Full Virtualization on Low-End Hardware: a Case Study

Adriano Carvalho*, Vitor Silva*, Francisco Afonso†, Paulo Cardoso*, Jorge Cabral*,
Mongkol Ekpanyapong‡, Sergio Montenegro§ and Adriano Tavares*

*Embedded Systems Research Group
Universidade do Minho, 4800–058 Guimarães — Portugal
†Instituto Politécnico de Coimbra (ESTGOH), 3400–124 Oliveira do Hospital — Portugal
‡Asian Institute of Technology, Pathumthani 12120 — Thailand
§Universität Würzburg, 97074 Würzburg — Germany

***Abstract*—Most hypervisors today rely either on (1) full virtualization on high-end hardware (i.e., hardware with virtualization extensions), (2) paravirtualization, or (3) both. These, however, do not fulfill embedded systems' requirements, or require legacy software to be modified. Full virtualization on low-end hardware (i.e., hardware without virtualization extensions), on the other end, has none of those disadvantages. However, it is often claimed that it is not feasible due to an unacceptably high virtualization overhead. We were, nevertheless, unable to find real-world quantitative results supporting those claims.**

In this paper, performance and footprint measurements from a case study on low-end hardware full virtualization for embedded applications are presented. More specifically, this paper presents: (1) an evaluation of the virtualization overhead in a Linux-based system as a guest on POK/rodosvisor, a real time operating system for embedded systems featuring low-end hardware full virtualization; (2) a detailed look at the performance of POK/rodosvisor's internal operation, namely, interrupt handlers and context switching; and (3) POK/rodosvisor's footprint for various configurations.

To the best of our knowledge, this is the first paper ever to present performance measurements about a Linux-based system as a guest on a hypervisor based on low-end hardware full virtualization, and therefore, targeting a common real-world scenario.

I. Introduction

Full virtualization on high-end hardware and paravirtualization are, by far, the most common types of virtualization found in the state of the art. With full virtualization in general, the hypervisor provides virtual machines with an interface identical to that of the underlying hardware platform (i.e., the real/physical machine); such a virtual machine is, thus, capable of hosting unmodified legacy software. In particular, full virtualization on high-end hardware (i.e., hardware with dedicated support for virtualization), however, is not suitable for embedded systems, since it currently leads to large system size, high weight, high power consumption, high cost, etc. On the other end, with paravirtualization, the virtual machines established by the hypervisor do not provide an interface identical to the underlying hardware platform and, instead, provide a different, more efficient interface. Paravirtualization is not dependent on particular hardware; nevertheless, the guests of (para)virtual machines need to be modified to fit a hypervisor-specific, often proprietary, interface. Full virtualization on low-end hardware, on the other end, has none of those disadvantages.

Full virtualization on low-end hardware (i.e., hardware without dedicated support for virtualization) must be accomplished using mechanisms which were not originally designed for it. Therefore, full virtualization on low-end hardware is not always possible because the hardware may not fulfill the necessary requirements (i.e., "the set of sensitive instructions for that computer is a subset of the set of privileged instructions" [1]). Furthermore, it is often claimed that it is not feasible due to an unacceptably high virtualization overhead (e.g., [2]); however, we were unable to find real-world quantitative results to support those claims.

In this paper, performance and footprint measurements from a case study on low-end hardware full virtualization for embedded applications are presented. More specifically, a case study to evaluate to what extent low-end hardware full virtualization is an alternative to high-end hardware full virtualization and paravirtualization, and provides compatibility with unmodified legacy software with acceptable performance and footprint. In this paper, first, an evaluation of the virtualization overhead in a Linux-based system is presented, by comparing several benchmarks performed on a bare metal Linux-based system and on a Linux-based system as a guest on POK/rodosvisor. POK/rodosvisor is a real time operating system for embedded systems featuring low-end hardware full virtualization of most of the feature set of the IBM PowerPC 405 [3], also known as partial virtualization. This is similar to what is presented in [4]; there, however, a hypervisor based on paravirtualization is used. Second, a detailed look at the performance of POK/rodosvisor's internal operation is presented (i.e., POK/rodosvisor's performance profile), namely: interrupt handlers and context switching. Third and last, POK/rodosvisor's footprint for various configurations is presented.

This paper is organized as follows. In the following section, section II, background is given on virtualization, POK/rodosvisor, and on the IBM PowerPC 405. In section III, the evaluation platform is described. In section IV, perfor-

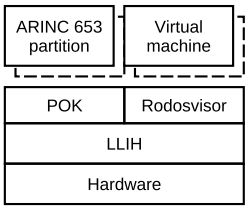

Fig. 1. The architecture of POK/rodosvisor.

mance and footprint measurements are presented, namely: the virtualization overhead in a Linux-based system, the performance profile of POK/rodosvisor, and POK/rodosvisor's footprint for various configurations. Finally, in section V, a summary of this paper is given and future work is proposed.

II. Background

A. Virtualization

Through time and space partitioning a hypervisor enforces workload isolation and enables workload consolidation. Workload isolation guarantees that the behavior of a virtual machine, such as a failure, will not affect the rest of the system (i.e., other virtual machines and the hypervisor). Furthermore, with workload isolation, the guests can be developed independently of each other, leading to lower development and certification effort. On top of workload isolation, workload consolidation enables a single computing unit to perform the same functions as multiple, physically independent computing units, leading to systems with a smaller size, lower weight, lower power consumption, lower cost, etc. [5].

Most hypervisors today feature high-end hardware full virtualization (e.g., [6], [7]), paravirtualization (e.g., [8], [9], [10], [2]), or both (e.g., [11], [12]). To the best of our knowledge, only Proteus [13], and POK/rodosvisor, presented in this paper, feature low-end hardware full virtualization. The authors of Proteus, however, failed to demonstrate compatibility with legacy software through a real-world scenario, such as a Linux-based system as a guest on top of their hypervisor; they only present simple synthetic benchmarks which are not good representatives of real applications.

B. POK/rodosvisor

POK/rodosvisor is a real time operating system for embedded systems featuring ARINC 653 partitions [14], and low-end hardware full virtualization of most of the feature set of the IBM PowerPC 405 [3], also known as partial virtualization. Even though the entire feature set is not virtualized, one can workaround the missing features using only the features that are already virtualized (instead of requiring the use a completely new set of features or interface, as it is the case with paravirtualization); provided, nevertheless, that access to the source code is possible. POK/rodosvisor is the result of the integration of POK [15], a real time operating system featuring ARINC 653 partitions, and Rodosvisor [16], [17], [18], a previously bare metal hypervisor, featuring low-end hardware full virtualization for the PowerPC 405.

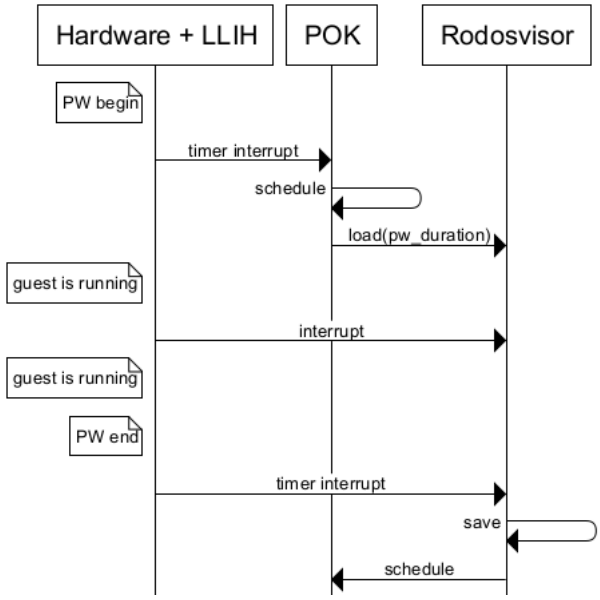

Fig. 2. Scheduling, loading and saving the context of a virtual machine in POK/rodosvisor.

The architecture of POK/rodosvisor is shown in Fig. 1. All partitions (ARINC 653 partitions and virtual machines) are scheduled according to a static scheduling policy, as defined by ARINC 653 [14]. Whenever there is an interrupt, it is first handled by the low-level interrupt handlers (i.e., "LLIH" in Fig. 1). When a virtual machine is running, interrupts are directed to Rodosvisor, which is itself composed by a set of interrupt handlers which take care of the virtualization of the PowerPC 405 as described in [16], [17], [18]. If, on the other end, an ARINC 653 partition is running, interrupts are instead directed to POK.

As illustrated in Fig. 2, when a virtual machine's partition window begins (i.e., "PW begin" in Fig. 2), POK loads the context of the virtual machine by calling Rodosvisor, while specifying the duration of the partition window (i.e., "load(pw_duration)"). After loading a virtual machine's context, Rodosvisor takes control over the hardware, and all interrupts are directed to Rodosvisor. When the virtual machine's partition window is over (i.e., "PW end" in Fig. 2), Rodosvisor saves/unloads the virtual machine's context, returns the hardware to the state expected by POK, and calls POK's scheduler, returning control back to POK.

C. IBM PowerPC 405

The IBM PowerPC 405 [3] is a 32-bits Reduced Instruction Set Computer (RISC) processor with a five-stage pipeline. It is a low power processor (0.9 mW/MHz) designed for high performance embedded systems. It features: hardware multiply/divide unit; 16 KB two-way set-associative instruction cache and 16 KB two-way set-associative data cache; a memory management unit with a software-managed 64-entry translation look-aside buffer; several timer units; debug facilities; etc.

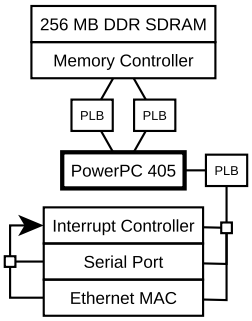

Fig. 3. The architecture of the hardware platform.

The PowerPC 405 has been chosen because (1) it is a simple, low power, low cost processor core, especially dedicated to embedded systems, and thus, a good representative of low-end hardware, and because (2) Xilinx provides good development support, enabling the construction and evaluation of various hardware configurations, such as single-core and dual-core processor configurations.

III. Evaluation Platform

The results shown in the following sections have been collected on a hardware platform with the architecture depicted in Fig. 3. It consists of: (1) a PowerPC 405 running at 300 MHz; (2) a memory controller (i.e., Xilinx Multi-Port Memory Controller, version 4.03a) connected to 256 MB of DDR SDRAM; (3) a serial port (i.e., Xilinx XPS UART Lite, version 1.00a) with one start bit, one stop bit, and a baudrate of 115200; (4) an ethernet media access controller (MAC) (i.e., Xilinx XPS Ethernet Lite Media Access Controller, version 2.00b), which supports the IEEE 802.3 media independent interface to industry standard physical layer devices, and provides 10/100 Mbps interfaces; (5) an interrupt controller (i.e., Xilinx XPS Interrupt Controller, version 1.00a) connected to the serial port's and the ethernet MAC's interrupt request signals; and lastly, (6) three processor local buses (PLB), version 4.6 (two of them connect the PowerPC 405 exclusively to the memory controller for improved performance, one for fetching instructions and another for data load/store operations, while the third connects the PowerPC 405 to memory-mapped I/O devices, namely, the ethernet MAC, the serial port, and the interrupt controller).

The hardware platform just described has been realized on a Xilinx University Program Virtex-II Pro Development System (XUPV2P) [19], which is equipped with a Virtex-II Pro (XC2VP30) field-programmable gate array (FPGA) [20], and a comprehensive collection of input and output devices, which can be used to create many different hardware configurations. Xilinx Embedded Development Kit, version 10.1, has been used to create, configure and generate the evaluation hardware platform, on a Fedora 19 host. All of the software has been compiled on a Fedora 19 host using a GCC cross-compiler for the PowerPC 405, version 4.8.1, with optimizations enabled (i.e., "-O2" compiler option).

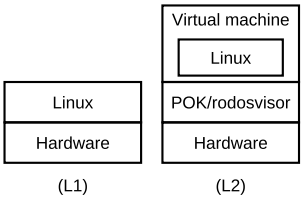

Fig. 4. Configurations used to evaluate the virtualization overhead in a Linux-based system: (L1) Linux-based system on bare metal; (L2) Linux-based system as a guest on POK/rodosvisor.

IV. Evaluation

A. Virtualization Overhead

In this section, the virtualization overhead in a Linux-based system, as a guest on top of POK/rodosvisor, is presented, for three different types of workloads, namely: compute-, I/O- and CPU-management-intensive workloads. For that two configurations have been developed and tested:

L1. A system based on Linux 2.6.39, with a small RAM-based file system (i.e., initramfs), running on bare metal, as illustrated in Fig. 4(L1).

L2. A system based on POK/rodosvisor, with one virtual machine whose guest is the same Linux-based system described in L1, as illustrated in Fig. 4(L2). The virtual machine is configured: (1) to have direct access to the serial port and to the ethernet MAC; and (2) with a virtual interrupt controller which emulates a physical interrupt controller. Direct access to a physical interrupt controller is not given because it can be configured to bypass the hypervisor (i.e., POK/rodosvisor) and violate time and space partitioning.

Currently, independent control over instruction and data address translation, a feature of the PowerPC 405, is not supported by POK/rodosvisor. Linux, however, uses this feature when flushing the cache during initialization. Therefore, the Linux kernel's source code has been modified to workaround this limitation, requiring 34 new/modified source lines of code. Still, only a subset of the features provided by the PowerPC 405 have been used to workaround this limitation, instead of requiring the use a completely new set of features or interface, as it is the case with paravirtualization. Other than that, no modifications were necessary.

On top of the two configurations described above, several benchmarks have been executed. Table I presents the results of those benchmarks, the associated virtualization overhead, and their respective type. In Table I, the following benchmarks can be found:

1) Boot, Decompression: The boot process of the Linux-based system has been divided into 3 stages, and this result corresponds to the first stage. It corresponds to the time required to decompress the Linux kernel image (including the initramfs). In this stage, the use of CPU management operations (e.g., privileged instructions, interrupts) is negligible, and thus, this benchmark has been classified as compute-intensive (i.e., "Compute" in Table I). Results show that the

TABLE I
THE RESULTS OF SEVERAL BENCHMARKS, PERFORMED ON TOP OF A LINUX-BASED SYSTEM, RUNNING ON BARE METAL (I.E., L1), OR AS A GUEST ON POK/RODOSVISOR (I.E., L2). FOR EACH BENCHMARK, THE ASSOCIATED VIRTUALIZATION OVERHEAD, IN PERCENTAGE, AND ITS TYPE ARE ALSO SHOWN.

	L1	L2	Overhead	Type
Boot (seconds)				
Decompression	36	36	0.00	Compute
Linux	2.9	4.2	44.83	CPU mgmt.
Shell	1	4	300.00	CPU mgmt.
Total	39.9	43.2	8.27	
Dhrystone (Dhrystones per Second)				
	577800.9	566989.9	1.91	Compute
Whetstone (C Converted Double Precision Whetstones (MIPS))				
	3.8	3.7	2.70	Compute
Netperf (Mbit/s)				
TCP_STREAM	17.4	13.9	25.18	I/O

virtualization overhead is non-existent. This is expected since there is very little use of CPU management operations, and therefore, the hypervisor is rarely activated.

2) Boot, Linux: This is the second stage of the boot process of the Linux-based system, and it corresponds to the initialization of the Linux kernel. In this stage the use of CPU management operations is high, therefore, this benchmark has been classified as CPU-management-intensive (i.e., "CPU mgmt." in Table I). Results show that the virtualization overhead is high (i.e., 44.83%), which is expected, since the use of CPU management operations is high and, therefore, the hypervisor is activated often. This overhead, however, includes not only the hypervisor's overhead, but also the overhead caused by cache trashing during the interaction between the hypervisor and the Linux kernel.

3) Boot, Shell: This is the third and last stage of the boot process, which corresponds to the initialization of a Buildroot-based shell [21]. Similarly to the previous stage, the use of CPU management operations is significant, thus, this benchmark has been classified as CPU-management-intensive. Compared to the previous stage, however, the use of CPU management operations is even higher as this stage begins with the execution of the "init" process, on top of Linux, and with the activation of process scheduling, context switching, system calls, etc. Therefore, as the results show, the virtualization overhead is also higher than in the previous stage.

4) Dhrystone and Whetstone: Dhrystone [22] is a benchmark which measures the performance of integer and string operations, and Whetstone [23] is a benchmark which measures, mainly, the performance of floating-point arithmetic. These two benchmarks have been classified as compute-intensive. Results show that the virtualization overhead is low, but noticeable. It was expected that, similarly to the first stage of the boot process, the virtualization overhead would be negligible, as the operations being benchmarked do not rely on CPU management operations, and thus, do not lead to the activation of the hypervisor. What these results show is the virtualization overhead during the execution of the benchmark, caused by Linux's normal operation (e.g., scheduling) as well as the overhead caused by cache trashing during the interaction among the hypervisor, the Linux kernel, the benchmark process, and other processes running concurrently on top of the Linux kernel.

5) Netperf: Lastly, Netperf [24] is a benchmark which measures the throughput and latency for various kinds of network connections (e.g., TCP, UDP). To perform this benchmark, *netserver* executed on the target while *netperf* executed on a Fedora 19 host. This benchmark has been classified as I/O-intensive (i.e., "I/O" in Table I). The results for the TCP_STREAM test profile, which measures throughput over a TCP connection, are shown. It can be seen that L1's maximum throughput is 25% higher than that of L2's. In this benchmark, the virtualization overhead includes not only the overhead caused by CPU management operations (in the Linux kernel, in the device drivers, in the protocol stack, etc.), but also the virtualization overhead of the virtual interrupt controller which is connected to the ethernet MAC. Considering the complexity of this benchmark, which includes two device drivers (i.e., one for the ethernet MAC and another for the interrupt controller) and a protocol stack, we expect the virtualization overhead to be much lower for most I/O-intensive workloads, which are not as complex.

To sum up, these results show that: (1) for compute-intensive workloads, the virtualization overhead is low and, in some cases, negligible; (2) for I/O-intensive workloads, the virtualization overhead is significant, but for most workloads, we expect it to be low; (3) for CPU-management-intensive workloads, however, the virtualization overhead can be quite significant. These results indicate that low-end hardware full virtualization is more adequate for compute-intensive workloads, with moderate use of I/O, and with low use of CPU management operations. Some of the reasons for this behavior will become more clear in the next section, which presents POK/rodosvisor's performance profile.

By showing the results of some benchmarks executed on top of a Linux-based system as a guest on POK/rodosvisor, it demonstrates, indirectly, compatibility with legacy software (i.e., a complete Linux-based system). To the best of our knowledge, this is the first paper ever to present performance measurements about a Linux-based system as a guest on top of a hypervisor based on low-end hardware full virtualization, and therefore, demonstrating compatibility with a common real-world scenario.

B. POK/rodosvisor's Performance Profile

To obtain POK/rodosvisor's performance profile, configuration L2, described in the previous section, was reused. POK/rodosvisor's built-in, custom profiler was enabled and profiling data were collected during 90 seconds since boot (it includes almost 45 seconds required to boot the Linux-based system, and another 45 seconds of idle time).

The profiling data were composed of several probes, each associated with a specific code path. For each probe the following data were collected: the number of samples (or activations), the minimum, the maximum and the total execution

TABLE II
THE NUMBER OF SAMPLES, THE AVERAGE AND THE TOTAL EXECUTION TIMES PER PROBE. EXECUTION TIMES ARE GIVEN IN CPU CLOCK CYCLES. IN PARENTHESIS, THE RATIO TO THE SUMMATION OF ALL THE VALUES IN THE SAME COLUMN IS GIVEN, IN PERCENTAGE.

Probe	No. samples	Average	Total
Program	1841211 (96.857%)	511	868824642 (91.639%)
Data TLB-Miss	29310 (1.542%)	1356	39366270 (4.152%)
Instruction TLB-Miss	22824 (1.201%)	1594	36074969 (3.805%)
System Call	4800 (0.253%)	419	1945588 (0.205%)
Data Storage	1875 (0.099%)	364	657625 (0.069%)
PIT	475 (0.025%)	1312	616728 (0.065%)
Context Switch In	90 (0.005%)	3247	292259 (0.031%)
Context Switch Out	89 (0.005%)	1815	161549 (0.017%)
Instruction Storage	253 (0.013%)	522	128658 (0.014%)
External	28 (0.001%)	988	27300 (0.003%)

time; the average execution time was obtained by dividing the total execution time by the number of samples.

In Table II, the number of samples, the total and the average execution times per probe are shown (only probes with a number of samples greater than zero are presented). This table includes all the interrupt handlers that have been activated (e.g., program, instruction and data TLB-miss), as well as the time to switch contexts in and out of the virtual machine.

It can be seen that the program interrupt handler, which is responsible for the emulation of privileged instructions, makes up for 91.7% of the total execution time. However, it is also, by far, the most activated interrupt handler (i.e., 96.9%), and it actually has a low average execution time. This partially explains why CPU-management-intensive benchmarks performed so poorly.

After the program interrupt handler, the instruction and data TLB-miss interrupt handlers make up for 8% of the total execution time, even though their contribution to the total number of samples is less than 3%. This happens because of the high (actually, the highest) average execution time for these two interrupt handlers.

The remaining interrupt handlers and context switching make up for less than 0.4%, which compared with the ones discussed above is negligible.

To sum up, these results indicate that low-end hardware full virtualization is a serious alternative for workloads which do not rely heavily on CPU management operations.

C. Footprint

In this section, the footprint of POK/rodosvisor for various configurations is presented. The footprint of POK/rodosvisor is also compared with the kernel's footprint of an equivalent POK-based, ARINC-653-compliant system. To do that, the following two sets of configurations have been developed and tested:

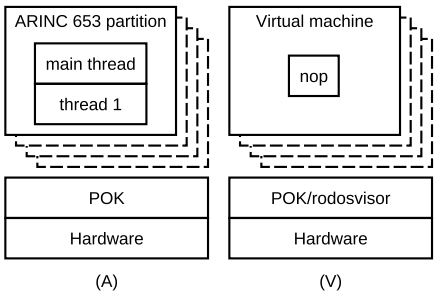

Fig. 5. Configurations used to measure and compare the kernel's footprint of (A) an ARINC-653-based system and (V) a virtual-machine-based systems.

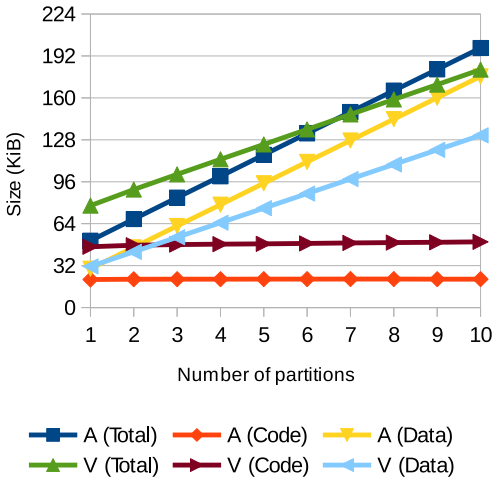

Fig. 6. The kernel's footprint, in terms of the total size, as well as the size of the code and data, for systems based on (A) ARINC 653 partitions, and (V) virtual machines.

A. Configurations with one up to 10 ARINC 653 partitions as illustrated in Fig. 5(A); each partition has two threads: a mandatory main thread, and one worker thread which performs no operation.

V. Configurations with one up to 10 virtual machines, as illustrated in Fig. 5(V); the guest is the same for all virtual machines and performs no operation.

All of the configurations just described have been compiled, and from the resulting executable and the system's behavior during run time (e.g., dynamic memory allocation), the size of the code and data has been measured. The size of the data includes: the size of read-only data, the size of read/write data, and the combined size of all stacks. The total size of the kernel, as well as the size of the code and data, for all configuration is shown in Fig. 6.

It can be seen that for configurations with a small number of partitions (i.e., ARINC 653 partitions or virtual machines), ARINC-653-based systems lead to a smaller kernel's footprint than virtual-machine-based systems. As the number of par-

titions increase, however, the difference decreases, and, for a large number of partitions (more than six), the kernel's footprint of a virtual-machine-based system is smaller than that of an ARINC-653-based system. This shows that, when a large number of partitions is required, a virtual-machine-based system enables a reduction of the kernel's footprint. These results also indicate that low-end hardware full virtualization does not necessarily lead to a large kernel's footprint.

V. Conclusion

In this paper, performance and footprint measurements from a case study on low-end hardware full virtualization for embedded systems have been presented. The case study evaluates to what extent low-end hardware full virtualization is an alternative to high-end hardware full virtualization and paravirtualization, and provides compatibility with legacy software with acceptable performance and footprint.

In this paper, compatibility with legacy software has been demonstrated by showing the results of some benchmarks executed on top of a Linux-based system as a guest on POK/rodosvisor. To the best of our knowledge, this is the first paper ever to present performance measurements about a Linux-based system as a guest on top of a hypervisor based on low-end hardware full virtualization, and therefore, demonstrating compatibility with a common real-world scenario.

The evaluation of the virtualization overhead, and POK/rodosvisor's performance profile showed that, low-end hardware full virtualization is more adequate for compute-intensive workloads, with moderate use of I/O, and with low use of CPU management operations.

In terms of footprint, it has been shown that the kernel's footprint for a virtual-machine-based system can be lower than the kernel's footprint for an ARINC-653-compliant system. This indicates that the requirements of low-end hardware full virtualization regarding the kernel's footprint are not necessarily high.

Altogether, we believe that these results demonstrate that, for many applications, low-end hardware full virtualization can be a serious alternative to high-end hardware full virtualization and paravirtualization, enabling: (1) a reduction of system size, weight, power consumption, cost, etc., when compared with high-end hardware full virtualization; (2) a reduction (in many cases, elimination) of the effort required to port legacy software to a hypervisor-specific interface, when compared with paravirtualization; as well as (3) in some cases, a reduction of the kernel's footprint.

As future work, we propose to explore methods to address or workaround the limitations and bottlenecks found in POK/rodosvisor (e.g., full virtualization of the complete feature of the PowerPC 405, and POK/rodosvisor's program interrupt handler bottleneck). Furthermore, we propose to perform similar case studies on other processor architectures, and thus, to evaluate its potential on other processor architectures. Finally, we hope that the lack of published results on hypervisor's performance and footprint is a gap which will be filled in the future.

Acknowledgment

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. The work of A. Carvalho and V. Silva was supported by FCT (grants SFRH/BD/81640/2011 and SFRH/BD/82732/2011, respectively).

References

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[2] S. Trujillo, A. Crespo, and A. Alonso, "MultiPARTES: Multicore Virtualization for Mixed-Criticality Systems," in *2013 Euromicro Conference on Digital System Design (DSD)*, Sep. 2013, pp. 260–265.

[3] IBM, *PowerPC 405-S Embedded Processor Core User's Manual*, 1st ed., 2010.

[4] S. Campagna and M. Violante, "On the Evaluation of the Performance Overhead of a Commercial Embedded Hypervisor," in *The First Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale (MEDIAN'12)*, 2012, pp. 59–63.

[5] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, 2008, pp. 11–16.

[6] "Main Page - KVM." [Online]. Available: http://www.linux-kvm.org/page/Main_Page

[7] "QNX Hypervisor." [Online]. Available: http://www.qnx.com/products/hypervisor/index.html

[8] "Codezero Embedded Hypervisor™ - B Labs | ARM Connected Community." [Online]. Available: http://community.arm.com/docs/DOC-7123

[9] A. Whitaker, R. Cox *et al.*, "Rethinking the design of virtual machine monitors," *Computer*, vol. 38, no. 5, pp. 57 – 62, May 2005.

[10] Philipp Eppelt, "Running RTEMS in POK Partitions." [Online]. Available: https://wwwpub.zih.tu-dresden.de/~s8940405/rtlws13_rtems_in_pok_partitions.pdf

[11] Green Hills Software, "INTEGRITY Multivisor," 2015. [Online]. Available: http://www.ghs.com/products/rtos/integrity_virtualization.html

[12] "The Xen Project, the powerful open source industry standard for virtualization." [Online]. Available: http://www.xenproject.org/

[13] K. Gilles, S. Groesbrink *et al.*, "Proteus Hypervisor: Full Virtualization and Paravirtualization for Multi-core Embedded Systems," in *Embedded Systems: Design, Analysis and Verification*, ser. IFIP Advances in Information and Communication Technology, G. Schirner, M. Götz *et al.*, Eds. Springer Berlin Heidelberg, Jan. 2013, no. 403, pp. 293–305.

[14] "653p3a AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE,PART 3a, CONFORMITY TEST SPECIFICATION FOR ARINC 653 REQUIRED SERVICES." [Online]. Available: http://store.aviation-ia.com/cf/store/catalog_detail.cfm?item_id=2189

[15] "POK homepage: home." [Online]. Available: http://pok.safety-critical.net/

[16] A. Tavares, A. Carvalho *et al.*, "A customizable and ARINC 653 quasi-compliant hypervisor," in *2012 IEEE International Conference on Industrial Technology (ICIT)*, 2012, pp. 140–147.

[17] A. Carvalho, F. Afonso *et al.*, "Cache full-virtualization for the PowerPC 405-S," Bochum, Germany, Jul. 2013.

[18] A. Tavares, A. Didimo *et al.*, "RodosVisor - an Object-Oriented and Customizable Hypervisor: The CPU Virtualization," S. Klaus, Ed., University of Würzburg, Germany, Apr. 2012, pp. 200–205.

[19] Xilinx, *Xilinx University Program Virtex-II Pro Development System: Hardware Reference Manual*, 1st ed. Xilinx, 2009.

[20] ——, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 4th ed. Xilinx, 2007.

[21] "Buildroot - Making Embedded Linux Easy." [Online]. Available: https://buildroot.org/

[22] "http://www.netlib.org/benchmark/dhry-c." [Online]. Available: http://www.netlib.org/benchmark/dhry-c

[23] "http://www.netlib.org/benchmark/whetstone.c." [Online]. Available: http://www.netlib.org/benchmark/whetstone.c

[24] "The Netperf Homepage." [Online]. Available: http://www.netperf.org/netperf/