A Virtualization Approach for MIPS-based MPSoCs

Alexandra Aguiar, Carlos Moratelli, Marcos L.L. Sartori and Fabiano Hessel
Faculty of Informatics - PUCRS - Av. Ipiranga 6681, Porto Alegre, Brazil
Email: alexandra.aguiar@pucrs.br, {carlos.moratelli, marcos.sartori}@acad.pucrs.br, fabiano.hessel@pucrs.br

Abstract—**Recently, virtualization techniques has been investigated as an interesting approach for complex embedded systems designs since they allow more secure systems, improve software design quality and reduce costs. However, the need to meet design constraints, mainly the real-time constraints, constitutes one of the biggest challenges that may prevent the wide adoption of virtualization in embedded systems. Industry designers and researchers believe that the use of hardware support to virtualization is a possible way of improving the system's performance and meeting its real-time constraints. In this paper we present our virtualization-aware architecture intended for MIPS processors with support to real-time applications. In our proposed approach no changes are needed in the Guest OS since we implement a full virtualization scheme. Real-time constraints are achieved by mixing the full virtualization technique with hardware support along with bare-metal application usage. Details of our virtualization platform are presented and discussed in the paper. Results demonstrate the effectiveness of our approach considering the hardware impact in terms of area, the software performance overhead, and the operating system port to allow its execution in a virtualized environment.**

I. INTRODUCTION

Nowadays new and exciting features are continually being embedded in the consumer electronics devices, increasing the number of functions and, consequently, the hardware and software design complexity. Most of these features were exclusively found only in general-purpose computers in the past. Now, to develop and download new applications, or to provide internet access through smart devices like a TV became common features in current embedded products [1]. This scenario pushes the industry designers and researchers to use more and more software layers, like embedded middleware, to achieve flexible and extensible platforms in order to respect the strict time-to-market while guaranteeing constraints such as energy consumption and real-time.

However, some design concerns are still present in this new embedded systems scenario, such as the need for better software design quality, software flexibility, maintenance and security levels while still reducing manufacturing costs. In this context, virtualization approaches, rather a successful technique exclusively applied for years on general-purpose computers, arises as a possible solution to minimize many of these problems [2]. Nevertheless, typical embedded constraints still prevent its wide adoption and much effort has been spent in order to demonstrate that virtualization can improve the overall system quality at a reasonable cost [3], [4], [5], [6], [7].

Among all these efforts to apply virtualization on embedded systems (ES), the main implementation obstacles concern some of their characteristics and needs, which can be often conflicting with non-virtualized ESes. For example, Heiser [8] highlights the need to run unmodified Guest OSes and applications besides providing strong spatial isolation to improve security. In [9], the need for low overhead components are said to be fundamental. The main problem, however, is that it is very difficult to target all such constraints at once.

Another challenge lies in the fact that these conflicting needs have a strong relationship with the hypervisor's[1] implementation that totally depends on the underlying hardware. The architecture itself, and the characteristics of its Instruction-Set Architecture (ISA) can make the use of embedded virtualization either easier or harder [10]. For instance, following the requirements for virtualizable ISAs defined by [10] the x86 architecture struggles in supporting virtualization since not all of its sensitive instructions cause traps when executing in user mode. In embedded systems, different architecture options are available and this fact is one additional obstacle to implement virtualization approaches that ca be instantiate in whole embedded systems. In this case, the designer needs to make a choice and we chose to implement our solution on a MIPS-based platform, since this architecture is widely adopted, being present in video-games, e-readers, routers, DVD recorders, set-top boxes, among others.

Besides, we can observe an increase of the software layers' importance to many systems which use each more elaborated and fully-functioned operating systems. Even though, in some cases, the need to mix general-purpose and real-time constrained applications can lead to other implementation approaches. An example can be found in smart TVs where high-level user interface must be managed to work along with communication protocols, access the cloud and meet its real-time constraints. In this case, a feasible approach consists of using different OSes for general-purpose in the main processor and an RTOS for specific functions using a co-processor. Still, the OS itself can be built in a layered fashion enabling timing constraints to be met. Moreover, it could be possible to isolate the needed real-time behavior in single bare-metal applications to be executed directly on the processor, with proper hardware support.

This paper presents a virtualized platform that provides full virtualization[2] of virtual machines and real-time execution of bare-metal applications. We use a MIPS-based platform,

[1]Hypervisor is the main controller of a virtualized platform and can also be named as Virtual Machine Monitor.

[2]Full virtualization consists of successfully emulating and simulating the underlying hardware for the virtual machines without requiring changes in the Guest OS.

in which hardware modifications were performed to allow better performance even when using full virtualization. The real-time behavior is guaranteed by extending the hypervisor presented previously in [11]. We use the concept of bare-metal applications and include a real-time scheduling policy into our hypervisor. Results show the efficiency of our virtualization approach in terms of its overheads and the correct functioning of real-time applications. Moreover, we present a case-study of the adaptation of a consolidated research OS into our platform.

The remainder of the paper is organized as follows. Section II shows some related work. Section III presents the virtualization model we consider. Section IV describes the hardware platform we've implemented and its virtualization support. Section V presents the software strategies needed to provide real-time. Section VI presents evaluation and discussions regarding the proposed platform. Section VII has final remarks of the paper and presents some future work.

II. Related Work

Virtualization is a consolidated technique which dates back more than 30 years, being primarily proposed by IBM. Throughout the years, two main approaches have been adopted to implement it successfully. In *full virtualization* an almost complete simulation of the actual hardware is performed, enabling Guest OSes to run unmodified. In *paravirtualization* Guest OSes need to be specifically modified to run, aiming to avoid the excessive amount of traps which occur when a Guest OS tries to execute a privileged instruction (when executing outside of its intended privilege ring).

However, full virtualization per se usually suffers from a large overhead from the emulation of privileged instructions while paravirtualization demands Guest OSes to be modified, which can increase both engineering cost and system's time to market. Hence, a viable solution is the use of hardware support for virtualization. For example, general-purpose processor vendors such as Intel and AMD have released, respectively, VT (Virtualization Technology) and SVM (Secure Virtual Machine) virtualization support for the x86 architecture.

As much as for the embedded market, hardware-assisted virtualization has recently been initiated. Intel itself has introduced the Intel VT technology also for its embedded processors [12]. In these case, many virtualization tasks are performed in hardware, such as memory address translation, which reduces the overhead and footprint of virtualization software improving its performance. For instance, switching between two OSes is significantly faster when memory address translation is performed in hardware compared to software. Still, it has unified the Intel VT along with the Intel AMT technology that provides remote management and maintenance capabilities, and Intel TXT that protects embedded devices and virtual environments against rootkit and other system level attacks, to provide the Intel vPro support aiming to reduce the total cost of ownership (TCO) of ESes.

ARM has also introduced a virtualization support with an extension for its ARM v7-A architecture [13]. Basically, it consists of introducing a new execution mode for the hypervisor with higher priority than the supervisor mode. This enables the hypervisor to execute at a higher privilege level than the Guest OSes, and the Guest OSes to execute with their traditional operating system privileges, removing the need to employ paravirtualization techniques. Still, improvements of mechanisms to aid interrupt handling are available, with native distinction of interrupt destined to secure monitor, hypervisors, currently active Guest OSes or non-currently-active Guest OSes. This dramatically reduces the complexity of handling interrupts using software emulation techniques and shadow structures inside the hypervisor. Finally, the provision of a System MMU (Memory Management Unit) that aids memory management and supports multiple translation contexts and two levels of address translation and hardware acceleration and abstraction.

Power.org (Power Architecture technology), announced virtualization support in the release of Power Instruction Set Architecture (ISA) Version 2.06 [14]. The document provides support for virtualization and hypervisors including a new guest mode and MMU extensions that enable the efficient implementation of hypervisors on the embedded Power Architecture platform. It allows a more efficient implementation of virtualization, partitioning of embedded systems, isolation of applications, and resource sharing.

Finally, it is possible to see that most of leading architectures already count on some kind of virtualization support. In our work, we describe a hardware-assisted virtualization platform for MIPS-based processors. Since we are dealing with a different architecture, it is not possible to fairly compare our modifications with the ones suggested for the other architectures. However, every hardware change intended for virtualization support mainly focus in decreasing the number of traps (possibly by adding another execution mode) and adjusting memory related issues.

III. Virtualization model

Our virtualization model was firstly presented in [11] and it is shown in Figure 1. On the physical level, we assume a bus-based homogeneous MPSoC with a shared memory. Above the CPUs we run the hypervisor, responsible for the creation and management of each Application Domain Unit (ADU). Into an ADU, applications can be mapped onto best-effort (non-real-time) and real-time Virtual CPUs (VCPUs), according to its needs.

In addition, an application domain can be heterogeneously multiprocessed in the sense it can count on multiple Best-effort VCPUs (BE-VCPUs) and Real-time VCPUs (RT-VCPUs), as showed in the ADU element of Figure 1. Thus, a Guest OS with proper multiprocessor support[3] can be used for the best-effort tasks. Besides, it is the responsible for instantiating an RT-VCPU for each RT-Application it desires to execute.

Figure 2 shows the possible flexible mapping and partitioning model for virtualized architectures based in our model.

[3]For a single Guest OS of a given Application Domain to manage multiple VCPUs, there is no model imposed restriction. However, this Guest OS must be implemented to support multiple processing cores.

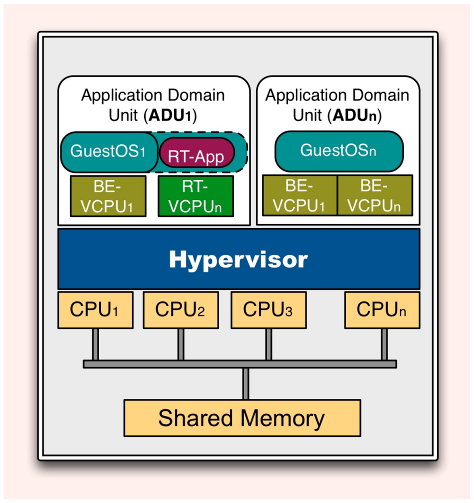

Fig. 1. Virtualization model and Application Domain for multiprocessed embedded systems

Each application domain has a given task-set associated with its VCPUs. From the VCPUs point of view, a single subset of the entire domain's task-set is available and is managed by the domain's Guest OS. This subset can be considered as the VCPU's task array. From the overall system point of view, many VCPUs (best-effort and real-time) per domain can be used as if in a matrix arrangement. Each matrix element is independently mapped onto the CPUs. Since we are providing a bus-based virtualization node, the CPUs can be represented as an array of physical processors available in the system. Thus, the separation provided by our virtualization model can ease the dynamic mapping of tasks among VCPUs (if supported by the Guest OS), VCPUs among CPUs, and even tasks among CPUs.

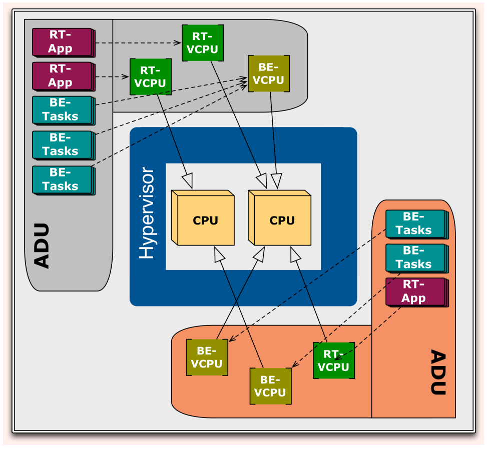

Fig. 2. Flexible Mapping model for multiprocessed embedded systems with real-time support

IV. Hardware platform and virtualization support

In this section we present details regarding our hardware base platform and the virtualization support we've added to it.

A. Hardware platform

The entire platform is described in VHDL language and the main hardware modules are showed in Figure 3. We use a Plasma MIPS CPU, which is a small synthesizable 32-bit RISC microprocessor that supports an interrupt controller, UART, SRAM or DDR SDRAM controller, and an Ethernet controller obtained from Opencores.org [15]. The Plasma CPU executes all MIPS I^{TM} user mode instructions[4] except for *unaligned load* and *store* operations. In addition, the virtualization capabilities added to the Plasma MIPS core resulted in the vPlasma MIPS, to be detailed later in this section. We adopted the Plasma processor since its VHDL description is freely available at OpenCores.org, which allows us to modify and prototype new versions of it. Also, it occupies small area, consumes low-energy and does not contain extra features, which makes it efficient to add only the structures needed to perform virtualization. Moreover, besides a licensing scheme that allows us to modify, Plasma has a software toolchain that can also be adapted when needed.

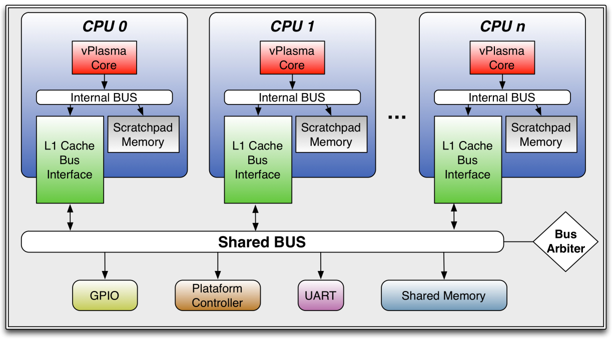

Fig. 3. Virtualization Hardware Platform main modules.

We use a multiprocessed platform in which several CPUs are interconnected through a *Shared BUS* that relies on a *Bus Arbiter*. Each CPU has a local memory (*Scratchpad Memory*) and an *L1 Cache*. CPUs have full visibility of all peripherals available at the *Shared BUS*, including the *Shared Memory*, where the ADUs are allocated. The hypervisor implements access policies to peripherals and manages the MMU for each processor, as better explained in Section V. The *GPIO* device is useful for external communication. The *UART* is the main platform's communication device and implements a typical 16550 UART device. Finally, the *Platform Controller* is used to generate interrupts among processors allowing preemptive communication calls. Besides, it is responsible for either

[4]This means that some kernel mode instructions, such as *rfe*, are not implemented in the core.

enabling or disabling cores. The platform has a hardwired-enabled core responsible for booting the system and powering the other cores up through the Platform Controller.

For inter CPU communication we use a 32-bit wide bus, which is word addressable, byte writeable, single-cycle arbitrated, half-duplex, and supports multiple masters (CPUs) where just one can communicate over the bus at a time. Decisions regarding the bus-owner at a given time are made by the Bus Arbiter. When not in use, both data and address bus' signals remain at high impedance levels. The address bus is always fed by the current master while the data bus is bidirectional, and fed by the master on writes and by the slave on reads. Writes can be performed at the granularity of bytes, half word and words in any possible endianness on the same bus, allowing mixed endian cores to execute without conflicts. A read is always performed on full words. When multiple devices request the bus at same time, the arbiter executes a master selection algorithm in a round-robin fashion, giving access to the next requesting device when the bus is free.

The L1 cache is a direct-mapped 1k-words cache organized in two banks of 64x8 entries each. It is implemented using two 512x32 BRAMS to hold data and two 64x22 LUTRAMS to hold the tags. Moreover, the cache implements the atomicity of the LL/SC instruction pair resulting in a synchronization mechanism between the processors.

B. Hardware virtualization support

Originally, the Plasma MIPS processor counts on only a single full-privileged execution mode and does not count on memory management, which is important for security and isolation. *Co-processor 0* (CP0) is responsible for controlling interruptions and timer, and can only be accessed through two special instructions: *mfc0* and *mtc0*. Such instructions are called *sensitive* [10] because their execution can change the processor behaviour. Thus, it is needed a mechanism to detect and trap such sensitive instructions and, to do that, three key features are implemented in the CPU core: (i) Memory Management Unit (MMU); (ii) Privileged Execution Mode, and; (iii) ISA's modification.

MMU. Our implementation is based on a 16-entry *Translation Lookaside Buffer* (TLB) and Figure 4 shows the MMU's block diagram.

When *enabled*, the MMU works by keeping the line of the last successful translation for both instruction fetch and data access into two 48-bit registers, named *iTLB* and *dTLB* for instruction and data access purposes, respectively. Then, in a scheme called *L1 TLB*, a comparator is used in order to validate the translation. The *L2 TLB* unifies both instruction and data and uses one 16x48 LUTRAM and three 4x28 LUTRAMs for its line tags. A successful search is performed in up to *4* cycles: a *3*-bit up-counter, enabled by a *L1 TLB* miss, has its *2* lower bits used to address the row and the tag memories, which outputs are compared with the current *ASID* and *virtual address* in order to determinate the translation. The combined *4*-bit address is used to address the 16x48 LUTRAM that feeds back the *L1 TLB*. The *L2 TLB* is responsible for

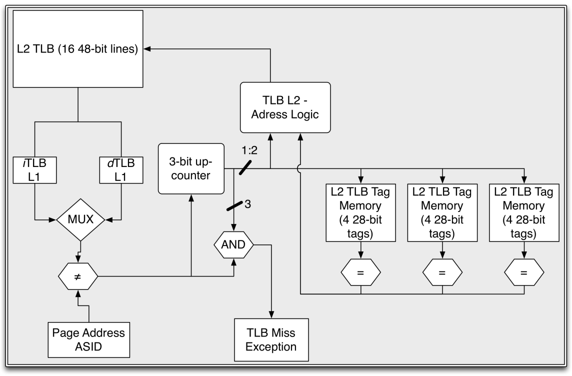

Fig. 4. MMU block diagram.

generating a *TLB miss exception* when none of the proposed translations are valid. In this case, the hypervisor feeds the *TLB L2* using the *tlbwi* and *tlbwr* privileged instructions, specially implemented to the vPlasma MIPS core. The MMU is *disabled* either at reset or whenever a software trap is performed. When desirable the MMU can be *enabled* using a special instruction called *jump register into virtual address* (JR.V), which enables the MMU and jumps to the requested address.

Privileged Execution Mode. In order to accomplish Popek and Goldberg's virtualization requirements [10] the core execution is divided in two distinct modes: *kernel* and *user*. Sensitive instructions (*mtc0*, *mfc0*, *tlbwi*, *tlbwr*) can be executed only in *kernel mode* (privileged). If an attempt to execute these instructions in *user mode* occurs, an exception is generated and the CPU enters into *kernel mode*, passing the control over to the hypervisor. In *kernel mode*, the MMU is automatically disabled so the hypevisor has full visibility of the *Shared Memory*. The *rfe* instruction is used to return the processor to its correct the state, that is, before the exception occurred.

ISA's modification. We added new instructions to the Plasma MIPS core ISA, in addition to *tlbwi* and *tlbwr*, presented earlier. Figure 5 brings a sequence diagram that shows software and hardware actions during an instruction emulation.

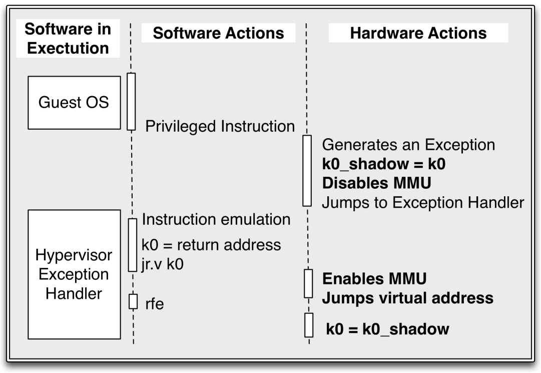

Fig. 5. Guest OS privileged instruction execution.

Originally, MIPS I ISA contains a instruction pair to return from exceptions: *jr* and *rfe*: *jr* jumps to address contained in the specified register whilst *rfe* is used in the branch delay-slot to return the processor to its prior state. However, this behaviour prevents a correct virtualization from functioning in MIPS I processors, since a register must be used to jump to the exception return address, forcing the modification of its original value, thus disrupting the normal operation of the ADU.

To solve this problem we implemented a modified version of the *rfe* instruction and added a new one to jump to a virtual address, named *jr.v*. Also, a new register (register #30 at CP0) was implemented as a shadow of the existing *K0* MIPS register[5] (named *k0_shadow*). In this case, when a exception occurs the last value of *K0* is saved in the *k0_shadow* register by the hypervisor. In the exception handling routine, registers are saved by software normally and restored before the return. However, *K0* is used to indicate the return address to the *jr.v* instruction. Such instruction is responsible for enabling the MMU, meaning that the address stored at *K0* corresponds to the virtual address which indicates the exception return address of the ADU. So, the modified version of the *rfe* instruction assigns the value preserved at the *k0_shadow* into *k0*, keeping register value consistency to the ADU.

V. Virtualization software and real-time

Figure 6 depicts a general view of our hypervisor composed of the following modules: (i) *Hardware Abstraction Layer (HAL)*, used to isolate layers, such as domain and scheduler from the further hardware details. It merges drivers interface, along with device drivers implementation, besides handling the VCPUs abstraction. The exception handler and other low level facilities are implemented directly in assembly aiming to obtain maximum performance; (ii) *Memory-Mapped I/O (MMIO)*, which manages the memory-mapped devices, including the hypercalls' subsystem; (iii) *Application Domain Unit (ADU)* described in Section III; (iv) *Real-time and Best-effort schedulers*, responsible to implement the EDF (for real-time constraints) and best-effort scheduling policies; (v) *Dispatcher*, responsible for dispatching the chosen VCPU to the physical CPU; and (vi) *Toolkit* that reunites a collection of software facilities, such as linked-list manipulation procedures.

Hypercalls. The hypervisor implements the hypercall concept to allow an ADU to instantiate several VCPUs (RT- or BE-). Hypercalls are widely used in paravirtualization 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 only to perform proper (BE- and RT-) VCPUs instantiation. 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. However, the GuestOS needs to use hypercalls only to create new VCPUs (BE- and RT-).

[5]*K0* register is normally dedicated to kernel use.

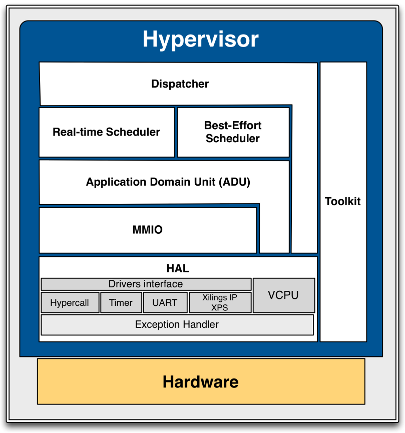

Fig. 6. Hypervisor block diagram

The system starts with a fixed number of domains configured previously by the designer, where each domain owns at least one BE-VCPU. Then, dynamically, each domain can manage its own VCPUs, as needed by the application. BE-VCPUs need to have a priority parameter define for their creation while RT-VCPUs require that real-time task model parameters, such as deadline, period, and capacity are specified to be properly handled by the hypervisor's scheduler.

In practice, a hypercall consists of a write performed by the Guest OS at a special memory address (*0xFFFFE800*) that causes a trap to the hypervisor. We also use this technique to emulate a shared peripheral, such as UART. The value written at *0xFFFFE8000* must reflect the address of a struct containing the new VCPU data, which contains parameters such as capacity and period (for RT-VCPUs) and priority (for BE-VCPUs).

The sequence flow that represents a Guest OS's execution of a hypercall to create a new RT-VCPU is depicted in the sequence diagram of Figure 7. Initially, the *create_rtvcpu()* system call is used as the responsible to fulfil a given struct (named *create_rtvcpu_cmd*) and write its address at *0xFFFFE800*. Then, when the hypervisor assumes the execution its first action consists of determining the hypercall type (in this case, an RT-VCPU creation). Next, the admission control algorithm is executed and, if there is enough system resources, the new RT-VCPU is accepted and assigned to a physical processor. Then, a proper return (indicating either success or failure) is sent back to the Guest OS, which follows its own execution flow.

Scheduler. The hypervisor implements both Earliest Deadline First (EDF) policy [16] and Best-Effort policies to deal with RT-VCPUs and BE-VCPUs, respectively. For the BE-VCPUs implementation, a single global Best-Effort queue is kept, while RT-VCPUs are kept in local individual queues per processor. The EDF is the main hypervisor's scheduler and it has a higher execution priority than the best-effort scheduler, which will not suffer from starvation since we use time reservation for it. Figure 8 presents this two-level scheduling scheme, where RT-VCPUs and BE-VCPUs are placed in

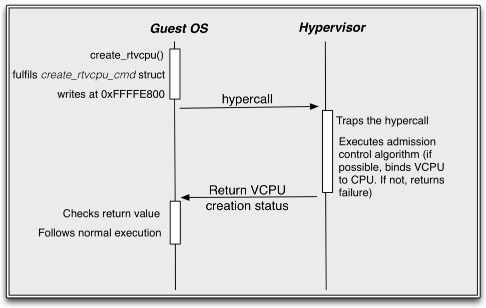
Guest OS
Hypervisor
create_rtvcpu()
fulfils create_rtvcpu_cmd struct
writes at 0xFFFFE800
hypercall
Traps the hypercall
Executes admission control algorithm (if possible, binds VCPU to CPU. If not, returns failure)
Return VCPU creation status
Checks return value
Follows normal execution

Fig. 7. Sequence diagram of RT-VCPU creation

different positions (global and local individual queues).

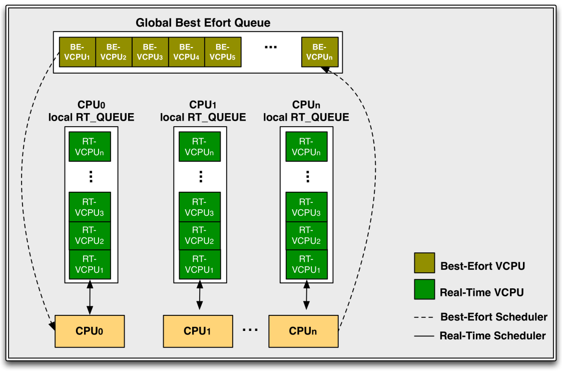

Fig. 8. Real-time and best-effort multiprocessed strategy

Scheduling Sample. In this example, we show how three VCPUs are scheduled in a system with a single processor. We have two RT-VCPUs and one BE-VCPU. For sake of simplicity, we will keep the period equal to deadline, using the EDF algorithm. We use the following real-time parameters for RT-VCPUs (p stands for period, d stands for deadline and c stands for capacity or worst-case execution time):

- **RT-VCPU 0**: p = 5, d = 5 and c = 3;
- **RT-VCPU 1**: p = 4, d = 4 and c = 1;

The overall system real-time utilization is determined by the $\sum \frac{c}{p}$ of all RT-VCPUs. In this case, we have a real-time utilization of 0.85 ($\frac{3}{5}+\frac{1}{4}$), it means that 85% of the system is dedicated to the RT-VCPUs. The remainder 15% is dedicated to BE-VCPUs.

Figure 9 shows the system scheduling when both RT-VPCUs utilize all of the capacity assigned to them. RT-VCPU *1* is scheduled first due to its nearest deadline, followed by RT-VCPU *0*. BE-VCPU only starts executing at tick time *9*, when both RT-VCPUs are waiting for their release time.

Moreover, Figure 10 shows the scheduler behaviour when RT-VPCU *0* releases itself from the processor, in an operation we named *Voluntary Preemption*. In this case, RT-VCPU *0* yields the processor in the middle of time slice *3* and the best-effort scheduler is invoked, scheduling BE-VCPU *0* until the end of the time slice, where a preemption occurs. By the time slice *7*, another voluntary preemption happens, and BE-VCPU

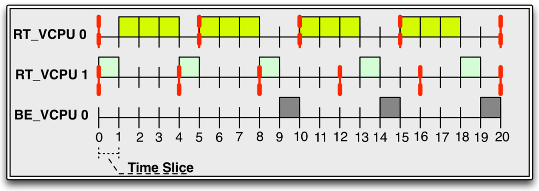

Fig. 9. Real-time scheduling sample

0 is scheduled again. Finally, at time slice *9*, both RT-VCPUs *0* and *1* are waiting for their release times, allowing BE-VCPU to be scheduled again.

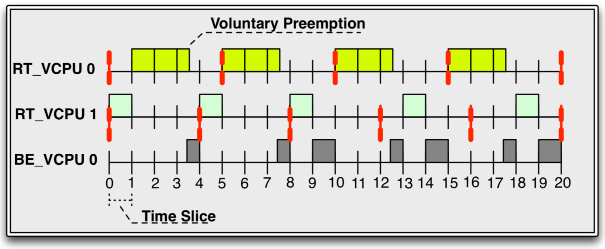

Fig. 10. Real-time scheduling sample with voluntary preemption

Time Reservation. To avoid starvation of BE-VCPUs, we adopted a time reservation strategy. In the moment of their creation, each ADU must indicate the system load capacity it desires to use. From the ADU point of view, this share of the system's entire capacity is seen as its own entire capacity. All the VCPUs (RT- and BE-) created by this ADU share its slice of the entire system's capacity. If a domain tries to allocate more than its maximum capacity, new VCPUs will fail to be created as they will not succeed through the admission control algorithm.

Admission Control. Whenever an application domain requests the creation of a new VCPU, the admission control algorithm checks if there is available physical resources that can satisfy that VCPU's requirements. In the particular case of SMP systems, even if there is enough free capacity in the entire system, this may not guarantee the new VCPU's execution since this capacity is actually fragmented among several physical CPUs. Thus, there is indeed an efficiency issue regarding the local CPU queues usage, but with the proper use of dynamic load balancing techniques this problem can be reduced.

VI. Evaluation and Discussion

A. Hardware evaluation

The VHDL description was synthesized to a Xilinx Virtex-4 XC4VLX60 FPGA Device using ISE 13.2 software. We performed three different synthesis in order to obtain the platform area occupation: the pure vPlasma MIPS core (without *schatchpad* and *L1 cache*), vPlasma processing node (with *schatchpad* and *L1 cache*) and the entire platform with four CPUs. Results are presented in Table I. The addition of the *L1 cache* and *scratchpad* represents an increase of about 1%

TABLE I
SYNTHESIS RESULTS ON A XILINX VIRTEX-4 FPGA.

	Number of occupied Slices	FPGA occupation (%)
Pure vPlasma core	2.226	8
vPlasma processor	2.658	9
Entire Platform (4 CPUs + interconnection and memory)	11.087	41
Entire Platform (8 CPUs + interconnection and memory)	21.016	78

in the total area occupation. Such area is a small price to pay considering the performance increase it brings. The entire platform with four vPlasma processors occupied 41% of area, and it is possible to synthesizes upto 8 processors in the available FPGA with 78% of occupation area. It is important to point out that each processor may contain more than one virtual processor. In this case, for many embedded applications that require high performance and therefore a high number of processors (typically applications that are implemented by NoCs), virtualization becomes an attractive alternative. Once you use less area, power consumption is lower if compared to a NoC approach, and the performance overhead does not significantly affect the entire system performance.

B. Software evaluation

We have determined the overhead of our implementation based in instruction counts for three different situations: (i) privileged instructions emulation; (ii) context switching (among virtual machines), and; (iii) device emulation (for shared devices). For the first and second cases the Guest OS execution causes a trap to the hypervisor. For the third case the Guest OS is preempted by the hypervisor and, if convenient, a new Guest OS is scheduled.

Thus, analysing the instruction count for all different instructions we emulated, we achieved an average of **230 instructions for the emulation of a privileged instruction.** We used the same technique to determine the overhead of creating and deleting VCPUs (both BE and RT). For that, we got an average of **840 instructions for the creation of a new VCPU** and of **712 instructions for the deletion of a VCPU**. The overhead of the emulation of a shared device was determined. Our emulated device is a UART port dedicated to communication to the external world. It represents a very simple device, where reading or writing a byte from/to the external word consists in an access to the 0xFFFFe000 address. In this case, the shared memory-mapped device is not mapped to a specific Guest OS, thus, a reading or writing performed in this specific address causes a trap to the hypervisor, which then emulates the device. **The average overhead detected is 245 instructions**. Although this can be considered as a very optimistic result, it is important to highlight that the more complex the device is the higher overhead it contains. Finally, we obtained the overheads of the EDF scheduler and **we detected an average of 612 instructions to preempt and schedule a new VCPU using the EDF algorithm**.

In order to validate our implementation, we elaborated an experiment where we aim to demonstrate the correct functioning of our virtualization system. The experiment consists of a producer/consumer application, a classic example of synchronization problems. Our producer/consumer implementation consists of a producer and a pool of consumers that share a common, fixed-size ring buffer as the producer allocates data in the ring buffer at a constant rate. In order to simulate variable process time, the value allocated in the ring buffer coincides with the number of time slices the consumer needs to process it. The producer/consumer is monitored by a BE-VCPU, known as management VCPU and the one responsible to control the load of the system. The ring buffer has a maximum configurable capacity of *50* integer elements. The system starts with a domain containing the management VCPU, a producer RT-VCPU and a consumer RT-VCPU. The management VCPU is allowed to instantiate three more consumers, putting the system to its maximum load, with four consumers. Figure 11 illustrates this scenario.

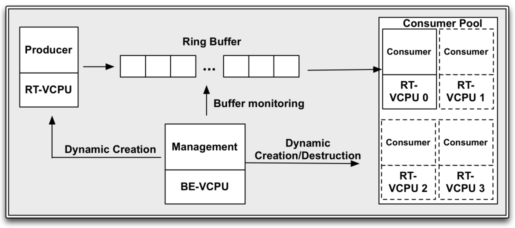

Fig. 11. Producer-consumer virtualization scenario

Firstly, RT-VCPUs from *1* to *3* are instantiated and destroyed dynamically by the management VCPU, which monitors the ring buffer when its occupation exceeds *80%*. If the buffer occupation reaches *100%*, the producer stops sampling. Still, the producer has a fixed CPU time reservation of *20%*. Each consumer, when executing, gets *10%* of CPU time. Thus, the system starts with a *30%* CPU usage rate, which can grow until *60%*, since *40%* of system capacity is reserved to best-effort tasks. Figure 12 illustrates this scenario. During start-up, the buffer occupation increases, since the producer CPU time is twice the consumers' capacity (consumers need one time slice for each buffer entry processing). Around time slice *470*, it is possible to see that the buffer reaches *80%* of its maximum capacity occupation causing the management VCPU to trigger three more consumers to avoid buffer saturation. Then, the system utilization increases up to *60%* as the buffer occupation decreases rapidly. However, around time slice *420*, the producer starts generating data that takes two time slices for the consumer to process. Thus, the system's balance is established, with an average of *33%* of buffer occupation and *60%* of system load. Still, around time slice *1700*, the producer starts generating data that takes *1* time slice for the consumer's treatment. Expectedly, the buffer occupation dwarfs. Then, the management VCPU reacts to the buffer occupation lower than *40%* and destroys two VCPUS. Then, the system gets balanced again at a *40%* CPU load rate.

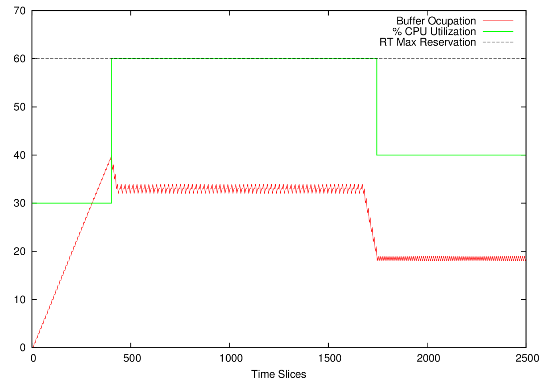

Fig. 12. Producer-consumer execution

C. *Case-study: porting HellfireOS to modified MIPS*

HellfireOS (HFOS) [17] is a real-time, micro-kernel based, highly customizable operational system. Suitable to run on low memory constrained architectures, like typical critical embedded systems. Primarily, it was designed to run on the Plasma MIPS core. In order to support the core modifications, some effort was necessary to modify the OS to the new vPlasma MIPS core. However, once the HFOS is running on the vPlasma MIPS, virtualize it is straightforward due to our full virtualization approach.

HFOS was originally designed to run on MMU-less processors. Despite the fact that vPlasma MIPS implements a MMU, HFOS does not need to be aware on it, since the MMU is managed exclusively by the hypervisor. Besides, HFOS is not affected by the new privileged execution mode. The hypervisor always schedules a Guest OS placing the processor in user mode.

Since the adaptations we performed in the Plasma processor to provide virtualization follow the MIPS R3000 Application Binary Interface (ABI) specifications, the OS modifications also follow this specification. Basically, since the *rfe* instruction was added to vPlasma's ISA, some modifications were required on the exception handler routine. Moreover, simply disabling interrupts to guarantee atomic execution of instructions is no longer enough, and a system call primitive (syscall) was implemented, allowing the OS to have similar functionality. On the OS implementation level, modifications concern exclusively the Hardware Abstraction Layer (HAL), which is the only hardware-dependent layer of the operating system. In addition, these modifications did not impact neither the OS's code size nor the data memory usage.

VII. Final remarks and future work

In this paper we presented a virtualization-aware architecture intended for multiprocessed embedded systems with real-time support where no Guest OS change is required, based on simple and constrained MIPS-like processors. We present the virtualization model we used as well as some implementation strategies concerning the hardware itself and software issues, such as synchronization primitives, hypercalls and scheduling for real-time. The main advantages of our approach are: (i) the small hypervisor size; (ii) the absence of required Guest OS's changes; (iii) the strong secure domain offered when hiding the hypervisor's memory from virtual machines and the virtual domains' memories among themselves, and; (iv) real-time support through EDF. We evaluated our platform in terms of its hardware consistency and software overheads. Besides, we show a case-study concerning the port of an existing OS into our virtualized solution. Immediate future work are focused in the improvement of the platform itself and on the implementation of a dynamic load balancing technique.

Acknowledgment

The authors acknowledge the support granted by CNPq and FAPESP to the INCT-SEC (National Institute of Science and Technology - Embedded Critical Systems - Brazil), processes 573963/2008-8 and 08/57870-9.

References

[1] Y. Zorian and E. Marinissen, "System chip test - how will it impact your design," in *DAC'2000 - Design Automation Conference*. Las Vegas, EUA: ACM Press, Jun 2000.
[2] G. Heiser, "Virtualizing embedded systems - why bother?" *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, 2011.
[3] "Mesovirtualization: lightweight virtualization technique for embedded systems," *Software Technologies for Embedded and Ubiquitous . . .*, 2007.
[4] J. Brakensiek, A. Dröge, M. Botteck, H. Härtig, and A. Lackorzynski, "Virtualization as an enabler for security in mobile devices," *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*, 2008.
[5] D. Su, W. Chen, W. Huang, H. Shan, and Y. Jiang, "SmartVisor: towards an efficient and compatible virtualization platform for embedded system," *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, 2009.
[6] A. Cohen and E. Rohou, "Processor virtualization and split compilation for heterogeneous multicore embedded systems," *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010.
[7] 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.
[8] G. Heiser, "The role of virtualization in embedded systems," *. . . on Isolation and integration in embedded systems*, 2008.
[9] F. Armand and M. Gien, "A Practical Look at Micro-Kernels and Virtual Machine Monitors," in *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, 2009.
[10] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.
[11] "Omitted for blind review," 2012.
[12] Intel Architecture Group, "Virtualization Technology Specification," http://www.intel.com/p/en_US/embedded/hwsw/technology/virtualization, 2011.
[13] ARM Architecture Group, "Virtualization Extensions Architecture Specification," http://www.arm.com/products /processors/technologies/ virtualization-extensions.php, 2011.
[14] Power Organization Group, "PowerISA v2.06 - Virtualization support," https://www.power.org/resources/ downloads/virtualization _for_Embedded_Power_Architecture.pdf, 2012.
[15] "Plasma - most mips i(tm) opcodes :: Overview," http://opencores.org/project,plasma, 2012, online; accessed 19-July-2011.
[16] W. H. Hesselink and R. M. Tol, "Formal feasibility conditions for earliest deadline first scheduling," Tech. Rep., 1994.
[17] "Omitted for blind review." 2010.