TZ-VirtIO: enabling standardized inter-partition communication in a TrustZone-assisted hypervisor

A. Oliveira, J. Martins, J. Cabral*, A. Tavares*, S. Pinto*
Centro Algoritmi - University of Minho
{a65319, a60141}@alunos.uminho.pt, *{jcabral, atavares, sandro.pinto}@dei.uminho.pt

***Abstract*—Virtualization technology allows the coexistence and execution of multiple operating systems on top of the same hardware platform. In the embedded systems domain, virtualization has been focused on the isolation of critical requirements like real-time, security and safety from non-critical characteristics. The strict confinement of guest partitions typically provided by virtualization does not suit the modular and inter-cooperative nature of embedded systems. The need for inter-partition communication has been addressed by multiple virtualization solutions, either to enable guest-level device para-virtualization or to ensure increased flexibility regarding cooperative partitions. However, the majority of existing approaches follow an ad hoc approach with limited to none applicability outside their solution's scope.**

This paper presents TZ-VirtIO, an asynchronous standardized inter-partition communication (IPC) mechanism on top of a TrustZone-assisted dual-OS hypervisor (LTZVisor). The implemented IPC uses the standard VirtIO transport layer. The experiments conducted on a physical platform show a scalable, high-bandwidth and low-overhead solution for both single-core and multi-core architectures.

***Index Terms*—TrustZone, Virtualization, Communication, Monitor, Security, VirtIO, ARM.**

I. Introduction

Virtualization technology has been used as one of the mainstream approaches to allow the coexistence of multiple operating systems (OSes) on the same hardware platform [1]. In the embedded systems field, virtualization has been focused on the isolation of critical requirements from non-critical characteristics. Embedded industrial, automotive and medical applications, need to guarantee the deadlines of real-time tasks and their security, while at the same time, integrating rich environments for monitoring and network purposes [2], [3].

Despite all the advantages of virtualization, the rigid isolation of traditional virtualized environments is not particularly suitable for meeting embedded industries requirements. The strict partition confinement interferes with the embedded systems modular and inter-cooperative nature. Hence, embedded systems virtualization disrupts from traditional virtualization given its need for low-overhead, high-bandwidth and secure communication channels bridging guest partitions [4]. The communication enables cooperation of the embedded subsystems, allowing for a possible balanced workload and cooperation between different and heterogeneous OS classes.

Several virtualization solutions such as Xen [5], SASP [6], SafeG [7], Jailhouse [8], and BlueVisor [9] have implemented inter-partition communication mechanisms, but all of them follow an ad hoc implementation. VirtIO [10], a device abstraction composed by a set of arrays and descriptors, was proposed as *de-facto* standard for virtual I/O devices, and adopted by KVM [11] and lguest [12] for enabling device para-virtualization. More recently, VirtIO, as a result of its abstraction capabilities and efficient performance, was adopted as the transport abstraction layer for several communication mechanisms [13] [14], including Texas' RPMsg. The pair RPMsg/VirtIO, a multi-core communication mechanism, was later included on popular multi-core heterogeneous frameworks such as OpenAMP and MEMF [15].

This paper presents the implementation of a standardized inter-partition/inter-core communication mechanism in a TrustZone-assisted hypervisor (LTZVisor [3], [16]). The communication mechanism uses the standard VirtIO as the transport abstract layer. The implemented mechanism features an asynchronous inter-partition communication supporting both single- and multi-core architectures. The conducted experiments demonstrate the feasibility of the proposed approach, presenting promising results in both system configurations (single- and multi-core). The proposed solution distinguishes from related work by implementing a standardized communication interface on a TrustZone-assisted hypervisor, while keeping the system's real-time capabilities.

II. Background

A. ARM TrustZone

ARM TrustZone is a security technology deployed on current system-on-chip (SoC). The technology has been available on ARM application processors (Cortex-A) for several years and has recently been extended to cover the new generation of ARM microcontrollers (Cortex-M). TrustZone for ARMv8-M has the same high-level features as TrustZone for Cortex-A series, but its design is optimized for microcontrollers and low-power applications. The remainder of this section focus on the specificities of TrustZone for the application processors.

At the heart of TrustZone approach is the concept of secure and non-secure worlds. These two virtual environments are completely hardware isolated, with non-secure software blocked from directly accessing secure world resources. The current world in which the processor runs is determined by the Non-Secure (NS) bit, and is propagated over the memory and peripheral buses. The transition between the secure and non-secure worlds can be bridged via the secure monitor, which runs at the highest privileged processor mode (monitor mode). To enter the monitor mode, a new privileged instruction, SMC

(Secure Monitor Call), was specified. The monitor can also be triggered by configuring it to handle exceptions in the secure world. Still at the processor level, to ensure a strong isolation between secure and non-secure states, some special registers are banked, and some security critical registers are totally unavailable to the non-secure world. The memory infrastructure outside the core can also be partitioned into the two worlds through the TrustZone Address Space Controller (TZASC). DRAM can be partitioned into distinct memory regions, each of which can be configured to be used in either world. Furthermore, the processor provides two virtual Memory Management Units (MMUs), and isolation is also present at the cache-level. System peripherals can also be configured as secure or non-secure through the TrustZone Protection Controller (TZPC). The Generic Interrupt Controller (GIC) provides both secure and non-secure interrupt sources, while allowing the configuration of secure interrupts with a higher priority than the non-secure interrupts.

B. *LTZVisor*

LTZVisor [3], from TZVisor Project[1], is an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticality systems. LTZVisor implements the classical dual-guest OS configuration: the secure world hosts the real-time operating system (RTOS) and the hypervisor, while the non-secure world is assigned to the general-purpose operating system (GPOS) (Fig. 1). LTZVisor provides support for a single-core configuration, and LTZVisor-AMP [16] implements support for a supervised asymmetric multiprocessing configuration.

The hypervisor runs in the highest privileged processor mode, i.e. the monitor mode. It is responsible for enforcing the inter-partition isolation, through several configurations such as memory and device partition, as well as exception handling. The RTOS kernel runs in the supervisor mode of the secure partition. Therefore, it has full view over the non-secure privileged software, which means it is part of the trusted computing base (TCB) and necessarily must have a small footprint. The GPOS kernel runs in the supervisor mode of the non-secure partition. The secure partition is completely isolated from the non-secure partition, and any attempt from the non-secure guest OS to access any of the secure world resources will immediately trigger an exception to be handled by the hypervisor. In LTZVisor-AMP, each partition has an assigned dedicated core following a one-to-one mapping between guest OSes, cores and processor states. The hypervisor runs in the secure world. Its main features run on the secure core, whereas a service layer in the non-secure core provides inter-partition communication support and exception handling for the GPOS.

Spatial isolation between guest OSes is enforced by the TrustZone-aware hardware. The hypervisor uses the TZASC to configure the security state of the memory blocks of the respective partition. LTZVisor follows the suggested ARM model, assigning *fast interrupt requests* (FIQs) to the secure partition and *interrupt requests* (IRQs) to the non-secure partition. In the single-core architecture, while executing in the non-secure world, FIQs are set to be handled by the hypervisor. The hypervisor will then trigger a context-switch to the secure partition, resulting in minimal interrupt latencies for the real-time OS.

[1]http://www.tzvisor.org/

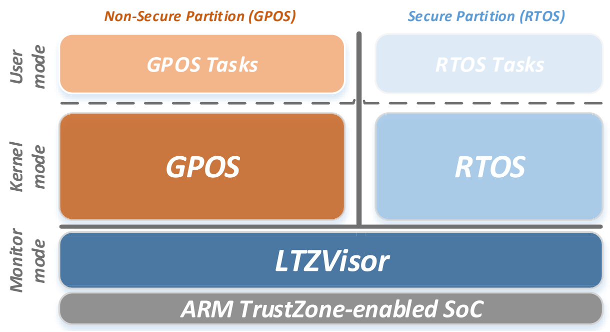

Fig. 1: LTZVisor generic architecture

C. *VirtIO*

VirtIO [10] emerged as an attempt to become the *de-facto* standard for virtual I/O devices in para-virtualized hypervisors. VirtIO is an abstract layer providing a set of front-end and back-end para-virtualization drivers in order to alleviate the complexity of emulating a device. Initially exclusively intended for Linux para-virtualization, it was later extended to bare-metal/RTOS within the OpenAMP project scope. VirtIO uses a set of arrays and descriptors to implement a virtual queue which encapsulates the control and shared data. Its efficiency results from its ring buffer structure which eliminates the need for mutual exclusion primitives or unnecessary data copies. The virtual queue conceptually binds the front-end and back-end drivers.

Amongst the VirtIO supported devices is the RPMsg device. The RPMsg device is not designed for para-virtualization; however, it takes advantage of the underlying transport abstraction layer, the virtual queues, used as its communication transport layer. The intended back-end and front-end drivers are converted in master and slave communication drivers, respectively. RPMsg communication was proposed under the scope of heterogeneous inter-core communication.

RPMsg defines a point-to-point non-persistent asynchronous communication. Each point-to-point communication is defined as a RPMsg channel. Each channel may contain several endpoints, with their own call backs, enabling several distinct communication applications within the same channel. Each RPMsg VirtIO device uses two sets of virtual queues, which are converted into unidirectional transport channels with specific and individual interrupts. Each channel has defined two sources of interrupts: one for message transmission notification; and the other for buffer freed notification. In order to enable unordered messages between different endpoints, the non-persistent communication requires the use of a header in each transaction.

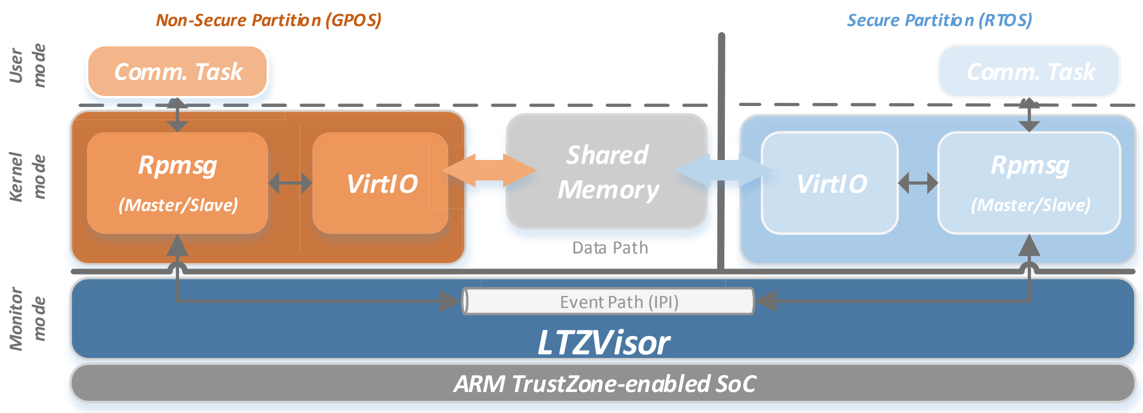

Fig. 2: TZ-VirtIO generic architecture

III. TZ-VirtIO

Fig. 2 depicts the proposed TZ-VirtIO communication mechanism implemention. TZ-VirtIO encompasses: 1) the RPMsg and VirtIO driver modules, 2) optional user-space communication tasks and 3) isolated data and event paths. The latter promotes unblocking asynchronous communication, essential for the timing requirements of the secure guest. It features a shared memory block at kernel-level and an event path routed through the hypervisor. An adaptation of the inter-core RPMsg defined communication protocol to an inter-partition supervised architecture was performed.

A. Data Path

The data path is defined by a shared block of memory configured as non-secure. The shared memory block is configured at compile-time and it is statically allocated at boot-time. The hypervisor sets this block memory as non-secure through the TZASC so both the non-secure and secure partitions can access it. Both guests allocate the shared memory at kernel-level. Linux, the GPOS used on the deployed system, assigns the shared memory block to the VirtIO device through the contiguous memory allocator (CMA) driver.

The RTOS should be designated as the communication master, even though both modes are available in both guests' communication mechanism. The communication master is responsible for the management of the shared memory at boot-time and consequently the initial organization of the VirtIO buffers, a task which should, therefore, be handled by the trusted privileged software, i.e. the RTOS.

B. Event Path

Existing TrustZone-assisted trusted execution environments (TEE), such as IIoTEED [17] and Linaro's OP-TEE[2], rely on the use of the SMC instruction to implement their event path. Typically, this approach implements an RPC schema, which requires a prompt world-switch and an immediate handling. This method does not seem suitable under the LTZVisor scope, because it would lead to considerable inter-guest interference, lack of predictability and even jeopardization of the real-time guarantees.

TZ-VirtIO explores the use of Inter-Processor Interrupts (IPIs) to implement the inter-partition notifications of RPMsg. These interrupts cannot be issued natively as the TrustZone hardware blocks the non-secure world from interrupting the secure world. Moreover, on the single-core configuration, a direct trigger of the interrupt would cause a self-interrupt on the same guest generating it. For these reasons the interrupts are routed through the hypervisor. The event path routing imposes a slight increase in the partition-switching time, however, it also guarantees the reliability of the communication as the hypervisor has control over every transaction.

The direct triggering of an IPI was replaced by an interrupt request to the hypervisor via the SMC instruction, forcing an immediate switch to the monitor mode. The hypervisor then follows one of two approaches, depending of the system configuration (single or multi-core): 1) stores the interrupt request in a circular buffer and during the next context-switch triggers a previously stored IPI to the respective guest OS or 2) it immediately generates the IPI. The first approach is valid for the single-core configuration while the second one can only be used on the asymmetric multiprocessing schema. Regardless of the configuration, the request interface remains the same, providing an architecture configuration abstraction at guest-level of the event path.

In the single-core configuration, the interrupt requests are stored in a buffer, part of the targeted guest virtual machine context block (VMCB). The storing mechanism does not require a prompt context-switch, protecting the real-time requirements of the RTOS. The interrupt will only be triggered during the next scheduled context-switch. A limit of one interrupt per world-switch was imposed in order to lower the communication interference in the partition-switching time. From the two interrupts available per channel, the message notification interrupt has the highest priority in the storage buffer. The natively supported burst mode (one interrupt for several messages) of RPMsg allows the aforementioned imposed limit without bottlenecking the communication performance. The storage mechanism chosen, a circular buffer which follows a

[2]https://www.linaro.org/initiatives/op-tee/

first-in first-out policy, enables the scalability of the solution in a multi-guest architecture.

C. RPMsg Adaptation

Russell's VirtIO [10] and Texas RPMsg implementations provide the foundation for TZ-VirtIO implementation on top of the GPOS, while OpenAMP RPMsg and VirtIO implementation provides the foundation for the RTOS approach.

Amongst the modifications in the communication adaptation is the Remoteproc module elimination, intrinsically connected with RPMsg and VirtIO. Remoteproc contains the processor-dependent software and is responsible for the processor cores life cycle management and application binary load on its native implementation. However, these last features violate the virtualization isolation imposed by the hypervisor and are already part of the LTZVisor framework. The VirtIO configuration and memory allocation are now performed statically by the RPMsg module. Furthermore, the pair GPOS-slave/RTOS-master modes were implemented using the same RPMsg/VirtIO standards. On both configurations the RPMsg blocking fetch of VirtIO buffers was removed.

D. Key Features

The key features of TZ-VirtIO can be summarized as:

- Memory fault proof – all of TZ-VirtIO resources are allocated statically, including the shared memory region and VirtIO's shared memory control data, hindering any access to unmapped or inexistent memory.
- Real-time – LTZVisor's real-time requirements are unaffected by TZ-VirtIO's communication due to its asynchronous multipath communication properties. The communication tasks are assigned with the lowest priority and VirtIO implements lock-free shared memory unidirectional FIFOs and non-blocking fetch of buffers. Furthermore, TZ-VirtIO contemplates, on both single- and multi-core configuration, an interrupt request disabler at hypervisor level, to be used by any of the receiving guests, foreshadowing an RTOS possible need for critical sections or a malicious use of the interrupt.
- Throughput – TZ-VirtIO efficiency relies on its VirtIO transport layer. The use of shared memory reduces the overhead induced by avoidable data copies. Moreover, the RPMsg burst message capability combined with the interrupt storing mechanism provide an efficient separated event path without unnecessary context-switch overhead.

E. Limitations

Although announced as RPMsg/VirtIO communication possible capabilities, some features such as zero-copy and message sampling, as well as the implementation of remote procedure call (RPC) pattern on top of the RPMsg were not implemented in TZ-VirtIO. The single interrupt triggering per context-switch can be seen as a limitation on specific cases, given that the "buffer free" interrupt has lower priority than the message notification interrupt inside the hypervisor storage buffer, and could represent a bottleneck on buffer fetch event-driven applications.

TABLE I: Memory Footprint (bytes)

	Sections			Total
	.text	.data	.bss	
Single-core	52868	284	452656	505808
Single-core comm	80548	752	453316	534616
Multi-core	46488	284	423984	470756
Multi-core comm	80312	752	424644	505708

TABLE II: Context-switch average duration

Context-Switch	Buffer	μ (clock cycles)	σ (clock cycles)
Secure to Non-Secure	- (1.1)	1675	39
	Empty (1.2)	2150	53
	Filled (1.3)	3885	49
Non-Secure to Secure	- (2.1)	3411	46
	Empty (2.2)	3990	49
	Filled (2.3)	5610	55

IV. Evaluation

The performance of the communication mechanism and its impact on LTZVisor was evaluated in the Zedboard, a TrustZone-enabled platform endowed with a dual ARM Cortex-A9 running at 667MHz. The hardware architecture enables the characterization of both the single-core and multi-core capabilities of the TZ-VirtIO communication. Performance results were gathered resorting to the Performance Monitoring Unit (PMU) component present in the SoC. Memory footprint results were collected using the size tool of ARM GNU Xilinx toolchain. Linux (version 4.0) and FreeRTOS (7.0.2) run as non-secure and secure guest OS, respectively.

A. Overhead

TZ-VirtIO introduces two sources of overhead in the LTZVisor system: memory overhead and context-switch time overhead. The context-switch overhead is only present in the single-core architecture and has two distinct parts: the handling of the interrupt storage circular buffer by the hypervisor and, in case of interrupt existence, its consequent triggering. Hence, the context-switch overhead was monitored in three different scenarios: with interrupts disabled (1.1 and 2.1 in Table II) and with interrupts enabled; the latter either with the interrupt buffer empty (1.2 and 2.2) or filled (1.3 and 2.3).

Table I displays the memory footprint in bytes for each software component for both single-core and multi-core configurations. The memory footprint refers exclusively to the system's TCB, i.e., the software running on the secure world side: LTZVisor and RTOS. In accordance with Table I, TZ-VirtIO represents an addition of 5,7% and 7,2% in the memory footprint for the single-core and multi-core configurations, respectively.

As for the context-switch performance, each test was repeated a hundred times, and the results report the mean (μ) value and the respective standard deviation (σ) of the collected

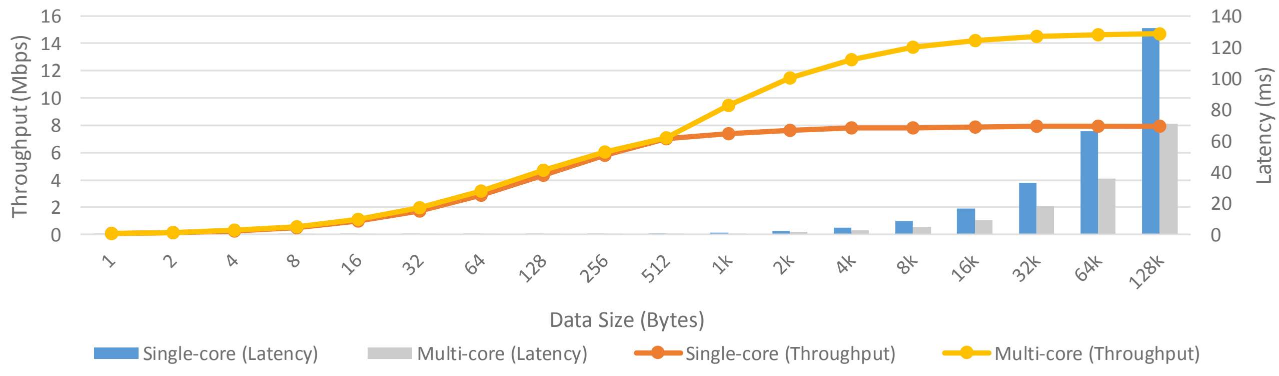

Fig. 3: TZ-VirtIO latency and throughput for different data size (single-core and multi-core)

measurements. The fluctuation in the measured values, in spite of the constant number of instructions executed, is justified by some dynamic architectural features of the Cortex-A9 processor, such as cache and branch prediction.

Table II presents the overhead introduced in the context-switch performance. An addition of 28,36% and 16,97% to the ”secure to non-secure” and ”non-secure to secure” context-switch time, respectively, is caused solely by the storage interrupt buffer handling (1.2 and 2.2). In case it requires an interrupt triggering, scenarios 1.3 and 2.3, the impact on the context-switch time is much higher, increasing the context-switch time by 132% and 64,4%, respectively.

The values presented in Table II indicate that, as expected, the inter-partition interrupt routing through the hypervisor has a slightly negative impact on the global system performance and latency. Although a mandatory characteristic in the single-core architecture, this event routing also represents a trade-off between performance and security. It guarantees the reliability of the communication, providing a secure interface for the inter-partition notification system.

B. *Performance*

To assess the performance of the implemented communication mechanism the experiments were conducted for a best case scenario. The low priority assigned to the communication tasks (i.e., for keeping LTZVisor real-time requirements near intact) results in an amount of non-deterministic restraints which make very difficult to experiment a worst-case evaluation (which ultimately could lead to the non-existence of communication). Therefore, to setup the best case scenario no real-time tasks were added to the system, which mean the communication tasks had the highest priority of execution.

For this test case scenario the RTOS was configured as master (communication is issued from the FreeRTOS to the Linux) and its buffer payload maximum size was set to 512 bytes. Each value represents the throughput and latency equivalent to the time measured between the issue of the first message until the arrival of the last message for the different payload sizes. Each sample reflects the average of 100 collected measurements.

Fig. 3 depicts the throughput and latency for different data sizes. For a single message, i.e. size below 512 bytes, the throughput is very similar for both single-core and multi-core configurations. For a data size of 1 byte, the throughput of the single-core configuration is 17% less than for the multi-core configuration. The reason behind this penalty is related to the extra context-switch time needed to handle asynchronous notifications, which do not happen in a multi-core configuration. This performance degradation decreases as the data size increases, reaching 1% for a message with 512 bytes. When the data size is higher than 512 bytes, i.e. more than one message is transmitted, the throughput for the single-core communication keeps near 8Mbps. For the the inter-partition communication in the multi-core configuration its throughput reaches 14,5Mbps, which means an improvement of 85% comparing with the single-core architecture. This values demonstrate the performance enhancement provided by the parallel processing of the multi-core configuration.

V. Related Work

Several virtualization solutions implement inter-partition communication mechanisms, either with the intent of providing guest-level device para-virtualization or merely to provide a communication infrastructure among partitions. However, the majority of existing solutions follows an ad hoc implementation with limited applicability outside their proposed solution’s scope.

Xen [5] implements its own communication mechanism with a transport system which was the inspiration for VirtIO. Despite its widespread popularity, the use of this communication system outside of Xen scope is severely difficult as it would require the support for Xen-Bus probing and configuration system. The inter-domain communication mechanism is mainly used for device para-virtualization. Contrarily to typical approaches, Xen places the back-end drivers on the most privileged guest (Dom0) which has full ownership over the systems devices. The least privileged guests (Dom-U) communicate with Dom0 through the front-end drivers.

SASP [6], a dual-OS TrustZone-based solution, implements a secure device access similar to Xen’s: only one partition has full access to the system devices, requiring device para-virtualization at guest level. The underlying communication mechanism between guests follows a similar approach to TZ-VirtIO on the event path level, featuring the use of IPIs. However, SASP implements a synchronous communication with shared memory at monitor level, which reveals a blocking mechanism with unnecessary data copies, favoring reliability at the expense of performance.

SafeG [7] implements a TrustZone-based dual-OS communication system. It features several unique mechanisms: shared memory at user-space level, range-checking control data, message filters, and interrupt limiters. The shared memory is managed by untrusted-privileged applications exposing it to potential malicious software attacks by untrusted-unprivileged applications. The communication presents some similarities with TZ-VirtIO such as availability of single event notification for several messages and transmission messages divided into two different paths (data and event); however, it follows a non-standardized interface.

Jailhouse [8] is an open-source Linux-based hypervisor. Shared device access and inter-domain communication is enabled by the hypervisor through shared mapped-IO and shared memory, respectively. Both kinds of communication are unsupervised, compliant with its minimalistic design, an unreliable approach which depends on the guests appropriate behavior. The inter-partition communications implementation is based on virtual peripheral component interconnect (PCI) using Message Single Interrupts (MSI-X) or legacy interrupts; optionally, a virtual ethernet link can be implemented on top of it. In systems lacking PCI hardware, Jailhouse emulates a simple generic PCI host controller.

KVM [11] and lguest [12], both Linux-hosted hypervisors, make use of the standardized VirtIO drivers to implement device para-virtualization. lguest was implemented as proof of concept and with the aim to standardize open-source para-virtualization techniques. KVM, a mature approach on Linux-centric hypervisor, followed lguest in the use of VirtIO. The communication mechanism in these hypervisors is, however, limited to guest-host communication.

VI. Conclusion

This paper presented a standardized inter-partition communication for a TrustZone-assisted hypervisor. TZ-VirtIO provides an asynchronous non-persistent communication mechanism independent of the core architecture of the hypervisor, supporting both single and multi-core configurations. The proposed approach is based on the VirtIO standard, and makes use of its infrastructure for the communication transport abstraction layer. Experiments demonstrated a negative impact on the hypervisor context-switch execution time, on the single-core configuration, mainly caused by the event path routing. However, without the event path the communication would become unsupervised and thus, unreliable. Assessed results demonstrate the maximum throughput increases from 8 Mbps to 14.5 Mbps when the system scales from a single- to a multi-core configuration.

Work in the near future will focus on the evaluation on a broader spectrum of scenarios and on the implementation of additional communication features such as zero-copy and message sampling. In the near feature, we also plan to merge this communication subsystem to the open-source project's repository.

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.

References

[1] 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.

[2] G. Heiser, "Virtualizing embedded systems - why bother?" in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2011, pp. 901–905.

[3] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the Key," in *29th Euromicro Conference on Real-Time Systems*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 4:1–4:22.

[4] G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors," in *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, ser. APSys '10. ACM, 2010, pp. 19–24.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

[6] S. W. Kim, C. Lee, M. Jeon, H. Y. Kwon, H. W. Lee, and C. Yoo, "Secure Device Access for Automotive Software," *ICCVE International Conference on Connected Vehicles and Expo*, pp. 177–181, 2013.

[7] D. Sangorrín, S. Honda, and H. Takada, "Reliable and efficient dual-OS communications for real-time embedded virtualization," *Information and Media Technologies*, vol. 8, no. 1, pp. 1–17, 2013.

[8] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look Mum, no VM Exits! (Almost)," *CoRR*, vol. abs/1705.06932, 2017.

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

[10] 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.

[11] 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.

[12] R. Russel, "lguest: Implementing the little Linux hypervisor," *IBM OzLabs*, vol. 7, pp. 173–178, 2007.

[13] S. Patni, J. George, P. Lahoti, and J. Abraham, "A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial," in *2015 1st International Conference on Next Generation Computing Technologies*, Sept 2015, pp. 6–9.

[14] F. Diakhaté, M. Perache, R. Namyst, and H. Jourdren, "Efficient shared memory message passing for inter-vm communications," in *Euro-Par 2008 Workshops - Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 53–62.

[15] F. Baum and A. Raghuraman, "Making Full use of Emerging ARM-based Heterogeneous Multicore SoCs," XCell95, Xillinx, October 2015.

[16] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares, "Lightweight multicore virtualization architecture exploiting ARM TrustZone," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, Oct 2017, pp. 3562–3567.

[17] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares, "IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices," *IEEE Internet Computing*, vol. 21, no. 1, pp. 40–47, Jan 2017.