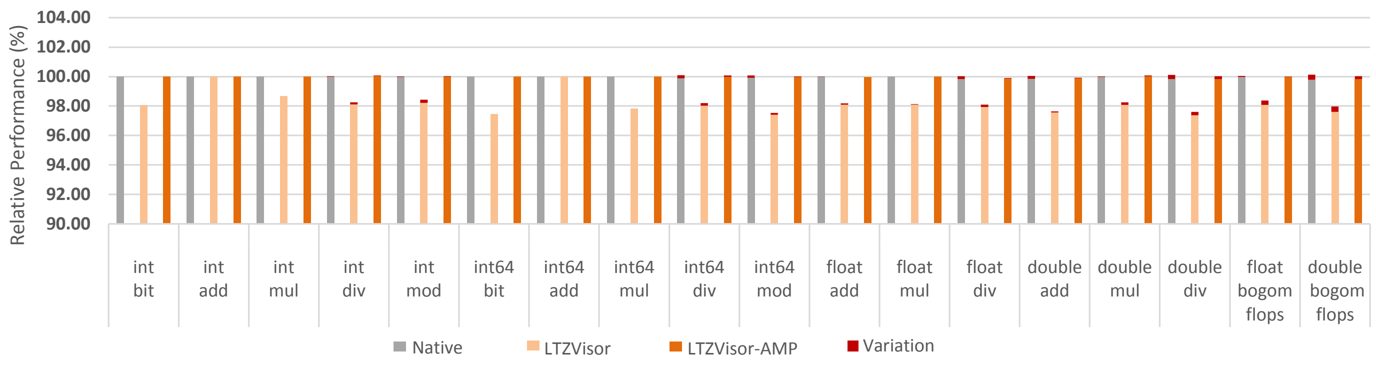

Fig. 3: LMBench arithmetic operations latency (lat_ops) benchmark results

formance, such as latency and bandwidth. The LMBench 3.0 suite includes more than forty micro-benchmarks within three different categories: *bandwith*, *latency*, and *other*. We focus our evaluation on the arithmetic operations latency micro-benchmark (***lat_ops***), in order to evaluate general CPU performance (VFP and Neon are disabled). We split the GPOS evaluation into two different experiments: firstly, we evaluate the performance overhead, from the general-purpose guest OS perspective, for a fixed guest-switching rate and when the real-time guest OS is idle; then we repeat the performance overhead evaluation, but for different guest-switching rates and real-time workloads.

*1) **RTOS idle**:* For the first part of the experiment, FreeRTOS was configured with a 1 millisecond tick rate (i.e., guest-switching rate) and no real-time tasks were added to the system, which represents a null workload on the RTOS side (FreeRTOS will be infinitely executing the idle task). We ran the micro-benchmarks in the native version of Linux (N) and compared it against the virtualized version running on top of LTZVisor and LTZVisor-AMP. L1 cache and branch prediction were enabled for both test case scenarios. For each micro-benchmark we performed 100 consecutive experiments. For each experiment the micro-benchmark was configured for 10 warm-ups and 1000 repetitions (-W 10 -N 1000). Presented results correspond to the average relative performance and variation (as well as the average absolute performance). Fig. 3 presents the achieved results for the arithmetic operations latency benchmark. In accordance with Fig. 3, the average performance degradation introduced by the virtualization layer in single-core configuration is around 2%, while in the multicore is null. For the single-core configuration the 2% overhead is justified by the context-switch time: since both OSes share the same core, for every single tick the general-purpose guest OS is preempted and the real-time guest OS is resumed to verify if any real-time task is ready-to-run. For the multicore configuration, since each OS runs independently in each core, the overhead introduced by the context-switch is non-existent. Once the GPOS is running over the primary core, the particular phenomenon highlighted in Section V-A was not observed.

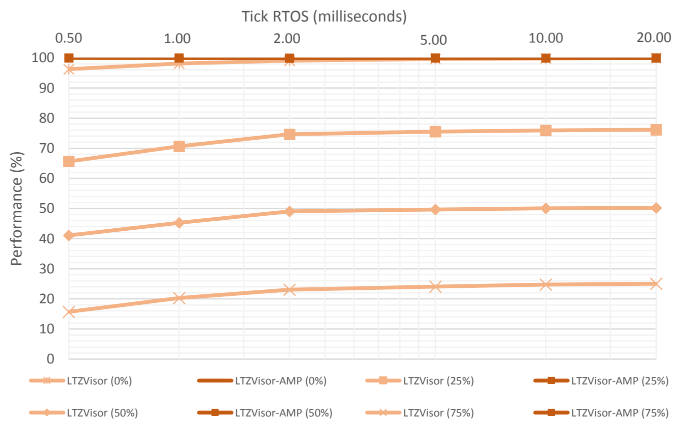

Fig. 4: LMBench arithmetic operations latency results: performance overhead vs RTOS tick, for different workloads

*2) **RTOS with different workloads**:* For the second part of the experiment, instead of fixing the FreeRTOS tick with a 1 millisecond rate, the same experiments were repeated for six different guest-switching rates within a time window ranging from 500 microseconds to 20 milliseconds. Furthermore, one real-time task was added to the system, in order to simulate different workloads on the real-time environment. We ran the arithmetic operations latency benchmark in the virtualized version of Linux running on top of LTZVisor and LTZVisor-AMP. L1 cache and branch prediction were enabled for all test case scenarios. For each micro-benchmark we performed 100 consecutive experiments, and for each experiment the micro-benchmark was configured for 10 warm-ups and 1000 repetitions (-W 10 -N 1000). Presented results correspond to the average performance overhead of measured results regarding the 18 (arithmetic) micro-benchmarks. Six different configurations were setup, corresponding to a tick rate of 0.5, 1, 2, 5, 10 and 20 milliseconds. Eight different tests were carried out, because four different workloads were setup: 0, 25, 50 and 75%. This means for a specific RTOS tick rate, the task will be consuming the CPU for the respective percentage of time. Fig. 4 presents the achieved results. For the single-core configuration it is clear the impact on performance when the RTOS tick is shortened, as well as when the real-time workload is increased. For example, when the real-time workload takes the CPU for 75% of the time of the system tick, the assessed performance ranges from 15.71% to 24.92% for a guest-switching rate of 0.5 and 20 milliseconds, respectively. This can lead, ultimately, to a complete starvation of the non-secure side, just in the case when the real-time environment does not release the CPU. On the other hand, for the multicore configuration the four lines are overlapped. This means the performance of the general-purpose guest OS is the same as the native system, regardless of the RTOS tick and its workload.

VI. Conclusion

In this paper we presented LTZVisor-AMP as an extension of a TrustZone-assisted hypervisor for a supervised asymmetric multi-processing configuration. A limitation of the majority of existent TrustZone-assisted virtualization solutions is related to the asymmetric scheduling policy. This design principle dictates the GPOS can run only during the idle times of the RTOS, which in a single-core configuration can lead to the starvation of the non-secure side (GPOS). Presented solution addresses and eliminates this problem, while simultaneously showing performance advantages for the GPOS when the real-time environment has different workloads.

Future work will mainly focus on investigating means of providing a higher degree of flexibility and scalability for the implemented AMP configuration. Current solution is limited to the one-to-one mapping between cores, guests and worlds. While this solution can support several applications under dual-core hardware platforms, it is not yet ready to scale for quad-core platforms. We will also conduct an in-depth investigation to find the motivational reasons behind the observed degradation of performance in running the RTOS over the secondary core (core1). Finally, the porting of LTZVisor-AMP for other TrustZone-enabled multicore platforms, as well as its extension for symmetric multi-processing (SMP) configuration and heterogeneous platforms are also at the top of our goals.

VII. Acknowledgements

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT - Fundação para a Ciência e Tecnologia - within the Project Scope: UID/CEC/00319/2013. Sandro Pinto is supported by FCT PhD grant SFRH/BD/91530/2012.

References

[1] G. Heiser, "Virtualizing Embedded Systems: Why Bother?" in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. ACM, 2011, pp. 901–905.

[2] ——, "The Role of Virtualization in Embedded Systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, ser. IIES '08. New York, NY, USA: ACM, 2008, pp. 11–16.

[3] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, May 2005.

[4] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach," in *2010 European Dependable Computing Conference*, April 2010, pp. 67–72.

[5] H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, and H. W. Jin, "Full virtualizing micro hypervisor for spacecraft flight computer," in *31st Digital Avionics Systems Conference*, Oct 2012, pp. 6C5–1–6C5–9.

[6] A. Tavares, A. Didimo, S. Montenegro, T. Gomes, J. Cabral, P. Cardoso, and M. Ekpanyapong, "RodosVisor - an object-oriented and customizable hypervisor: The CPU virtualization," *IFAC Proceedings Volumes*, vol. 45, no. 4, pp. 200 – 205, 2012.

[7] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive E/E-systems," in *IEEE International Symposium on Industrial Embedded Systems*, June 2014, pp. 189–198.

[8] C. Lee, S. W. Kim, and C. Yoo, "VADI: GPU Virtualization for an Automotive Platform," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 277–290, Feb 2016.

[9] Y. Kaneko, T. Ito, and T. Hara, "A measurement study on virtualization overhead for applications of industrial automation systems," in *IEEE International Conference on Emerging Technologies and Factory Automation*, Sept 2016, pp. 1–8.

[10] J. Shuja, A. Gani, K. Bilal, A. U. R. Khan, S. A. Madani, S. U. Khan, and A. Y. Zomaya, "A survey of mobile device virtualization: Taxonomy and state of the art," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 1:1–1:36, Apr. 2016.

[11] U. Steinberg and B. Kauer, "NOVA: A Microhypervisor-based Secure Virtualization Architecture," in *European Conference on Computer Systems*, ser. EuroSys '10. ACM, 2010, pp. 209–222.

[12] C. Dall and J. Nieh, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor," *SIGPLAN Not.*, vol. 49, no. 4, pp. 333–348, Feb. 2014.

[13] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone," in *IEEE International Conference on Emerging Technologies and Factory Automation*, Sept 2014, pp. 1–4.

[14] S. Zampiva, C. Moratelli, and F. Hessel, "A hypervisor approach with real-time support to the MIPS M5150 processor," in *International Symposium on Quality Electronic Design*, March 2015, pp. 495–501.

[15] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves, "Implementing Embedded Security on Dual-Virtual-CPU Systems," *IEEE Design Test of Computers*, vol. 24, no. 6, pp. 582–591, Nov 2007.

[16] T. Alves and D. Felton, "TrustZone: Integrated Hardware and Software Security," *Technology In-Depth*, vol. 3, no. 4, pp. 18–24, 2004.

[17] J. Winter, "Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms," in *ACM Workshop on Scalable Trusted Computing*. ACM, 2008, pp. 21–30.

[18] M. Cereia and I. Bertolotti, "Virtual Machines for Distributed Real-time Systems," *Computer Standards & Interfaces*, vol. 31, no. 1, pp. 30–39, Jan. 2009.

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

[20] S. Oh, K. Koh, C. Kim, K. Kim, and S. Kim, "Acceleration of dual OS virtualization in embedded systems," in *Int. Conference on Computing and Convergence Technology*, Dec 2012, pp. 1098–1101.

[21] 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)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 76, 2017, pp. 4:1–4:22.

[22] P. Varanasi and G. Heiser, "Hardware-supported Virtualization on ARM," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys '11. ACM, 2011, pp. 11:1–11:5.

[23] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a TrustZone-assisted Hypervisor for Real Time Embedded Systems," *IEEE Computer Architecture Letters*, 2016.

[24] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases," in *Int. Conference on Collaboration and Internet Computing*, Nov 2016, pp. 445–451.

[25] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008.

[26] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.