A Hypervisor Approach with Real-time Support to the MIPS M5150 Processor

Samir Zampiva, Carlos Moratelli, and Fabiano Hessel
Faculty of Informatics - PUCRS - Av. Ipiranga 6681, Porto Alegre, Brazil
Email: {samir.zampiva, carlos.moratelli}@acad.pucrs.br, fabiano.hessel@pucrs.br

***Abstract*—Embedded software has become a major concern in current embedded systems. Recent embedded products include dynamic software features that are driven by the users' needs. Still, real-time applications are required to execute along non real-time applications as both of them can be part of a bigger and more complex software system. Virtualization has emerged as a feasible technique for embedded systems as it provides more secure platforms, improves software design quality and reduces costs. Nevertheless, real-time and memory constraints require the development of different techniques from those widely applied to enterprise computing. However, distinguished vendors have designed virtualization extensions for their processors and a few manufacturers have adopted them. This work presents a hypervisor implementation approach with real-time support to the MIPS M5150 processor which supports hardware-assisted virtualization. The results show that the implementation allows full-virtualization and communication among virtual machines with minimal overhead while providing strong spatial and temporal isolation between virtual machines.**

***Keywords*—Embedded Systems, Virtualization, hardware-assisted virtualization, real-time.**

I. Introduction

The growing processing power of the embedded systems (ES) induced its software becomes larger and more complex to develop and maintain. Aiming to reduce the hardware cost and complexity, the system designers may choose to integrate different embedded software systems on a shared device. A technical solution for hardware sharing are virtualization technologies. Widespread for enterprise and server applications, virtualization has recently being used for ES [1]. Although server virtualization is a well-known and mature technology widely applied for commercial use, ES virtualization has its application restrict to a few scopes. The main restriction to the broad use of virtualization for ES is that its requirements differ from server and enterprise systems. Most notably, time constraints and limited resources are the main concerns around ES virtualization.

One important use case for embedded virtualization is that of running several *operating systems* (OS) on the same processor. The growing number of functionalities in ESs make many of them similar to general purpose computers, e.g. smart phones [2]. This kind of equipment requires that a *general purpose operating system* (GPOS) with rich set of functionalities is kept along a typical *real-time operating system* (RTOS) responsible for real-time support. A common solution is to execute the RTOS on a secondary processor dedicated to it. Virtualization can break the one-to-one correspondence between logical and physical system decreasing costs [3], since, it allows the execution of both OSs at the same physical processor.

The coexistence of different systems on one platform requires that its functionalities cannot interfere with each other or lead to a failure of a functionality or the whole system [4]. Thus, it is essential to keep a strong spatial and temporal isolation between systems [5]. Spatial isolation means that a system cannot interfere in the memory area of another system. Temporal isolation means that each system has its execution time reserved, and it is not allowed a system use the processor time reserved for another system. A hypervisor may be designed to overcome these problems. Usually, spatial isolation is achieved using processor's Memory Management Unit (MMU) while temporal isolation requires a carefully designed process scheduler.

In this paper, we will demonstrate an implementation approach for a hypervisor with real-time support for the MIPS M5150 processor core. Several approaches are used to improve the real-time capabilities in ES virtualization. A common approach consists to give to the RTOS the highest priority, and thus, the RTOS can preempt the execution of a GPOS [6]. However, this brings the hierarchical scheduling problem. A hypervisor schedules *Virtual CPUs* (VCPUs) and a guest RTOS over the VCPU schedules its processes or tasks. Even ensuring real-time characteristics in the VCPU level, it is difficult to ensure real-time execution to the tasks over the RTOS. Although hierarchical scheduling analysis is a well-known technique, it requires that both schedulers are modeled accordingly [7] requiring modifications in the *guest OS* (virtualized OS). Our approach avoids the hierarchical scheduling problem allowing that real-time tasks are scheduled directly by the hypervisor. This is accomplished by implementing real-time and communication facilities as *extended services* available to the guest OS[1]. An extended service is exposed to the guest OS as hypercalls[2], thus, our virtualization model mix full- and para-virtualization.

Our main contribution is to present an innovative hypervisor implementation to a recently released processor: the MIPS M5150 [8]. The M5150 is the first released MIPS processor implementing the MIPS Virtualization Module (MIPS VZ). This module adds hardware support for virtualization on the MIPS architecture. By the best of the authors' knowledge this is the first time that an implementation with performance measurement using the MIPS hardware virtualization support is published. The hypervisor supports full-virtualization of the

[1]Guest OS is a virtualized instance of an OS.

[2]Hypercall is a software call from a VM to the hypervisor. It is extensively used in the para-virtualization approach.

CPU and mix para-virtualization to provide extended services to the virtualized platform as communication among virtual machines (VM) and real-time support.

This work is organized as follows: Section II presents the related works. Section III brings a brief description of our virtualization model. Section IV describes in details the hypervisor's implementation for the M5150 processor. Performance results and measurements are showed in Section V. Finally, Section VI brings the conclusion and discussions.

II. Related Work

There are several hypervisors designed for ESs with different purposes. Such variety of approaches is expected due to the large amount of applications were ESs are required. This section presents the most relevant hypervisors implementation available for ESs and how our implementation differentiate from them.

The OKL4 is a virtualization layer that adopts the microkernel[3] approach, thus, being called Microvisor. It was developed targeting mobile devices and currently supports Linux, Android, Symbian and Windows Mobile OSs. OKL4 powered the first commercial virtualized phone in 2009, named Motorola Evoke QA4, executing two VMs on top of the OKL4 Microvisor: a Linux VM to handle user interface and a RTOS to the BREW (Binary Runtime Environment for Wireless). The microvisor is the only software layer that executes in privileged mode while GPOSs, bare-metal applications and even device drivers executes in unprivileged mode. It has the ability of execute several GPOSs and bare-metal aplications concurrently. With OKL4 is possible to execute both GPOS and a real-time environment on a single ARM processor, while providing high performance communications between them. Such scheme eliminates the need for a second ARM processor reducing the final cost and design complexity.

Crespo et. al. [10] presents XtratuM, a type 1 [4] hypervisor specially designed for real-time embedded systems aiming to achieve temporal and spatial requirements of safety critical systems. In order to meet real-time constraints, XtratuM was developed the following set of requirements: a) data structures are static which allow better control over the resources being used; b) XtratuM code is non-preemptive making the code simpler and faster; c) all hypercalls are deterministic; d)peripherals are rather managed by the VMs and; e) interrupt occurrence isolation meaning that when a VM is executing only its interrupts are enabled. Initially developed over x86, nowadays, XtratuM supports LEON2, LEON3, LEON4 and ARM processor architectures. On XtratuM, each VM is named partition and each partition supports an RTOS or a bare-metal application. There are two types of partitions: normal and system. Normal partitions have restricted functionality while system partitions are allowed to manage and monitor the state of the system and other partitions.XtratuM virtualizes not just CPU and interrupts, but, some specific peripherals using para-virtualization. If there is no need to share a peripheral between two or more partitions, it is directly mapped to a specific partition. When a peripheral needs to be shared between partitions, XtratuM implements a device driver responsible to serialize the access and the guest OS must be modified in order to implement hypercalls to the device driver.

The VHH (Virtual Hellfire Hypervisor) [11] is a type 1 hypervisor designed for the MIPS 4Kc processor and it implements full-virtualization of the CPU. It has temporal and spatial isolation among domains. Each domain can run a guest OS. The VHH support real-time applications using a para-virtualization approach. Any guest OS can instantiate a real-time application, which is then directly scheduled by the VHH real-time scheduler and is run independently of the guest OS. The MIPS 4Kc doesn't have hardware support for virtualization. In order to achieve full-virtualization of the CPU, some modifications were made in the processor architecture. All virtual memory segments were removed and the TLB-translation disabled when kernel mode is active. The processor kernel operation mode is exclusively used by the hypervisor. The guest OS and it's applications run on the user operation mode enabling the hypervisor to isolate the domains from each other. This modifications, however, create some limitations. The core modifications breaks the software compatibility with the existing OSs, i.e, despite their hypervisor provide full-virtualization, the OS must be ported to support the modified core.

The presented hypervisors implement para-virtualization of the CPU with some of them supporting full-virtualization on ARM processors. To the best of the authors knowledge, the VHH was the first hypervisor supporting full-virtualization on MIPS processors. However, it requires modifications on the MIPS architecture making it not applicable to existing MIPS processors. Para-virtualization is widely used on virtualized ES but it requires the guest OS to be modified increasing its development and maintenance costs. Our hypervisor overcomes this limitations by using hardware virtualization support now present on the MIPS architecture. It implements full-virtualization of the CPU without any modifications on the MIPS architecture. Another important difference of our hypervisor is when dealing with real-time applications. The OKL4 and XtratuM hypervisors support RTOSs and schedule them on a VM level. They do not take in consideration the individual parameters of the tasks running inside the RTOSs. In our approach the developer can use a special real-time VM that schedule its tasks directly over VCPUs avoiding the two levels scheduling problem. The VHH can also directly schedule real-time tasks, however, these real-time tasks are created by the GPOS' tasks and lack spacial and temporal isolation among them.

III. Virtualization Model

Figure 1 depicts our type 1 virtualization model. In the hardware level, it is assumed a bus-based homogeneous MP-SoC along with a shared memory. The main elements of the model are: hypervisor, Real-Time Virtual Machines (RT-VM), Best-Effort and Real-Time VCPUs and Extended Services.

The **hypervisor** is responsible for the creation and management of each VM. It provides a logic arrangement responsible for associating each VM to its VCPUs, besides, each VM has

[3]The microkernel concept consists in to reduce the OS kernel code to fundamental mechanisms, and implement the remainder system services in user level [9].

[4]On a type 1 virtualization model the hypervisor executes directly on the CPU opposed to the type 2 model where the hypervisor is a process executing in an OS.

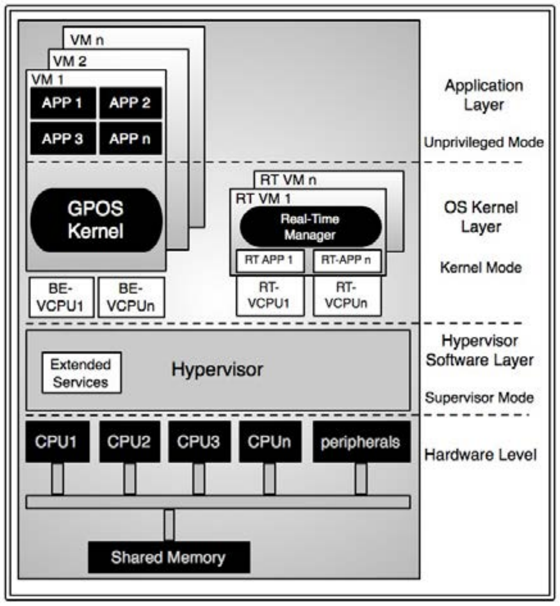

Fig. 1. The overall view of our virtualization model.

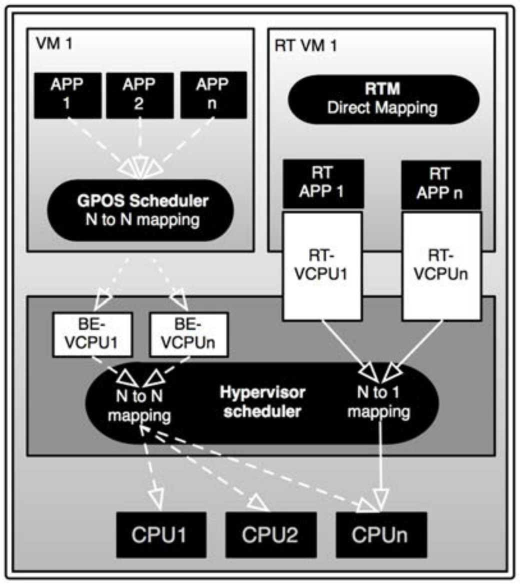

Fig. 2. Flexible Mapping model for multiprocessor embedded systems with real-time support.

its memory space controlled by the hypervisor providing memory isolation. Still, the hypervisor implements the **extended services** that are special services available to the guest OSs through hypercall mechanism. Typical services may be real-time tasks instantiation or communication mechanism among VMs. Note that our model allows full-virtualization although hypercalls are used to extend the guest OS capabilities resulting in a mixed virtualization approach.

Real-Time Virtual Machines (RT-VM) is a special VM designed to provide a strong temporal isolation between real-time services and GPOSs. The RT-VM does not support RTOSs. Instead, it implements the called *Real-Time Manager* (RTM) which supports communication facilities and primary user libraries. The RTM has the ability of map its tasks directly onto the hypervisor scheduler, i.e., it does not implement a scheduler on its level avoiding the hierarchical scheduling problem and improving performance.

Best-Effort and Real-Time VCPUs (BE-VCPU and RT-VCPU, respectively) allow a strong temporal isolation between VMs. RT-VCPUs are kept in a per CPU queue and scheduled according an EDF policy. Otherwise, BE-VCPU are kept in a global queue and are scheduled according a best-effort policy. Thus, BE-VCPUs can be executed at any idle CPU providing load balancing among the CPUs. However, aiming to guarantee temporal isolation the RT-VCPUs are pinned to a single CPU. GPOSs are mapped to one or more BE-VCPUs[5] while real-time tasks are mapped directly to RT-VCPUs by the RTM.

Direct communication among VMs is possible using shared memory areas. Still, extended services can be implemented to allow message passing mechanisms using the hypervisor as communication arbiter. Nevertheless, the model is flexible enough to support typical sockets style [12] communication among VMs. Thus, when communication with the external world is desired, the hypervisor may virtualize a network adapter emulating network capabilities to the platform.

[5]For multiprocessing OSs.

Figure 2 shows the possible flexible mapping and partitioning model for virtualized architectures based in our model. A GPOS schedules its tasks according its policies to the available BE-VCPUs. The hypervisor will schedule the BE-VCPU to any idle CPU in a N-to-N fashion. The RT-VM does not implement a scheduler; otherwise, it maps the real-time tasks directly over RT-VCPUs. During the RT-VCPU instantiation, the hypervisor inserts the RT-VCPU to the local queue of one available CPU. Thus, the RT-VCPUs are scheduled in N-to-1 fashion (N RT-VCPUs over 1 CPU).

IV. Hypervisor Implementation

This section describes in details the implementation of our virtualization model to the M5150 processor. Subsection IV-A brings a brief description of the M5150 processor. Subsection IV-B shows the hypervisor's software architecture.

A. M5150 processor

The M5150 processor was released by the end of 2013 and was one of the first core based on the MIPS32 Release 5 (MIPS32r5) specification. It is a MIPS processor fully synthesizable designed for embedded applications. The main features of the processor are: single-core, 5-stage pipeline, 32-bit address and data paths, MIPS32 and microMIPS ISA [13], 16 or 32 dual-entry joint Translation Lookaside Buffer (TLB) with variable page sizes, Multiply/Divide Unit, Floating Point Unit (FPU), upto 16 General Propose Registers (GPR) shadows sets [6] and Virtualization Module Support (VZ). Still, as part of the virtualization module, the core has two instances of the COP0: one for the guest OS and the other exclusive to the

[6]Shadow sets are copies of the normal GPR allowing to avoid the need to save and restore GPRs on entry to high-priority interrupts or exceptions.

hypervisor. In the M5150 terminology, the COP0 state for the guest OS is named *Guest Context* while the COP0 state for the hypervisor is named *Root Context* of the processor.

B. Software Architecture

The source code of our hypervisor was written mainly in C programming language, but some Hardware Abstraction Layer (HAL) parts were written in Assembly programming language. All source code has 6072 lines of code (LOC), where 3648 LOC were written in C language, and 2424 LOC were written in Assembly language for the MIPS32r5 ISA. At this point, the hypervisor can virtualize the Hellfire OS [14] and deal with the RT-VMs. In order to validate our implementation we used the MIPS Instruction Accurate Simulator (IASim), which is a hardware simulator for MIPS processors able to simulate an entire platform. It performs fast simulation aiming to deliver a virtual platform for embedded software development without the need of the real hardware platform.

Figure 3 shows the hypervisors block diagram composed by the following modules: (i) Hardware Abstraction Layer (HAL) which implements the low level Application Programming Interface (API) used to isolate higher layers from further hardware details. Some HAL parts were written in Assembly due to necessity of use specific COP0 access, TLB or cache control instructions. After reset, the M5150 processor starts the instruction fetch at address 0xBFC0_0000 in the kseg1 memory segment (non-cacheble). The boot init code is placed at this address being the first code executed after reset. It must configure the processor accordingly and copy the hypervisor code to a cacheable memory area, in this case, the kseg0 memory segment. (ii) Real-time and Best- effort schedulers, responsible for implementing the EDF (for real-time constraints) and best-effort scheduling policies; (iii) Dispatcher, responsible for dispatching the chosen VCPU to the physical CPU; (iv) The Instruction Emulation module is needed to emulate some instructions that cannot be directly executed by the guest OS, e.g., write to specific bits of the COP0 that changes the overall processor behavior as the Reduced Power mode bit in the status register; (v) VM Instantiation is used to configure and startup the VMs during system initialization; and (vi) Toolkit that reunites a collection of software facilities, such as linked-list manipulation procedures.

Hypercalls. The hypervisor implements the hypercall concept to provide extended services to the guest OSs. Hypercalls are widely used in para-virtualization based approaches, where the guest OS needs to be modified, invoking hypercalls instead of using privileged instructions. However, we use the full-virtualization technique, where the guest OS does not need to be modified in order to be virtualized, although it must use hypercalls to take advantage of our extended services, e.g., real-time services and communication among VMs. It is important to highlight that our technique *does not require any modification of the OS* to be virtualized, thus saving engineering efforts and time-to-market. Table I resumes the hypervisor extended services available through hypercalls. The services are divided into three different groups: i) *VM identification* which is a unique identification number issued by the hypervisor during startup. A VM can check its identification number using the respective hypercall as describe in Table I. ii)

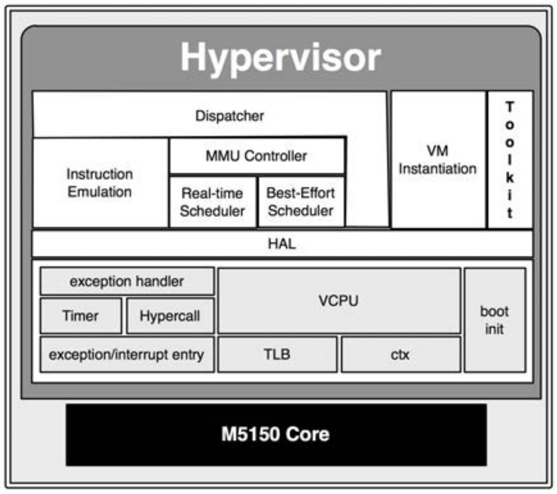

Fig. 3. Hypervisor's software block diagram.

TABLE I. HYPERCALLS AS HYPERVISOR'S EXTENDED SERVICES.

Extended Service	Hypercall	Description
VM identification	HCALL_INFO_GET_ID	Return the VM ID.
RT-VCPU Manag.	HCALL_RT_CREATE_APP	RT app. Instantiation.
	HCALL_RT_LAUNCH_APP	RT app. launch.
	HCALL_RT_DELETE_APP	RT app. delete.
Commun. Services	HCALL_IPC_SEND_MSG	Send messages.
	HCALL_IPC_RECV_MSG	Receive messages.

RT-VCPU management are a group of three hypercalls responsible to manage the RT-VCPUs. iii) *Communication services* are composed by two hypercalls designed for communication purposes. Altogether, six different hypercalls are supported and may be implemented by a guest OS when needed.

Hypervisor's initialization and execution. Figure 4 shows a view of the hypervisor's software structures after its initialization for a system configuration with two GPOSs and two real-time applications at an RT-VM. The developers, at design time, determine the number of VM instances even as the real-time services available to the guest OSs. During the boot, the hypervisor's initialization code instantiate and allocate the BE-VCPUs to the global (be_vcpu_list_ready) queue. The RTM is executed, and the RT-VCPUs are created using the respective hypercall and kept at an idle list (rt_vcpu_rt_inactive). An RT-VCPU enters in execution when the hypercall HCALL_RT_LAUNCH_APP is invoked from a GPOS. In this moment, the RT-VCPU is moved to the rt_vcpu_list_ready list. If a VCPU is in waiting state for some reason, e.g., I/O purposes, it is moved temporarily to the waiting list. In despite the M5150 be a single core processor, the RT-VCPUs are still kept in a separate queue for optimization purposes.

The hypervisor uses the GPR shadows banks intensively to avoid the need to save the GPR state during context switching. On start-up, the hypervisor checks the amount of shadows available on hardware and reserve the highest shadow page for itself. Thus, any exception or interrupt directed to the hypervisor will be handled in the highest page being unnecessary to save the GPR state for the hypervisor execution. Still, if there are enough available shadows, each VCPU is

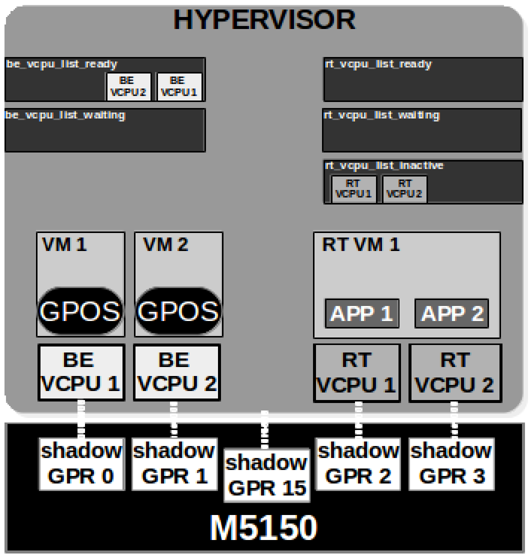

Fig. 4. Hypervisor's initial state after boot.

pinned to a different shadow page. During a context switching the hypervisor checks where the VCPU is pinned and configures the processor to use that page during guest execution. When there is not enough free pages to allocate all VCPUs, the hypervisor will save the GPR state before the context switching. Still, in the Figure 4, is possible to see the BE-VCPUS at VM 1 and 2 pinned to the GPR 0 and 1 respectively. The GPRs 2 and 3 are reserved to the RT-VCPU 1 and 2. Finally, the hypervisor reserved the GPR 15[7] for its exception handler.

Interruptions. The M5150 supports *interruption pass-through* meaning that certain interrupts may be directly handled by the guest OS without hypervisor intervention. Our implementation uses this technique to the guest OS timer interrupt. Thus, there is not additional overhead during guest OS context switches. However, specific interrupts may be shared among VMs (*shared interrupts*) which requires hypervisor intervention. A typical example is a shared network interface requiring the hypervisor to receive incoming messages and routing among VMs. Still, it may be needed to insert interrupts to the guest OS (*virtual interrupts*). As the M5150 supports virtual interrupts our hypervisor uses this technique to implement the communication services as further explained.

Communication Services. The hypervisor implements a message passing mechanism to allow communication among VMs. When a guest OS desires to support communication it must implement the extended services for communication as describe in Table I. Once the guest OS triggers the HCALL_IPC_SEND_MSG or HCALL_IPC_RECV_MSG hypercalls the hypervisor acts as an arbiter between the communicant VMs. Each VCPU implements its own incoming message queue as a circular buffer, statically allocated for performance purposes. A message destined to a determined VCPU will be copied to its queue and the hypervisor will insert a virtual interruption to the VCPU. The next time that the VCPU is executed it will attend the virtual interrupt and call the hypercall to retrieve the message. The hypervisor uses the VM identification number to route the messages among the VMs. The hypervisor requires the address, size and ID destination in order to route a message, which are discovered during the hypercall. Thus, the hypervisor does not do any assumption respecting the message formatting; this is an entire responsibility of the communicant guest OSs. For example, if a multi-task guest OS needs to demultiplex [12] incoming messages among different tasks, it may add a header to the message indicating the origin and destination task id. In this case, different guest OSs must agree about the header format.

[7]For a core with 16 GPR shadows.

V. Hypervisor overhead measurements

In order to validate our virtualization approach and evaluate its performance we used IASim [15], which is a hardware simulator for MIPS processors, instruction-accurate and able to simulate an entire platform. Still, it performs fast simulation aiming to deliver a virtual platform for embedded software development without the need of the real hardware platform. The guest OS is the HellFire OS [14] ported to the M5150 processor. Such OS was primarily designed for small ESs with strong restrictions for memory and processing power. Still, the OS supports real-time models when executing natively. Once virtualized, due to the two levels scheduling problem it does not respond accordingly to real-time. Thus, just its best-effort scheduler was used for virtualization purposes. However, we implemented the hypercalls for extended services aiming to support real-time and communication as explained in the Subsection IV-B. This section presents some performance results of our hypervisor implementation. It is organized in two subsections as follows: Subsection V-A evaluates the additional overhead caused by context switching among VMs, and Subsection V-B shows overhead penalty for communication between VMs.

A. Context Switching Overhead

Experiment 1. Aiming to determine the context switching overhead imposed by the hypervisor it was measured the number of instructions performed by a native execution of a guest OS versus its virtualized execution. We ported a benchmark application to the guest OS named ADPCM (Adaptive Differential Pulse Code Modulation) algorithm from WCET (Worst-Case Execution Time) project [16]. The ADPCM benchmark performs an adaptive differential pulse code modulation algorithm used for analog/digital conversion and was chosen because it does not requires any extended service, i.e., real-time or communication. Thus, it is possible to measure the additional overhead imposed by the virtualization layer when using exclusively full-virtualized services. The hypervisor is executing one instance of the guest OS. However, the guest OS still will be preempted at the end of each scheduler quantum, since, the hypervisor must perform the scheduler to check if another VM is ready to execute. Three experiments were conducted where the hypervisor scheduler quantum was configured to 1, 5 and $10ms$, respectively. The guest OS scheduler quantum was kept in $10ms$ for all experiments.

The Figure 5 shows the percentage overhead imposed by the hypervisor virtualization layer for a scheduler quantum of 1, 5 and $10ms$. The overhead percentage is terminated using the increased number of instruction from the native to the

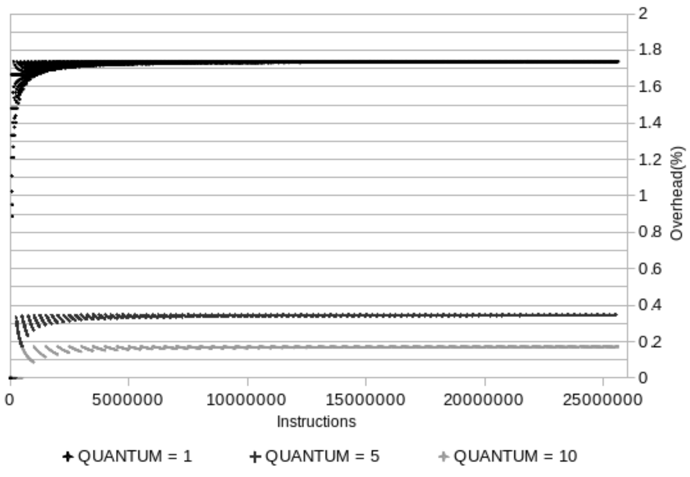

Fig. 5. Overhead for 1, 5 and $10ms$ quantum.

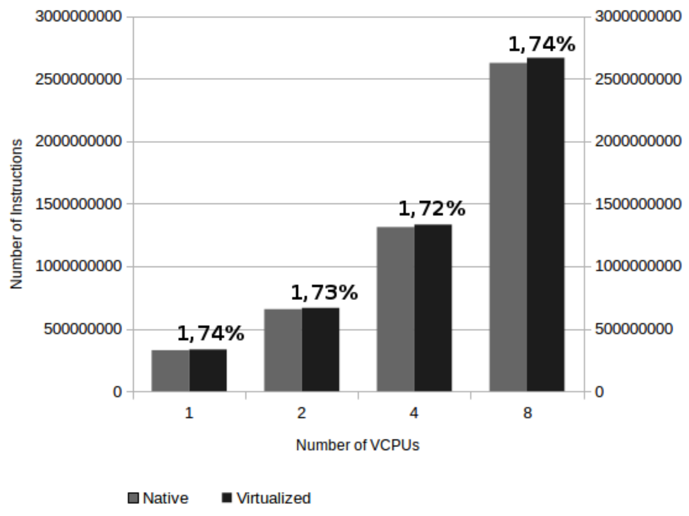

Fig. 6. Comparative overhead for up to 8 VMs.

virtualized execution. At the initial execution, the overhead is 0, since, the hypervisor's scheduler was not performed yet. After a short period, the overhead starts to increase. However, the overhead converges to 1.74% for the $1ms$ quantum with the increasing number of instructions. For a $5ms$ scheduler quantum the overhead converges to 0.34% meaning five times less overhead. Such behavior is expected since the scheduler is performed five times less frequently than using $1ms$ quantum. Still, when analysing the scenario to a scheduler quantum of $10ms$ the overhead converges to 0.17%, i.e., ten times less overhead when compared to a $1ms$ quantum. Thus, the overhead decreases linearly with quantum steps.

Experiment 2. A second experiment was conducted to determine the overhead imposed by an increasing number of guest OSs being virtualized concurrently. In this scenario, we are using the same guest OS running the ADPCM application, but, adding up to eight virtualized instances. The hypervisor scheduler quantum was kept in $1ms$. In order to determine the overhead against native execution, we accounted the total number of instructions performed during the tests. For example, the total number of instructions performed to execute one instance of the virtualized OS was compared to the total number of instructions of the native execution. However, the M5150 is a single core being impossible to execute two native instances concurrently. In this case, the same instance was executed twice sequentially, and the amount of instructions was accounted becoming possible to compare with two virtualized instances. Thus, we conducted tests for 1, 2, 4 and 8 virtualized instances aiming to determine the overhead impact when adding VMs. The Figure 6 shows that there is not overhead impact to the hypervisor's scheduler when executing up to 8 VMs concurrently. Since, the overhead was around 1.7%, the same overhead showed in the Figure 5 for one VM and scheduler quantum of $1ms$.

The optimistic performance results found in experiments 1 and 2 are partially due to the use of the GPR shadows banks. Since, the hypervisor fixes each VM to a different GPR shadow it is not necessary to save/restore the GPR during context-swiching. Of course, if the number of VMs is greater then the number of the available GPR shadows the hypervisor will save/restore the GPR increasing the overhead. This restriction can be improved if GPR shadow banks are recycled based on a LFU (Last Frequently Used) scheme. Other important hardware optimization is the interrupt pass-through that allow the timer interrupts to the guest OS trigger directly the OS's exception handler avoiding the hypervisor interference.

B. Extended Services Overhead - Communication

In order to determine the overhead imposed by the communication subsystem, we conducted an experiment to measure the number of instructions needed to deliver a message from a VM to another. The experiment consists in to execute two communicant instances of the guest OS with a synthetic application that exchange messages of different sizes: 20, 40, 60, 80 and 100 bytes long. It was measured the overhead to the HCALL_IPC_SEND_MSG and HCALL_IPC_RECV_MSG hypercalls. Still, the overhead amount to deliver a message consisting in to invoke the HCALL_IPC_SEND_MSG by the sender and schedule the receiver guest OS that invokes the HCALL_IPC_RECV_MSG to complete the deliver was determined. The guest OS implements the extended services for communication as explained in the Subsection IV-B. Such OS is a multi-task one which means it must demultiplex the incoming messages among different tasks. As explained before, the hypervisor does not do any assumption about the message formatting; its goal is to deliver a message to the destination VM using the VM ID (see Section IV). Thus, the OS can implement any message formatting as necessary. In this case, the OS implements a message header to hold the necessary information for demultiplex purposes. The header is six bytes long and it is composed by the following fields: i) *target_cpu*: Destination VCPU (2 bytes long); ii) *source_cpu*: Source VCPU (2 bytes long); iii) *target_task*: Destination task (1 byte long); and iv) *source_task*: Source VCPU (1 bytes long).

Table II shows the results. In order to send a message, the guest OS must invoke the hypercall HCALL_IPC_SEND_MSG (see Table I). Such hypercall will trigger the hypervisor copying the message from the

TABLE II. OVERHEAD OF EXTENTEND SERVICES FOR COMMUNICATION.

Experiment	Message Size (#bytes) 20	40	60	80	100
HCALL_IPC_SEND_MSG	1420	1485	1550	1615	1680
HCALL_IPC_RECV_MSG	1042	1107	1172	1237	1302
Message delivery	2594	2724	2854	2984	3114

source buffer to the VCPU's incoming queue and inserting a virtual interruption on it. As showed in Table II, the overhead to send a message increases slightly with the message size. This was expected, since, the message is copied from the source buffer to the VCPU's incoming queue. However, with just 1680 instruction is possible to deliver a message with 100 bytes long which is very optimistic. The MIPS32 implements a load/store architecture with single-cycle ALU operations, which means that the core at 100MHz performs roughly 100 millions of instructions per second. Thus, 1680 instructions represents 0.00168% of CPU time during 1 second. Aiming to receive a message the guest OS must handle the virtual interrupt inserted during the sending process. In the interrupt handler, the guest OS must invoke the hypercall HCALL_IPC_RECV_MSG which will trigger the hypervisor copying the next message available in the incoming queue to the destination buffer. After that, the guest OS will use the information in the header to deliver the message to the correct destination task. Again, the overhead increase slightly with the message size. However, all this process requires just 1302 instruction for a 100 bytes long message which is very optimistic. Finally, we measured the overhead for message delivery, i.e., the number of instructions needed to send and receive the message in the destination guest OS. Table II shows that a message delivery requires slightly more instructions to be completed than just the send or receive process, e.g., a 100 bytes long message requires 1680 instructions to be sent and 1302 to be received, but 3114 to complete the delivery. This is expected since a message delivery requires context-switching between the communicant VMs. As expected the overhead increases slightly with the message size due to intermediate buffer copies.

The message passing mechanism implemented by the hypervisor introduces extra overhead due to the required copies of the messages from a VM to another. Mechanisms like shared memory can delivery messages directly from a VM to another. However, message passing provides stronger spatial isolation among VMs, since, the hypervisor manages the message copies. Still, our implementation allows to queue the incoming messages in the destination VCPU. Thus, the VCPU can deal with multiple incoming message from different senders without requiring additional synchronization using semaphores or spinlocks.

VI. CONCLUSION

In this paper, we presented an implementation approach for a hypervisor to the M5150 processor with virtualization extensions. Such implementation supports full-virtualization of the CPU, communication facilities and real-time capabilities. The implementation was based on a virtualization model that offers real-time support and communication as extended services. The extend services consists in hypercalls that may be implemented by a guest OS if it desires to use the hypervisor's communication or real-time capabilities, otherwise, it can be virtualized unmodified. Thus, the hypervisor mix full- and para-virtualization in a mixed-approach. The implementation relies on hardware features to improve performance using extensively the GPR shadows, interruption pass-through among other hardware facilities. Results were measured in the number of instructions to determine the imposed overhead of the virtualization layer. Such results are optimistic and show that the hardware support was essential to improve the performance. As future work, we are aiming to support the Linux Kernel and conducting performance evaluation in the development board to the M5150.

REFERENCES

[1] K. Sandstrom, A. Vulgarakis, M. Lindgren, and T. Nolte, "Virtualization technologies in embedded real-time systems," in *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, Sept 2013, pp. 1–8.

[2] G. Heiser, "Hypervisors for consumer electronics," jan. 2009, pp. 1 –5.

[3] M. Asberg, N. Forsberg, T. Nolte, and S. Kato, "Towards real-time scheduling of virtual machines without kernel modifications," *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, 2011.

[4] D. Muench, M. Paulitsch, and A. Herkersdorf, "Temporal separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using pcie sr-iov," in *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, Feb 2014, pp. 1–7.

[5] T. Nakajima, Y. Kinebuchi, H. Shimada, A. Courbot, and T.-H. Lin, "Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances," in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, Jan 2011, pp. 645–652.

[6] T.-H. Lin, H. Mitake, and T. Nakajima, "Improving gpos real-time responsiveness using vcpu migration in an embedded multicore virtualization platform," in *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, Dec 2013, pp. 693–700.

[7] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constraints in virtualised services," in *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, vol. 2, July 2009, pp. 73–78.

[8] "M-class m51xx core family," http://www.imgtec.com/mips/warrior/mclass.asp, 2014, online; accessed 25-August-2014.

[9] 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.

[10] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The xtratum approach," in *Dependable Computing Conference (EDCC), 2010 European*, April 2010, pp. 67–72.

[11] 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*, March 2014, pp. 84–90.

[12] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. Prentice Hall, 2011.

[13] Imagination Technologies Ltd, "MIPS Architecture For Programmers Volume I-B: Introduction to the microMIPS32 Architecture," Tech. Rep., 9 2013.

[14] A. Aguiar, S. Filho, F. Magalhaes, T. Casagrande, and F. Hessel, "Hellfire: A design framework for critical embedded systems' applications," in *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, March 2010, pp. 730–737.

[15] Imagination Technologies Ltd, "Navigator ICS Getting Started Guide," Tech. Rep., 1 2014.

[16] "Wcet project / benchmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2014, online; accessed 29-August-2014.