Exploring Embedded Systems Virtualization Using MIPS Virtualization Module

Carlos Moratelli
Faculty of Informatics
PUCRS - Porto Alegre, Brazil
carlos.moratelli@pucrs.br

Sergio Johann
Faculty of Informatics
PUCRS - Porto Alegre, Brazil
sergio.filho@pucrs.br

Fabiano Hessel
Faculty of Informatics
PUCRS - Porto Alegre, Brazil
fabiano.hessel@pucrs.br

ABSTRACT

Embedded virtualization has emerged as a valuable way to increase security, reduce costs, improve software quality and decrease design time. The late adoption of hardware-assisted virtualization in embedded processors induced the development of hypervisors primarily based on para-virtualization. Recently, embedded processor designers developed virtualization extensions for their processor architectures similar to those adopted in cloud computing years ago. Now, the hypervisors are migrating to a mixed approach, where basic operating system functionalities take advantage of full-virtualization and advanced functionalities such as inter-domain communication remain para-virtualized. In this paper, we discuss the key features for embedded virtualization. We show how our embedded hypervisor was designed to support these features, taking advantage of the hardware-assisted virtualization available to the MIPS family of processors. Different aspects of our hypervisor are evaluated and compared to other similar approaches. A hardware platform was used to run benchmarks on virtualized instances of both Linux and a RTOS for performance analysis. Finally, the results obtained show that our hypervisor can be applied as a sound solution for the IoT.

CCS Concepts

•**Computer systems organization → Embedded software; Real-time operating systems;** *Real-time system architecture;*

Keywords

Embedded Virtualization, Hardware-assisted Virtualization, Real-time

1. INTRODUCTION

In recent years, virtualization technology has quickly moved towards embedded systems (ESs) motivated by the increasing processing power of the embedded processors. Similar to what happened to general purpose computers, the late adoption of hardware-assisted virtualization for embedded processors made the para–virtualization rule the development of ES hypervisors. However, the main ES manufacturers already designed virtualization extensions for their processor families, e.g., PowerPC, ARM and MIPS. This enabled more design choices for hypervisor developers. Hardware–assisted virtualization makes it feasible to virtualize an operating system (OS) without any modification – an important feature to support legacy software. However, advanced hypervisor features like inter-VM (virtual machine) communication or shared devices may require para–virtualization. The resulting hypervisor has a hybrid architecture that combines full-virtualization and para-virtualization. It is important to highlight that the architectural diversity of ESs' goals and requirements have driven the appearance of several ES hypervisors, e.g., OKL4 [6], MultiPARTS [14] and AUTOSTAR [12], each one with its specific purposes.

The Hellfire Hypervisor was designed to support the recently released MIPS virtualization module (MIPS-VZ) and to keep small footprint for IoT applications. In this paper, we analyze the MIPS-VZ module showing how it can be used to achieve the essential features for embedded virtualization. Our main contribution is to demonstrate how to provide temporal isolation while reducing the interrupt delivery time for guest OSs taking advantage of hardware-assisted virtualization. Additionally, we conducted experiments to quantify four different aspects of our hypervisor: memory footprint, virtualization overhead, inter-VM communication performance and real-time capabilities.

The paper is organized as follows: Section 2 presents the related work, showing advantages and disadvantages of other similar approaches. Section 3 explains the implementation strategy using the MIPS VZ module of the M5150 processor. Section 4 depicts the results of our experiments. Finally, Section 5 presents the conclusion.

2. RELATED WORKS

Type-1 hypervisors are a software layer that interacts directly with the hardware, creating an abstraction layer between the virtualized OSs and the hardware platform. Type-2 hypervisors perform on top of a native OS. Both types are currently used for embedded virtualization. However, Type-1 hypervisors are preferable, especially for the IoT field, since they do not require an underlying OS resulting in a smaller footprint. Three different embedded virtualization approaches are discussed below. The memory footprint and overall performance are compared to our hypervisor in Section 4.

KVM (Kernel-based Virtual Machine) is a Type-2 hypervisor primarily designed for the x86 architecture with virtualization extensions (INTEL VT and AMD-V). It consists of a loadable Linux kernel module that provides a virtualiza-

CF'16, May 16-19, 2016, Como, Italy
© 2016 ACM. ISBN 978-1-4503-4128-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2903150.2903179

tion infrastructure [8]. KVM executes multiple unmodified (full-virtualized) Linux or Windows VMs with virtualized hardware: network card, disk, graphics adapters among others. Dall and Nieh [4] proposed a KVM port to ARM architecture with virtualization extensions, called KVM/ARM. This architecture takes the advantage of the wide Linux hardware support to the ARM family to simplify the hypervisor development and maintenance. It became the standard ARM hypervisor for Linux platforms. Experimental results showed an average of 10% overhead when compared to native execution and significantly lower overhead when compared to KVM x86 virtualization. KVM was designed for general-purpose computing. Therefore, it does not comply with ES constraints like small footprint and real-time. Additionally, KVM/ARM relies on the Linux OS scheduler which means it is difficult to improve real-time response on virtualized OSs.

Yunfang et al. [13] proposed the KVM-Loongson, a virtualization solution for MIPS based on KVM to the Loongson-3A [9] processor. Such processors do not implement the recently released MIPS virtualization extensions. Nonetheless, MIPS without the VZ module does not allow full–virtualization because it cannot support complete virtualization of kernel virtual address space. A possible solution targeting full-virtualization is to modify the processor core as presented at [2]. However, the authors modified the KVM (primarily designed for full-virtualization) to support para-virtualization. The authors performed an extensive study to determine all Linux kernel privileged instructions that could be substituted by hypercalls. As a result, they showed that about 98.6% of the privileged instructions can be substituted. Additionally, their memory virtualization overhead is about 4% on average. The main disadvantage of their technique is the effort to para-virtualize the Linux kernel and the impossibility to support proprietary software.

Xvisor [11] is an embedded hypervisor which supports both full and para-virtualization. Similar to other embedded hypervisors, it aims to provide a lightweight layer with reduced overhead and small footprint. Full-virtualization is supported by the use of ARM virtualization extensions avoiding the need for guest OS modifications. Still, it can map interrupts directly to guests, allowing guest interrupt handling without the intervention of the hypervisor. However, Xvisor only supports para-virtualization on the MIPS 24k processor model under the Qemu emulator.

To the best of our knowledge, the Hellfire Hypervisor is the first embedded hypervisor to implement support for the MIPS-VZ module with adequate performance and footprint targeting IoT class devices.

3. VIRTUALIZATION STRATEGY

The development of the presented hypervisor was guided by a set of key features required for embedded virtualization, as described:

- *Hardware-assisted virtualization*: Hardware-assistance reduces the hypervisor complexity and contributes to a lower memory footprint. Specifications of the virtualization extensions for the ARM and MIPS architectures have already been presented and several manufacturers have released their virtualization platforms;
- *Real-Time Support*: Real-time is an intrinsic characteristic of embedded systems. The hypervisor should be predictable and ensure temporal isolation between general-purpose operating systems (GPOSs) and real-time applications;
- *Coexistence of multiple GPOSs and Real-Time Instances*: GPOSs are required for certain ESs, such as smartphones and setup boxes, due to its wide diversity of software. However, GPOSs have poor real-time responsiveness. Thus, the hypervisor must guarantee temporal isolation between GPOSs and real-time operating systems (RTOSs);
- *Direct Mapped and Shared Devices*: When it is necessary to share a physical device, e.g. an Ethernet device, the hypervisor can use the para-virtualization technique avoiding excessive overhead caused by emulation techniques. However, when device sharing is not needed, the hypervisor must map the device directly to the desired guest OS for better performance;
- *Security*: The hypervisor must provide robust spatial isolation between VMs, i.e., a misbehavior in the guest OS should not affect the behavior of the other guest OSs or even the hypervisor;
- *Inter-VM communication*: A virtualized system is composed by a set of VMs. Possibly, these VMs will require some level of interaction among them. Thus, an efficient inter-VM communication mechanism must be available in the hypervisor.

The following subsections describe our implementation strategy for memory virtualization, fast interrupt delivery and real-time support. Additionally, our experience to support Linux in the hypervisor is described.

3.1 Virtualization Model

The Hellfire Hypervisor, first presented in [15], is a Type-1 hypervisor specially designed for embedded virtualization. Figure 1 shows the architecture for virtualization implemented by our hypervisor. The first software layer is the hypervisor, which is responsible for the creation and management of each VM. The current implementation supports only single-core processors, since the M5150 processor used for test and validation is a single-core device. The hypervisor controls the memory space of each VM for memory isolation, which is better explained in Subsection 3.2. To guarantee temporal isolation, GPOSs are mapped onto best-effort virtual CPUs (VCPUs) while real-time instances are mapped onto real-time VCPUs, which is further explained in Subsection 3.3. The hypervisor keeps spatial isolation between the guest kernel and guest applications while protecting itself. This protection is possible because processors implementing the MIPS-VZ module support a third level of execution (supervisor mode) with higher privilege designed for the hypervisor. Therefore, the hypervisor is protected from the actions of malicious or misbehaving guest OSs, and the guest OSs are protected from their applications. Embedded hypervisors designed for processors without proper virtualization assistance cannot distinguish the memory space between the guest kernel and applications, since the entire guest executes in the unprivileged processor's mode [16].

The hypervisor allows the implementation of the para-virtualization concept to provide extended services to the guest OSs. Extended services are useful to expand the virtualization functionalities, i.e., to implement functions that do not exist in a pure fully-virtualized system. For example, inter-VM communication and real-time support.

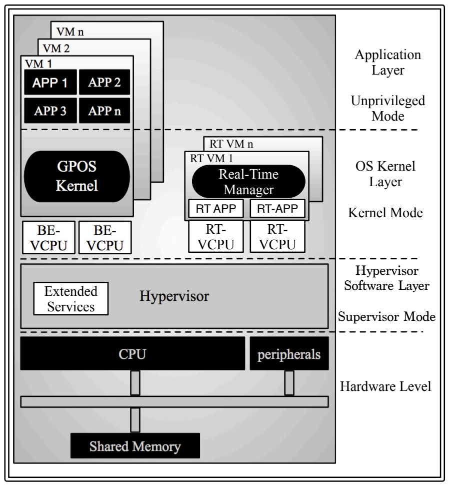


Figure 1: The overall view of our virtualization model.

3.2 Virtual Memory Management

Processors designed to support rich OSs must support a memory management unit (MMU) to provide virtual memory mechanisms. Thus, the OS can implement memory isolation between processes increasing software reliability and security. On MIPS, translation look-aside buffer (TLB) is a common hardware construction to support MMU. In a non-virtualized system, the OS translates virtual addresses (VAs) to physical addresses (PAs) using its page table to configure the TLB. In a virtualized system, VAs are translated to intermediate physical addresses (IPAs). Processors without the proper virtualization support require a technique that keeps the correct translation from IPA to PA in an intermediate page table managed by the hypervisor. This technique is called shadow page table.

Current virtualization extensions for embedded processor families, like MIPS and ARM, implement a second-stage TLB translation in hardware. Essentially, the hardware performs the translation from IPA to PA without software intervention. The hypervisor still manages its page table mapping IPA to PA in a second-stage TLB. The guest OS is allowed to directly configure the first-stage TLB. The resulting PA is generated by the hardware combining both TLBs. This mechanism decreases the number of hypervisor exceptions drastically and also hypervisor complexity. The M5150's TLB supports a second-stage TLB translation and a range of page sizes from 1Kbyte to 256Mbytes. Our hypervisor takes advantage of large pages to avoid TLB misses during IPA to PA translations. VMs are loaded into a contiguous memory region, and the IPA translation is statically mapped to reserved TLB entries. For example, in order to allocate 32Mbytes of physical address space to a VM, the hypervisor uses a dual-TLB entry (MIPS' TLB supports two pages by TLB entry) to map two consecutive 16Mbyte pages in the second-stage TLB. Figure 2 depicts this scheme. The guest OS manages the first-stage address translation in the exactly same way on a non-virtualized system. In the second-stage translation, the hypervisor maps a virtual memory address range to a contiguous physical address range.

A simplified virtual memory management is expected to bring advantages to ESs. First, it avoids the second-stage TLB misses by keeping the VM entirely mapped at the TLB during its execution. Thus, RTOSs that do not implement

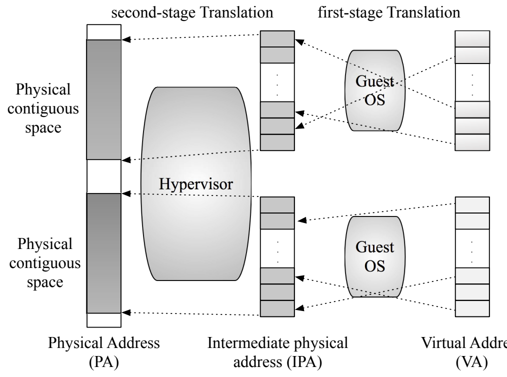


Figure 2: Virtual memory organization view of our hypervisor.

virtual memory support will not suffer additional delays due to hypervisor paging management. Secondly, some ESs have a limited number of virtual machines that can be executed and some of them keep a static configuration during their execution. For those systems, memory fragmentation due to contiguous guest OS allocation is not a major problem.

3.3 Temporal Isolation

The hypervisor specifies two different kinds of VCPUs: best-effort VCPUs (BE-VCPUs) and real-time VCPUs (RT-VCPUs). RT-VCPUs have priority over BE-VCPUs and follow the policy of the Early Deadline First (EDF) [7] scheduling algorithm. BE-VCPUs are scheduled by a best-effort scheduler algorithm that is invoked when there are not RT-VCPUs ready to execute.

To improve the temporal isolation between BE-VCPUs and RT-VCPUS, a special VM called RT-VM was designed. The RT-VM does not support RTOSs. Instead, it implements what is called a Real-Time Manager (RTM), which supports communication facilities and basic user libraries. The RTM can map its tasks directly to RT-VCPUs in the hypervisor scheduler, i.e., it does not implement a scheduler on its level avoiding the hierarchical scheduling problem [10] and improving performance. The hierarchical scheduling problem happens when the hypervisor schedules VCPUs and a guest executing in a VCPU schedules its own processes or tasks. Thus, the guest's task will be negatively influenced by the hypervisor scheduler. In addition, the RT-VM provides spatial isolation between BE and RT-VCPUs eliminating the priority inversion problem, i.e., when a BE-VCPU holds a lock required by a RT-VCPU.

The RTM API implements the hypercall concept to provide control to other VMs over the real-time services in the RT-VM. Thus, a GPOS can implement hypercalls to take advantage of the real-time services. This API consists of simple calls, such as start or stop a real-time task. Thus, other OSs, like Linux, can control the real-time services.

3.4 Fast Interrupt Delivery

In some cases, it is desirable to map a hardware device directly to a guest OS, e.g., a serial port or a USB device. However, in processors without proper hardware virtualization, all interrupts are handled by the hypervisor. This prevents an efficient implementation for directly mapped I/O, because it would require the processor's ability to redirect interrupts to the target guest OS without hypervisor intervention. This mechanism is called interrupt pass-through. Additionally, the hypervisor must configure an entry in its

Table 1: Example of a set of rules for a system configured with three guest OSs.

		Rules	
Guests	**DMI**	**Root Int**	**Preemptible**
RT-VM	Timer	None	No
RTOS	Timer, Serial 1	Serial 1	Yes
Linux	Timer, Ethernet	Network	Yes

page table that maps the physical device address to the intermediate physical address (see Subsection 3.2) as expected by the guest OS. Thus, the guest OS will handle interrupts and read or write to the device without any hypervisor intervention.

When an interrupt is asserted during the target guest OS's execution, it will be handled without any hypervisor intervention. However, if the interrupt is asserted during the execution of any other guest OS, the hypervisor may delay the interrupt or intercept the interrupt and schedule it to the target guest. The first option results in long delays. Our hypervisor implements a fast interrupt delivery policy to allow general-purpose OSs and RTOSs to coexist. This policy allows for the description of the system behavior with different sources of interrupts and guest OSs with different needs. For each guest OS, the designers define which interrupts are directly mapped for each guest OS. Additionally, a higher priority guest OS can be marked as non-preemptible. Table 1 shows a possible configuration for three guest OSs instances: Linux, RTOS and RT-VM. The RT-VM is marked as non-preemptable. The RTOS has the Timer and Serial 1 interrupts directly mapped. The column Root Int describes which interrupts will trigger the hypervisor when the guest is not executing. In this case, the hypervisor will intercept the Serial 1 rescheduling the RTOS. The Linux guest has the Timer and the Network interrupts sources directly mapped. The hypervisor will intercept network interrupts when the Linux guest is not executing only if the current guest is marked as preemptible. In the example, the hypervisor will preempt the RTOS to deliver network interrupts to Linux, but the same interrupts will be delayed if the RT-VM is executing. This configuration is used in Subsection 4.4 to evaluate the effectiveness of the technique.

3.5 Linux port experience

Linux/MIPS is the port of Linux to the MIPS architecture. We worked with the Linux kernel release 4.0.0 which already has proper support for the SEAD-3 platform board.

Virtualization provides a subset of the entire hardware resources to a guest OS. In this case, we choose to reserve the network card, a serial port and a small amount of the main memory for the Linux guest, while sharing the processor with one or more RTOSs. Its is important to highlight that we are not interested in sharing the network or serial port among guest OSs. To begin with, we are focused on exploring assisted virtualization benefits. However, we support a para-virtualized driver on our Linux guest for inter-VM communication purposes. It is important to highlight that the para-virtualized driver is intended for inter-VM communication only and is not required for Linux virtualization. Thus, our full-virtualized hypervisor supports Linux without any kernel modification. Additionally, we focused on avoiding hypervisor interventions during Linux guest execution as much as possible. Thus, we take advantage of the configurable privileges access to the MIPS VZ guest coprocessor 0 (GCP0). The M5150 processor allows the designer to configure privileged access to different instructions and GCP0 subsets. The designers can choose fewer hypervisor interventions, consequently less control over the processor hardware. Or they can decide for more hypervisor interventions which gives them better hardware control. Yet, some virtualized systems require more accurate hardware control, e.g., when guest OSs wish to configure different cache management algorithms. In this case, the hypervisor will keep the cache instructions privileged and control the cache operation. In our hypervisor, we allowed complete access to the GCP0 co-processor during Linux execution. However, some registers or specific bits of registers are always privileged. For example, the reduced power mode bit in the status register will always trap the hypervisor on guest write attempts. Another example is the processor identification (PID) register. When the guest OS is trying to identify the processor, the hypervisor can return a different processor identification. For example, in the M5150 processor the hypervisor can return the 4Kc processor identification limiting the guest to a compatible processor subset.

We determined the number of guest exceptions generated during the Linux kernel boot process. However, we noted that, even with the maximum level of privilege, the kernel was still generating an excessive number of exceptions due to GCP0 access to privileged registers. In fact, the Linux/MIPS kernel performs the pooling of the PID register, which is always a privileged GCP0 register in the MIPS VZ specification. In the MIPS R4000/R4400 processors before version 5.0, there is a hardware bug that avoids generating the timer interrupt if the counter register is read at the exact moment that it is matching the compare register. Thus, the Linux/MIPS always checks the PID to avoid reads to the counter register on R4000/R4400 processors. In the MIPS architecture, it is easy to obtain the PID performing a *mfc0* instruction. However, a virtualized Linux/MIPS instance will suffer a huge overhead impact. Thus, we modified the Linux/MIPS source code to avoid the frequent reads from the PID register. The modifications are restricted to three punctual routines: *can_use_mips_counter()*, *get_cycles()* and *random_get_entropy()*. As the kernel already keeps the processor identification number obtained in the early boot stages in a data structure, we just substituted the GCP0 reads to data access in memory. This punctual modification reduced the number of guest exceptions from 1,476,000 to just 6. It is important to highlight that such a modification does not imply para-virtualization, since, we do not exchange the privileged GCP0 read by a hypercall. Instead, the processor identification number is obtained from a privileged read to the GCP0, but the subsequent reads were substituted by memory accesses. The remaining 6 guest exceptions are read/write to GCP0 privileged registers including the PID register itself.

4. EXPERIMENTAL RESULTS

In order to validate our tests and conduct performance measurements we based our experiments on the SEAD-3 development board. The SEAD-3 supports MIPS processors allowing the user to evaluate the cores in a FPGA environment. The board can be used for performance benchmarking and software development. Our board is configured with a M5150 processor soft-core running at 50MHz and 432Mbytes of main memory. Additionally, the board has several peripherals including Ethernet network device, two serial ports and USB, among others. For now, our hypervisor is supporting direct mapping for both serial and network devices. We quantified four different aspects of our hypervisor: memory footprint (Subsection 4.1) virtualization overhead (Subsection 4.2), inter-VM communication delay (Subsection 4.3) and the fast interrupt delivery associated to real-time services (Subsection 4.3). The experiments environment consist of the Linux/MIPS kernel with our proposed modification to avoid the PID register polling as explained in subsection 3.5. The same kernel binary was used for both non-virtualized and virtualized experiments since our hypervisor allows full-virtualization.

4.1 Hypervisor Memory Footprint

The memory footprint is the amount of main memory that an application occupies while executing. Similar to OSs, the hypervisor footprint must be as conservative as possible. This is especially critical when targeting the IoT class of applications, where target devices tipically have severe memory limitations.

The Hellfire Hypervisor requires 35,008 bytes of memory for the code segment (using default GCC compiler optimizations). The read-only data segment needs 2,300 bytes. The data segment (global variables) is only 976 bytes. The amount reserved for the stack is 2,048 bytes. Finally, the memory segment reserved for dynamic allocation (heap) is 20,480 bytes. Thus, the hypervisor footprint during its execution is 60,812 bytes. However, the heap size requirement may vary depending on the application. The VMs and VCPUs data allocation happen dynamically during the hypervisor initialization. The data structure to represent a VM requires 56 bytes of the heap. The VCPU data structure varies depending on the inter-VM communication requirements. If inter-VM communication is not required for a virtualized system, the VCPU data structure will only need 780 bytes. Otherwise, the space occupied by the message queue will be added to the total amount required by the data structure. For example, a VCPU compiled to support a queue for five messages with 128 bytes each will result in 1460 bytes. After the initialization, heap allocation is no longer required. The maximum number of VMs allowed by the processor's hardware is 7. Thus, a virtualized system executing 7 VMs with a VCPU attached to each one will require 10,612 bytes of the heap.

KVM requires a kernel module (3 to 4MB) and the Qemu (20MB). Besides, it requires a Linux host. Xvisor requires from 4 to 16MB of RAM for execution. The footprint of both hypervisors is unacceptable for IoT devices which is typically around a few hundred kilobytes. In addition to the small footprint, our hypervisor allows more demanding applications of the IoT class to run on more powerful embedded devices such as the SEAD-3 board.

4.2 Virtualization Overhead

We analyzed the virtualization overhead caused by the hypervisor for CPU-bound and I/O-bound applications in a Linux guest. For a better understanding of the influence of our hypervisor's context-switching on guest performance, we conducted experiments for different hypervisor scheduler time slots or quantums. Thus, the non-virtualized performance was compared to scheduler quantums of 30, 20 and 10 milliseconds (ms). For all experiments, we used just one Linux guest, since, we were focused on obtaining the direct hypervisor influence over the performance. Additionally, 32MB of main memory was reserved for the Linux guest in all experiments .

CPU-bound benchmarks. UnixBench is a benchmark

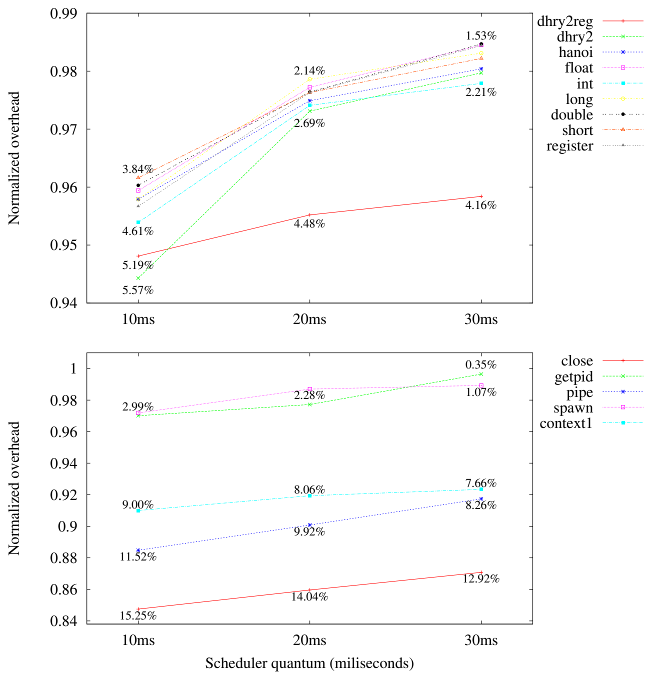


Figure 3: Performance overhead for user-land and syscall applications relative to non-virtualized performance.

used to evaluate performance on Unix-Like systems providing indicators for different system aspects. In this work, we used the byte-unixbench [1] version. We divided the benchmarks into two different groups: user-land and system calls (syscalls) applications. The user-land group is composed of synthetic applications that perform CPU-intensive computation in user-space, i.e., they do not require context-switching between the user and kernel-space. The syscall group consists of synthetic applications that make intensive use of syscalls which require context-switching between the user and kernel-space. However, these benchmarks do not require context-switching between the guest and hypervisor. Thus, the only intervention during guest execution is the hypervisor's scheduler interrupt timer. We performed each benchmark 10,000 times to determine the average execution time. Figure 3 shows the performance results for the user-land and syscall application groups. The results were normalized relative to non-virtualized performance. Thus, higher values mean higher performance, since they are closer to the native execution speed. Moreover, the percentage values shown in Figure 3 represent the performance lost. As expected, the overhead increased slightly with a smaller quantum due the increasing hypervisor intervention. In the user-land group, the most affected application was dhry2 with a performance penalty of 5.57% when compared to a 10ms scheduler quantum. Additionally, most of the applications suffered an overhead lower than 3% with a 30ms scheduler quantum. The syscall group suffered a greater performance impact since the benchmarks force context-switching between the user and kernel-space in the Linux guest. In the worst case, the syscall close resulted in a penalty of 15.25% in performance with a 10ms scheduler quantum. However, for the majority of the cases the overhead was lower than 9%.

I/O-bound benchmark. Our hypervisor implementation supports the SEAD-3's network device directly mapped to the Linux guest allowing it to receive network packets

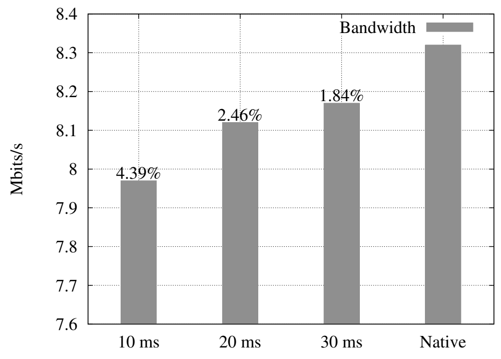


Figure 4: Iperf bandwidth results for TCP protocol comparing native versus virtualized execution with different hypervisor's scheduler *quantum*.

without any hypervisor intervention. We used the Iperf tool to measure the network bandwidth between the SEAD-3 board and a Linux host. Thus, we determined how our virtualization layer affects the I/O performance of the network device when directly mapped. Iperf consists of a client/server application originally developed by NLANR/DAST as a tool for measuring maximum TCP and UDP bandwidth performance [5]. In our experiment, we executed the Iperf server on a Linux host and the Iperf client at the SEAD-3 board. We performed the Iperf TCP bandwidth measurement for 1 minute for each experiment case. Figure 4 shows the results. The percentage values over the bars are the virtualization penalty relative to native execution. We obtained a throughput of 8.32 Mbits/second in the native Linux execution. With a 30 ms scheduler quantum, the overhead was 1.84%. As expected, the overhead increased slightly with a smaller quantum reaching 4.39% for a 10 ms quantum. Even with smaller quantums, we consider the overall results optimistic and this can be attributed to minimal hypervisor intervention.

4.2.1 *Comparative Analysis*

A direct performance comparison among different embedded hypervisors is difficult due to the large variety of embedded processors and benchmarks used for performance measurement. Thus, only relative performance between native and virtualized execution can be considered. Memory virtualization directly affects performance on virtualized systems. Moreover, virtualization on processors without hardware-assistance will cause many more TLB miss exceptions in virtualized instances than in native executions.

The KVM-Loongson [13] uses different strategies to minimize this issue such as hypercalls to avoid the trap-and-emulate technique, and large page sizes to reduce the number of TLB-misses. Thus, KVM-Loongson has an overhead between 6 and 22% for virtualized applications, compared to the non-virtualized execution measured by the SPEC CINT2000 benchmark. The KVM/ARM [4] approach uses the ARM virtualization extensions minimizing the virtualization overhead. However, KVM/ARM still implements a complete memory management mechanism. The authors claim an overall overhead around 10% using different applications and benchmarks like MySQL, Apache, and hackbench. Xvisor presents overhead up to 12.5% lower than KVM/ARM on the hackbench benchmark. Our approach does not implement a complete virtual memory management in the hypervisor level (see Subsection 3.2). However, it uses the second-stage TLB and large pages to map the guest avoiding any hypervisor intervention for memory management during guest's execution. The average overhead of the 30ms scheduler quantum, measured by the UnixBench benchmark, was 3.43% relative to the native performance for CPU-bound applications.

4.3 Inter-VM communication response time

We determined the round-trip time (RTT) of the message for our inter-VM communication mechanism between a Linux and an RTOS guest. In order to do so, an echo server application in the RTOS was implemented, i.e., an application that replayed all messages received. In Linux, we implemented a client application to send 16, 32 and 64 bytes long messages reading the server's response. The measurement of the average RTT for 10,000 messages for each different size resulted in 59.90, 59.90 and 59.99ms with a standard deviation of 0.71ms, 0.71ms and 0.69ms, respectively. An increase in the message size from 16 to 64 bytes does not impact significantly in the RTT. The imposed overhead is caused by message copies from the sender's buffer to the target VCPU and from the target VCPU to the receiver's buffer (similar to Linux pipes). Our message exchange mechanism causes an additional copy but it allows a better communication control on the hypervisor. In this experiment, we applied our fast interrupt delivery policy. Thus, once the Linux guest sent a message, the hypervisor rescheduled the VMs to the destination guest resulting in a faster response and reducing the standard deviation.

4.4 Interrupt Handling Interference on Real-time Tasks

A sequence of four experiments were performed to show how the hypervisor can be configured to support real-time services while minimizing the interrupt delivery delay on BE-VCPUs. Additionally, these tests determined how RT-VCPUs are influenced by external system interrupts. For this, the RT-VM and the fast interrupt delivery capabilities were combined in the same virtualized system. The system configuration for this experiment consisted of a Linux guest, a RTOS guest and a RT-VM. The Linux guest was responsible for network communication with external devices because it was configured with TCP/IP support and the Ethernet device was directly mapped to it. The RTOS was responsible for managing the real-time services (start and stop). The RT-VM implemented a real-time service to perform the adaptive differential pulse-code modulation (ADPCM) algorithm [3]. ADPCM is a technique to convert analog sound to digital information. It is used for voice communication and sound storage. When the algorithm is encoding or decoding for sound reproduction purposes, it must keep a constant bit rate to avoid sound glitches. Thus, the hypervisor must guarantee that concurrent events will not affect the ADPCM execution. On the RT-VM, the ADPCM algorithm was mapped to a RT-VCPU that is scheduled by the EDF algorithm. The real-time parameters of the RT-VCPU were: deadline 10, period 10, and capacity 1. Thus, a tenth of the processor time is reserved for the RT-VCPU. The remaining time is available to the BE-VCPUs responsible for executing the Linux and RTOS guests. The final system configuration consisted of two BE-VCPUs and 1 RT-VCPU.

Experiment 1. This experiment consisted in two different measurements. First, we determined how much time the ADPCM algorithm takes to encode and decode an array

Table 2: Median (m), 95th percentile (p^{th}), worst execution case (WEC) and best execution case (BEC) to ADPCM encoding and decoding and the RTT of the messages for different system configurations in milliseconds.

# Exp.	Test	m	p^{th}	WEC	BEC
1	**Enc.**	1487	1487	1487	1217
	Dec.	1187	1188	1188	917
	RTT	1.05	1.24	3.25	0.99
2	**Enc.**	1487	1488	1488	1216
	Dec.	1188	1188	1189	917
	RTT	5.78	45.5	66.6	0.78
3	**Enc.**	2082	2705	2997	1217
	Dec.	1211	2390.1	2677	917
	RTT	1.45	1.74	63.6	1.07
4	**Enc.**	1487	1488	1490	1217
	Dec.	1188	1188	1188	917
	RTT	1.48	20.81	65.8	1.05

of raw data. Second, we determined the network interrupt response time for a native Linux execution. Thus, the ADPCM algorithm was performed during 1,000 encoding and decoding cycles and the time of each execution was recorded. Table 2 shows the numeric results for this experiment, see row (1). Additionally, it was plotted in the histogram shown in Figure 5(a). During this measurement, the RT-VCPU ran without interventions. Thus, obtaining the execution time without the influence of other VCPUs or interrupts. The worst case execution (WEC) time for encoding was 1,487 ms and 1,217 ms for decoding. To measure the network interrupt response time in the Linux, the ping tool was used to generate echo request messages from a host computer to the SEAD-3 board at a rate of 20Hz. Each test case consisted of 10,000 echo request messages followed by their respective response messages (echo reply). Thus, the round-trip time (RTT) of the messages for network communication with the SEAD-3 board was determined. The histogram is showed in Figure 5(b).

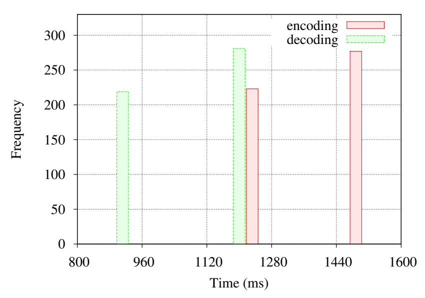
Experiment 2. The second experiment showed the behavior of the ADPCM execution time and the RTT of the messages when the fast delivery policy was not used. This is the simplest configuration, where all interrupts are postponed to the next execution of the target VCPU. This scheme guarantees a temporal isolation to the RT-VCPU since interrupts to other VMs do not preempt its execution. Line (2) of the Table 2 shows the numeric results. It confirms that the ADPCM's execution time was not affected by the external interrupts. The corresponding histogram of the ADPCM execution was omitted since it is similar to Figure 5(a). However, this scheme had an undesired effect over the RTT of the messages: the average response time increased substantially. Delays in the Linux interrupt handler caused by the hypervisor impacted directly in the RTT. The numerical results for the RTT are showed in Table 2 (see line (2)). Figure 5(c) is a histogram for 10,000 messages recorded during the experiment.

Experiment 3. The third experiment tried to minimize the RTT of the ICMP messages applying the fast interrupt delivery policy. However, the RT-VM was kept as preemptable for an evaluation of how system interrupts can interfere on the RT-VCPU's execution. Thus, both the RT-VCPU and the RTOS could be preempted for interrupts targeting the Linux guest. Similar to the previous experiments, 1,000 ADPCM encoding and decoding cycles and 10,000 messages RTT were recorded. Figure 5(e) and the numerical results (see line (3) of Table 2) show that the RTT of the messages improved significantly. Most of the RTTs were close to native execution speed, according Figure 5(b). However, some RTTs suffered a long delay, around 31 ms and 61 ms. This is the result of the fast interrupt policy with recycled quantum, i.e., when the target guest is rescheduled due to an interrupt it performs only during the remainder of the scheduler quantum. This is to preserve the scheduling time coherency to the EDF algorithm. Thus, the long delay is due to the insufficient time remaining in the current quantum. Moreover, the interrupt handling finishes only on the next Linux guest execution. On the other hand, Figure 5(d) showed that ADPCM execution time was affected, increasing significantly. This can be confirmed by the numerical results in the line (3) of Table 2.

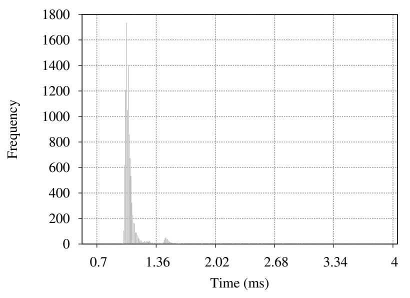
Experiment 4. The fourth experiment consisted of combining the fast interrupt delivery policy with a non-preemptable RT-VM. This system configuration is similar to that proposed in Table 1. This configuration provides the best temporal isolation for RT-VCPUs while improving the response for external interrupts. This is possible because the fast interrupt delivery policy preempts the VCPUs for faster response time. However, the RT-VM is marked as non-preemptable and the result is that its RT-VCPUs cannot be preempted. This guarantees a tenth of the CPU to the RT-VCPU executing the ADPCM algorithm. Line (4) of Table 2 shows the numerical results for this experiment. The worst execution case to encoding and decoding the ADPCM algorithm is similar to the execution time presented in line (1) (non external interference). The corresponding histogram of the ADPCM execution was omitted since it is similar to Figure 5(a). This shows the effectiveness of the temporal isolation provided by the hypervisor. In addition, the RTT of the messages improved significantly. The histogram in Figure 5(f) shows that some RTTs are spread between 0.7 and 31ms. This is caused by the RT-VCPU in the RT-VM. The ICMP messages arrived during its execution are delayed until the end of the quantum. The peak near 61ms is due to the recycling quantum scheme, as explained before.

5. CONCLUSION

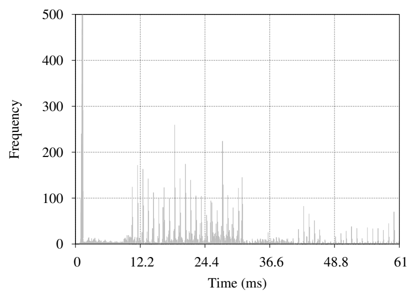
In this paper, we explored the MIPS-VZ module to implement a lightweight virtualization layer for embedded systems. A sequence of experiments was performed in the SEAD-3 development board to evaluate different aspects of the hypervisor. One important aspect discussed was the memory footprint. The hypervisor footprint is considered acceptable for IoT devices. The overhead impact on Linux was determined using benchmarks for virtualized and non-virtualized executions showing that the overall overhead is small and comparable to other hypervisors. Moreover, the hypervisor presents a low overhead for directly mapped devices. The inter-VM communication mechanism presented a considerable overhead caused by the data copies between the VMs and the hypervisor. However, the usage of the hypervisor as a communication arbiter improves the security. The temporal isolation was tested, showing that the fast interrupt policy with non-preemptable RT-VMs is an effective method to combine real-time tasks and fast interrupt handling. The overall results are promising and show that the hypervisor implementation is efficient in in terms of both performance and responsiveness. Moreover, the experiments showed that the hardware-assisted virtualization can be used to improve responsiveness while keeping the simplicity of the implementation.



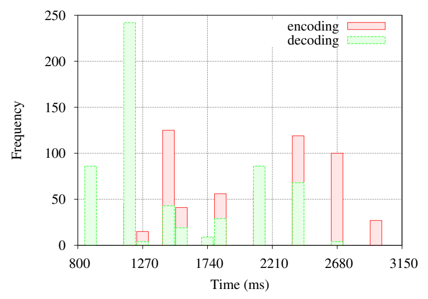
(a) Experiment 1. ADPCM execution.



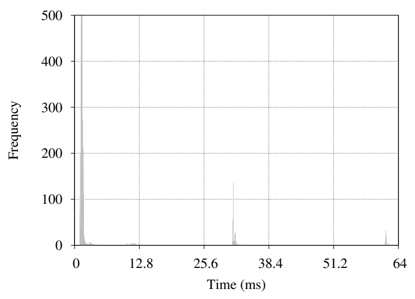
(b) Experiment 1. RTT of the messages.



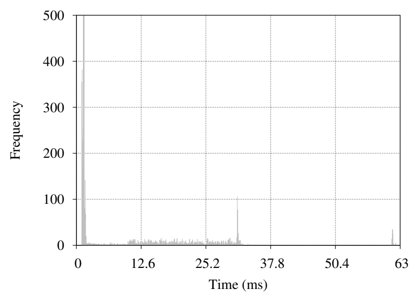
(c) Experiment 2. RTT of the messages.



(d) Experiment 3. ADPCM execution.



(e) Experiment 3. RTT of the messages.



(f) Experimet 4. RTT of the messages.

Figure 5: Histograms of the resulting measurements of the experiments.

6. REFERENCES

[1] byte-unixbench: A unix benchmark suite. Available: http://code.google.com/p/byte-unixbench/, 2014.

[2] A. Aguiar, C. Moratelli, M. Sartori, and F. Hessel. Adding virtualization support in mips 4kc-based mpsocs. In *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, pages 84–90, March 2014.

[3] P. Cummiskey, N. Jayant, and J. Flanagan. Adaptive quantization in differential pcm coding of speech. *Bell System Technical Journal, The*, 52(7):1105–1118, Sept 1973.

[4] C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[5] M. Gates, A. Tirumala, J. Dugan, and K. Gibbs. *Iperf version 2.0.0.* NLANR applications support, University of Illinois at Urbana-Champaign, Urbana, IL, USA, May 2004.

[6] G. Heiser and B. Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. *APSys '10: Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, 2010.

[7] W. H. Hesselink and R. M. Tol. Formal feasibility conditions for earliest deadline first scheduling. Technical report, 1994.

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, June 2007.

[9] Loongson Technology Corp. Ltd. Loongson 3a processor manual, 2009.

[10] H. Mitake, T.-H. Lin, Y. Kinebuchi, H. Shimada, and T. Nakajima. Using virtual cpu migration to solve the lock holder preemption problem in a multicore processor-based virtualization layer for embedded systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 270–279, Aug 2012.

[11] A. Patel, M. Daftedar, M. Shalan, and M. Watheq El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 682–691, March 2015.

[12] D. Reinhardt and G. Morgan. An embedded hypervisor for safety-relevant automotive e/e-systems. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 189–198, June 2014.

[13] Y. Tai, W. Cai, Q. Liu, and G. Zhange. Kvm-loongson: An efficient hypervisor on mips. In *Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1016–1022, July 2013.

[14] S. Trujillo, A. Crespo, and A. Alonso. Multipartes: Multicore virtualization for mixed-criticality systems. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 260–265, Sept 2013.

[15] S. Zampiva, C. Moratelli, and F. Hessel. A hypervisor approach with real-time support to the mips m5150 processor. In *Quality Electronic Design (ISQED), 2015 16th International Symposium on*, pages 495–501, March 2015.

[16] R. Zhou, Z. Ai, J. Yang, Y. Chen, J. Li, Q. Zhou, and K.-C. Li. A hypervisor for mips-based architecture processors - a case study in loongson processors. In *International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th*, pages 865–872, Nov 2013.