Hardware-assisted virtualization targeting MIPS-based SoCs

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

***Abstract*—Virtualization has become a hot topic in embedded systems for both academia and industry development. Among its main advantages, we can highlight (i) software design quality; (ii) security levels of the system; (iii) software reuse, and; (iv) hardware utilization. However, it still presents constraints that have lessened the excitement towards itself, since the greater concerns are its implicit overhead and whether it is worthy or not. Thus, we detail how to adapt an existing MIPS-based architecture aiming to support the virtualization principles. In this paper we present detailed information about the architecture implementation and results demonstrating its correctness and efficiency.**

I. Introduction

Multi-functioned Embedded Systems (ES) are now considered as a solid reality in everyday's lives since each more new and exciting features and devices are available. However, such a wide range of applications impacts directly on their design, constraints and goals. Also, embedded systems are increasingly counting on typical general-purpose computers' characteristics, such as the possibility of the final user to develop and download new applications onto the device throughout its lifetime [1]. Within this context, embedded software itself has become a subjacent layer in the design flow unlike older approaches, where hardware itself used to be more prominent.

In spite of that thought transformation, some traditional differences between general-purpose and embedded systems still remain [2]. Usually timing constraints are present along with limited energy consumption budgets and limitations regarding the total memory size. Still, the wide variety of predominant architectures present in ESs also contributes to increase the difficulty in their current design.

Ergo, virtualization, rather a successful technique exclusively applied on general-purpose computers, arises as a possible solution to many of these problems, as it can increase ESs' performance, software design quality and security levels while reducing their manufacturing costs [3]. However, due to the typical embedded constraints, much effort has been spent in order to demonstrate that virtualization can indeed improve the overall system quality at a reasonable cost [4], [5], [6], [7], [8].

Among all these efforts to apply virtualization on embedded systems, the main implementation obstacles concern some of these systems' characteristics and their needs, which can be often conflicting with non-virtualized embedded systems. For example, Heiser [9] highlights the need to run unmodified guest OS and applications besides providing strong spatial isolation to improve security. In [10], 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 is that these conflicting needs have a strong relationship with the hypervisor's implementation that totally depends on the underlying hardware. Thus, the architecture itself (and the characteristics of its Instruction-Set Architecture - ISA) can make the use of embedded virtualization either easier or harder [11]. In embedded systems, different architecture options are available 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, etc.

In this paper, we present how to add virtualization support to a MIPS-based architecture, in this case, specifically into the MIPS4K processor [12]. We target systems where memory protection among virtual machines is mandatory and no changes in the guest OSs are desired. Therefore, our solution contemplates implementation strategies to soften the implicit overhead of non-paravirtualized solutions as will be shown throughout this paper. As pointed in [13] and in [3], the use of architectural support to enable full virtualization is already a reality also for embedded systems[1] as one of the most used architectures, ARM, has already had its own architectural support announced [14].

The remainder of the paper is organized as it follows. Section II shows the virtualization model we are considering. Then, Section III presents detailed information about the changes in the MIPS4K processor implementation to add virtualization support. Section IV presents the simulation methodology followed by Section V where the main preliminary results are presented. Finally, Section VI concludes the paper.

II. Virtualization Model

This section discusses the model we adopted to achieve and support virtualization for mono and multiprocessed embedded systems. The overall model is depicted in Figure 1 and detailed in the remainder of this section. Some key concepts of our model are described hereafter.

[1] In general-purpose systems, manufacturers such as Intel and AMD released processors with hardware support for virtualization, increasing its usage performance. Namely, Intel-VT (Virtualization Technology) and AMD SVM (Secure Virtual Machine).

978-1-4673-2789-3/12/$31.00

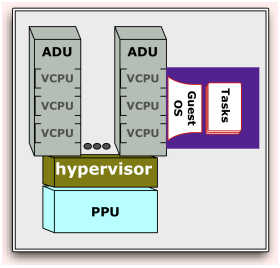

Fig. 1. Virtualization model for embedded systems

- ***Application Domain Unit - ADU***[2]. Each Application Domain Unit corresponds to a virtual machine and is intended to be used to divide the system into specialized pieces.
- ***Virtual Processing Unit - VCPU***. Each application domain can be composed by one or several Virtual Processing Units, which are individually scheduled onto physical processing units. Thus, different mapping strategies can be used. For example, VCPUs from the same virtual domain could be placed onto different physical processors to increase their performance. Still, a VCPU contains a copy of the physical processor's registers.
- ***Guest OS and Tasks***. Guest OSs must be able to execute their own task-set and no modifications in their source code are needed. This occurs because we base our platform in the use of highly optimized implementation to the hypervisor and perform emulation at low overhead cost.
- ***Hypervisor***. The core of our virtualization proposal is carefully implemented aiming to reduce the overheads of a virtualized platform. It manages the creation and execution of VCPUs and Application Domains. Besides, the hypervisor is responsible for an efficient scheduling scheme where the physical processing units are always aware of the next VCPU that needs to be executed, decreasing their idle time.
- ***Physical Processing Unit - PPU***. We propose the virtualization of a MIPS-based platform. We expect this model to be extended to multiprocessed embedded systems connected through a bus. Although we are aware of the limitations of bus-based architectures, we intend to provide nodes for future use in cluster-based multiprocessed embedded systems [15]. The quantity of physical nodes can be limited by the bus implementation's constraints.

The overall model presented in Figure 1 has a large dependence on the real implementation to be worthy and we present our strategies regarding the processor in Section III. We adopted the concept of VCPUs and PPUs as it allows more flexible mapping strategies.

Initially, each virtual domain has a given task-set, associated with its VCPUs. However, 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 GuestOS. This subset can be considered as the VCPU's task array.

From the entire system point of view, we have the possibility of many VCPUs per domain, as if in a matrix arrangement. Each matrix element is independently mapped onto the PPUs. Since we are providing a bus-based virtualization node, the PPUs can be represented as an array of physical processors available in the system.

Thus, the separation provided by the virtualization model we propose can ease the dynamic mapping of tasks among VCPUs (if supported by the GuestOS), VCPUs among PPUs and even tasks among PPUs. Figure 2 depicts this flexible mapping model for virtualized architectures.

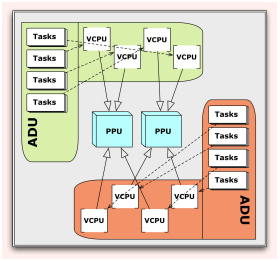

Fig. 2. Flexible Mapping model for multiprocessed embedded systems

A. Related Work Concerns

We are using a mixed approach that combines full virtualization and native execution, as this appears to be a valid and ideal approach for embedded systems [13]. We are aware of the challenges since the emulation's overhead can become prohibitive, but we believe that appropriate hardware support must fill that gap. Still, in spite of the fact that most related work deal with paravirtualization solutions, we believe that some are worth to be mentioned.

EmbeddedXEN Project is an academic project of the XEN.org research group where the main target regards embedded real-time applications executed in ARM cores. It creates a page table for each Guest OS when the guest domain is created, in order to support virtual memory systems. Though some RTOSs do not use any virtual memory technique, using the physical memory itself, the hypervisor can map the physical memory allocated by a guest RTOS into the same virtual memory, statically. At run time, the guest OS is executed as if it was using a physical memory, being isolated one from another by the page table provided by the hypervisor. This is a very simple approach which enables the use of unmodified OSs [16]. We use a similar concept to hide the hypervisor from the virtual machines with the advantage that our approach requires no changes in the guest OS.

OKL4. Implemented by OK Labs (Open Kernel Labs), it is based in an L4 family microkernel [17]. It has a high performance IPC (Inter-process communication) mechanism, which helps the low overhead virtualization. A system call activates the microkernel exception manager, converting this event into an IPC message to the guest OS. The client deals

[2]Also referred during this paper as *virtual domain* and *application domain*

with this process as a normal system call and the answer is returned through another IPC message [17]. In comparison to our approach, OKL4 claims to need less than 12KLoC (lines of code) whereas our solution counts on around 2KLoC. Although OKL4 offers some features we are still working on, the fact that we did not base our hypervisor on a microkernel contributes to this significant difference.

SPUMONE. A virtualization layer that works with paravirtualized systems but claims to have small engineering cost in terms of needed modifications in the guest OS[18]. The solution provides the VCPU and the idea that multiple virtual processors can be associated with a single application domain. However, SPUMONE provides a virtualization layer that executes in privileged space as does the guest OS. That is an important difference to our approach. We designed a hypervisor that is totally invisible to the guest OS and does not require any engineering cost, once the guest OS can execute partially directly on the physical processor.

III. MIPS4K MODIFICATION AND VIRTUALIZATION SUPPORT

In this section we present our implementation of a MIPS-based processor to include the virtualization proper support. The MIPS4K family is formed by three members: the 4Kc™, 4Km™, and 4Kp™ cores. The cores incorporates aspects of both MIPS Technologies' R3000® and R4000® processors although they differ mainly in the type of Multiply-Divide Unit (MDU) and the Memory Management Unit (MMU).

In this case:

- the 4Kc core contains a fully-associative Translation Lookaside Buffer (TLB)-based MMU and a pipelined MDU;
- the 4Km core contains a fixed mapping (FM) mechanism in the MMU, which is smaller and simpler than the TLB-based implementation used in the 4Kc core, and a pipelined MDU (as in the 4Kc core) is also used, and;
- the 4Kp core contains a fixed mapping (FM) mechanism in the MMU (like the 4Km core), and a smaller non-pipelined iterative MDU.

Figure 3 depicts the most relevant blocks on the MIPS 4Kc core: (i) Execution Core; (ii) Multiply-Divide Unit (MDU); (iii) System Control Coprocessor (CP0); (iv) Memory Management Unit (MMU) and TLB; (v) Cache Controller; (vi) Bus Interface Unit (BIU); (vii) Instruction Cache (I-Cache), and; (viii) Data Cache (D-Cache).

Among these blocks, there are a few points we need to highlight. First, the CP0 in the responsible for controlling the TLB, the cache protocols, the processor modes of operations and interruptions. Still, this CP0 contains 32 registers that differ from the 32 general-purpose registers contained in the MIPS architecture. Finally, these specific CP0 registers can only be accessed through the use of the privileged instructions *mtc0* and *mtf0*. Whenever these instructions are executed in User mode, a trap is generated.

In our work, we adopted the 4Kc core due to necessity of memory management within our virtualization model. In the following sections we describe the architecture we used and the modifications that were needed to support virtualization.

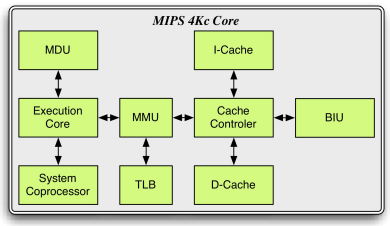

Fig. 3. MIPS 4K core

A. Memory Management

The MMU in a 4K processor core is conceived to translate any virtual address to a physical address before sending requests either to the cache controllers for tag comparison or to the bus interface unit for an external memory reference. This translation is a very useful feature for operating systems when leading the physical memory to accommodate multiple active tasks in the same memory. Other features handled by the MMU are memory areas' protection and the definition of the cache protocol.

In the 4Kc processor core, the MMU is based in a TLB that consists of three address translation buffers: (i) a 16 dual-entry fully associative Joint TLB (JTLB); (ii) a 3-entry instruction micro TLB (ITLB), and; (iii) a 3-entry data micro TLB (DTLB). Thus, when an address is translated, the appropriate micro TLB (ITLB or DTLB) is accessed first. If the translation is not found in the micro TLB, the JTLB is then accessed. If there is a miss in the JTLB, an exception is taken.

Still, all the 4K processor cores support three modes of operation: (i) **User mode**, mostly used for application programs; (ii) **Kernel mode**, typically used for handling exceptions and privileged operating system functions, including *CP0* management and I/O device accesses, and; (iii) **Debug mode**, used for software debugging usually within a software development tool. For sake of simplicity, we are not considering such mode in this study.

Finally, it is important to highlight that the address translation performed by the MMU depends on the mode in which the processor is operating. For example, Part A of Figure 4 depicts the differences between the memory segments that can be seen according to the active processor mode. It is possible to observe that whilst, in kernel mode, several segments are available (from *kseg0* to *kseg3*, including the *kuseg*), in user mode of operation, only the *useg* (with virtual addresses equivalent to the *kuseg* segment) is available.

Virtual Memory Segments. Originally, the MIPS 4K processor contains virtual memory segments, which are differently used depending on the mode of operation, as briefly discussed. Figure 4 shows the segmentation for the 4 GB virtual memory space addressed by a 32-bit virtual address for both *user* and *kernel* modes of operation. Initially, the core enters into the kernel mode during reset and whenever an exception

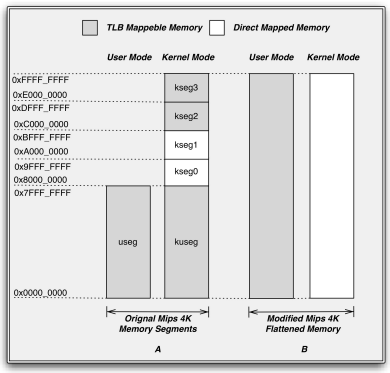

Fig. 4. MIPS 4K memory management for User and Kernel modes of operation

is recognized. While in Kernel mode, the software has access to the entire address space, as well as to all CP0 registers. On the other hand, User mode accesses are limited to a subset of the virtual address space (0x0000_0000 to 0x7FFF_FFFF) and can be inhibited from accessing CP0 functions. Still, while in User mode, virtual addresses 0x8000_0000 to 0xFFFF_FFFF are invalid and cause an exception whenever they are accessed.

This virtual memory segments scheme, which is adopted by MIPS 4k core, is very useful for a non-virtualized operating system. In this case, the OS can keep the user application and the kernel isolated by running them in different segments. Typically, the user applications run in User mode in a segment named *useg*, which allows the isolation of the OS software components from user applications with possible malicious behaviour. Besides, in the 4Kc core the OS can isolate the user applications from each other through the limited memory visibility for each application provided by the TLB. This is adopted so a user application with unpredictable behaviour does not influence other system applications.

Still, an important motivation to use virtual memory segments consists in allowing the OS to have privileged access in certain memory areas. For example, the Exception Vector (memory address where the beginning of handler routines for exceptions are placed) located at 0x8000_0000 coincides with the start of the *kseg0* segment, showed in Figure 4. Also, another interesting example is the *kseg1* where the cache is disabled to allow direct access to the registers of memory-mapped I/O devices. In this case, both segments' addresses are not eligible to be mapped by TLB, thus, they have a fixed-mapping where both segments *kseg0* and *kseg1* are mapped to the physical address 0x0000_0000.

Although the virtual memory segments scheme is strongly recommended to non-virtualized systems, since it increases software reliability, it brings undesirable restrictions to a scenario where virtualization is desired indeed. MIPS 4K core does not count on a special execution mode for hypervisors and, due to the ring de-privilege situation, the only piece of software that can be executed in privileged mode is the hypervisor itself while the Guest OSs will execute in a simple User mode.

Specifically, analysing the 4K core, it means that only the first 2GB of the virtual memory will be available to the virtual machines. A Guest OS running in the User mode will not be able to address virtual memory above 2GB. The second - and very critical - limitation can represent a major barrier to achieve virtualization in MIPS 4K core: the fixed-mapping of *kuseg0* and *kuseg1* segments. In this case, the hypervisor needs to register its exception routine under the Exception Vector address (at 0x8000_0000) in order to take the control of the execution of privileged instructions by the Guest OS, as well as hardware interruptions, TLB misses and other system conditions.

On the other hand, a Guest OS will try to register its own exception handler routine, what conflicts with the hypervisor implementation and possibly with other Guest OSs. Since the Exception Vector is located at a fixed-mapped address, the hypervisor is not able to move the virtual address 0x8000_0000 to a different physical address attending the Guest OSs' needs. The same scenario description can be applied to the *kseg1* segment, when the hypervisor tries to virtualize a given device.

In this case, providing virtualization in a system under such circumstances implies in complex modifications in the Guest OS, in a technique named as paravirtualization. However we aim to build a full-virtualization system where no software efforts on the Guest OS are required whatsoever.

Therefore, aiming to support full-virtualization on a MIPS 4Kc core, we propose two main modifications on the processor's core:

- removing the all virtual memory segments, specially the fixed-address segments (*kseg0* e *kseg1*), and;
- disabling the TLB-Translation when the Kernel mode is active.

The removal of all virtual memory segments implies that no virtual memory address is mapped to the physical memory when the TLB does not have a valid entry. However, once the TLB translations have been turned on and a TLB flush routine has been executed, there is no way to turn the TLB off again. This imposes that a valid entry needs to be kept in the TLB in order to map the hypervisor area to the physical memory.

Such scheme is not transparent for a Guest OS that tries to configure its own TLB entries. Then, to avoid such conflicts we have modified the MIPS 4Kc core so the TLB is turned off whenever the Kernel mode is active. In this condition the modified core translates each single virtual address directly to the matching physical address, thus giving full visibility of the memory only to the hypervisor.

Finally, we extended the visibility of the virtual memory in User mode to 4GB allowing the Guest OS to require addresses above 0x7FFF_FFFF. This is necessary when the Guest OS tries to access either the Exception Vector or a memory mapped device[3]. The new memory map for both User and Kernel modes are depicted in Part B of Figure 4.

[3]It is important to highlight that the Guest OS executes in User mode.

B. Logical Memory Organization

We implemented the hypervisor also to be responsible for controlling the memory visibility to each virtual machine using the TLB correctly. In terms of logical memory organization, Figure 5 depicts how we divided the implementation of our hypervisor into four different logical areas: (i) hypervisor private area; (ii) hypervisor scratchpad, which holds the hypervisor's stack; (iii) exception vector that contains the MIPS 4K exception vector, and; (iv) available memory that is where virtual machines can be allocated.

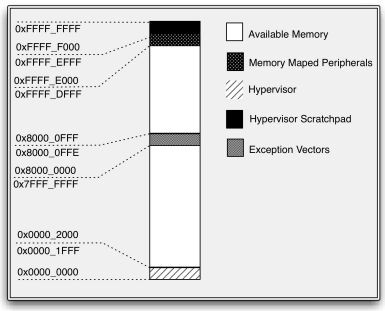

Fig. 5. Hypervisor memory logical organization

C. Exception Vector

MIPS 4K contains a fixed address designated for the Exception Vector starting at 0x8000_0000 (when the processor is operating in any mode except for the debug mode). This causes an address conflict between the hypervisor and the Guest OSs because both of them try to register their exception routine at the same address. Thus, to solve this problem it is needed to create a virtual mapping to the Guest OSs, where the physical address 0x8000_0000 is mapped to a virtual address. So, the hypervisor can register its exception routine at the physical address 0x8000_0000 while the Guest OSs use a virtual address, as described in Figure 6.

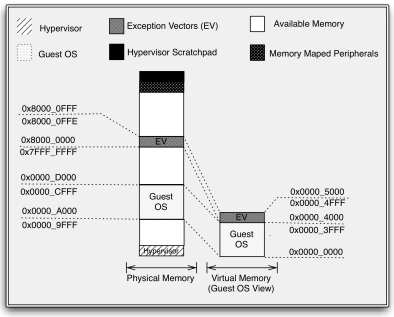

Fig. 6. Exception Vector modification

D. Exception Return

The Guest OSs run in User mode, as the MIPS 4K core generates an exception whenever a privileged instruction is executed outside of its intended privilege level, enabling the hypervisor to intercept such instruction and emulate it. Then, after the software emulation of the privileged instruction occurs, the hypervisor must return the control to the Guest OS. In this context, the ERET instruction (MIPS R4000) is used to return from an exception and the return address used by it is programmed at the EPC (Exception Program Counter). The EPC register at CP0 ($14) has the virtual address of the instruction that was the direct cause of the exception. The hypervisor accesses the address contained in the EPC register to find which instruction should be emulated. After this, the EPC register is incremented to the address of the next instruction and ERET instruction is performed. Figure 7 depicts which software and hardware actions are performed when a privileged instruction needs to be executed by the Guest OS.

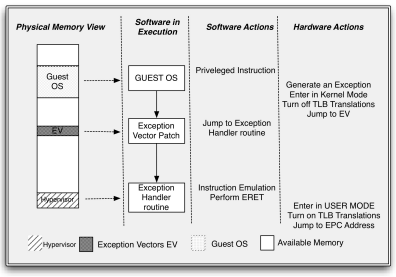

Fig. 7. Guest OS privileged instruction execution

E. Timer

For timing purposes we use the internal MIPS 4K core timer. It is programmed to generate a hardware interruption which causes the control to be assumed by the hypervisor that is responsible for scheduling a new Guest OS.

F. Memory-mapped peripherals

Currently, our hypervisor deals with memory mapped peripherals using either directed mapped or shared approaches. In the first case, the directed mapped peripheral technique is desirable when there is a need for high performance and/or when there are real-time constraints. The hypervisor maps the peripheral directly to a Guest OS and any requests from other Guests are simply denied. In such way, no overhead is added when the Guest OS accesses its directly mapped peripherals. Thus, the implementation of this technique consists in mapping the memory region where the peripheral is located to its Guest OS owner, by using the TLB. This guarantees that accesses to a peripheral by its Guest OS owner do not trap to the hypervisor, whereas not allowed Guest OS accesses trap to the hypervisor, generating an exception that is treated accordingly.

The shared peripheral approach is desirable for peripherals that are needed by more than one Guest OS. For instance, serial ports or ethernet controllers can be considered as shared peripherals because they allow connectivity to the external world and can be used by several Guest OSs. This approach requires a more complex treatment from the hypervisor point

of view. A shared peripheral does not have its memory area mapped for any Guest OS specifically, that is, the peripheral memory area is unmapped in User mode. An access to such area by a Guest OS causes a trap to the hypervisor that can identify where is the request coming from and then emulates the peripheral. This means that the hypervisor needs to implement a device driver specifically for each shared peripheral.

A memory mapped peripheral can be a GPIO pin, a serial port, an Ethernet controller, or even a high-speed PCI-e peripheral. The decision concerning the placement of a certain peripheral, if either shared or directly mapped, occurs at design-time. For instance, if Ethernet capabilities are desired for more than one Guest OS, it might be interesting to share this device. Otherwise, if a single Guest OS is the responsible for all the Ethernet communication, probably the best decision is to map it directly. Still, high-speed or real-time constrained peripherals should be directly mapped to a specific Guest OS due to the achievement of lower overhead and better response time rates.

G. Issues and future improvements

Currently our implementation is concerned in a main point: to offer virtualization support in a MIPS architecture that requires no modification in the Guest OS. We are aware that these modifications cause the break of compatibility between the MIPS 4Kc core and the MIPS 4Kc with virtualization support. This means that, in order to be executed in our platform, the Guest OS does not need to be changed or paravirtualized to use specific hypercalls (system calls provided by the hypervisor): it simply needs to be ported to MIPS 4Kc core.

However, at this time, unfortunately, it is not possible for a Guest OS to be executed *directly* on the MIPS 4Kc with virtualization support without the hypervisor, since we need it to manage the TLB fulfilment correctly. However, after this first version has been implemented and deployed we are working on the development of an *extension* for the MIPS 4Kc core where the compatibility before the virtualization support is still kept.

IV. Simulation methodology

To simulate our platform we used OVP [19], which is a hardware simulator written in C language, instruction-accurate, open-source and able to simulate an entire platform. OVP offers a large open-source model database, supporting several processor families (like MIPS, ARM and PowerPC) besides many peripherals. Still, it performs fast simulation aiming to deliver a virtual platform for embedded software development without the need of the real hardware platform.

In our case, the implementation of our virtualization technique requires several modifications in the processor core. Currently, we have no HDL implementation available of a MIPS 4K core. Besides, there is no MIPS core with virtualization extensions that could allow full virtualization technique at the present time. So, we are proposing a MIPS 4K core modified that allows full virtualization to be achieved. In this scenario, the OVP simulator and its open-source models allow us to implement the new processor's core behaviour and simulate our software stack.

V. Results

Given the lack of a hardware implementation of the architecture and even a cycle-accurate simulator, no real performance evaluation is possible. Thus, tests were only performed using the OVP simulator, which does not model neither memory access nor cache timing correctly. However, the resulting instruction counts can still be used to get an approximate idea of the performance score we got besides assuring that the implementation works as expected.

Therefore, 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 cases (i) and (ii) the Guest OS execution causes a trap to the hypervisor. For the case (iii) the Guest OS is preempted by the hypervisor and, if convenient, a new Guest OS is scheduled.

The instruction counts were obtained by configuring the OVP simulator to output the exact sequence of instructions executed by the core in an understandable assembly code format. Such feature increases the simulation time but is useful for a detailed analysis of the execution sequence or for debug purposes. Following, we show a sample of the privileged instruction sequence emulation:

```
1 - 0x00000070 : mtc0    t0,c0\_status
2 - 0x80000180 : sw      k0,-2048(zero)
3 - 0x80000184 : lui     k0,0x0
4 - 0x80000188 : addiu   k0,k0,228
5 - 0x8000018c : jr      k0
6 - 0x80000190 : lw      k0,-2048(zero)
7 - 0x000000e4 : sw      k0,-2024(zero)
9 - ...
10 - 0x00000228 : lw      sp,116(k0)
11 - 0x0000022c : lw      k0,104(k0)
12 - 0x00000230 : eret
13 - 0x00000074 : mtc0    zero,c0\_cause
14 - 0x80000180 : sw      k0,-2048(zero)
```

Code *line 1* contains a privileged instruction which is being executed by a Guest OS[4]. The processor core switches to Kernel mode and jumps to the exception vector at the physical address 0x80000180 (exposed in code *line 2*). The exception vector routine jumps to the hypervisor specific handler routine (*line 5*) at the physical address 0x000000e4 (*line 7*). The specific handler routine code was resumed among *lines 7* and *11* for sake of simplicity. Then, *line 12* shows the hypervisor returning the control to the Guest OS using the ERET MIPS R4000 instruction explained previously. This instruction jumps to the address configured in the EPC register at CP0 and switches the core to User mode. The next instruction in the Guest OS is another privileged instruction (*line 13*) at virtual address 0x00000074. This causes a new exception trapped by the hypervisor and a repetition of this sequence.

[4]the Guest OS is being executed in User mode, since the address 0x000000070 is a virtual address

Thus, analyzing the instruction count for all different instructions we emulated, we achieved an average of **220 instructions for the emulation of a privileged instruction.** We used the same technique to determine the overhead of a context switch among virtual machines. Context switches between applications on the same Guest OS will not trap to the hypervisor and there is no overhead. For the sake of simplicity, at the present time, our scheduling algorithm running in the hypervisor is a round-robin algorithm. We are planning to implement an EDF scheduling algorithm to handle real-time constraints in the future. For now, **we detected an average of 420 instructions to preempt and schedule a new VM**.

Finally, 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. As discussed previously, a 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 260 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 estimated the number of Lines of Code (LoC) needed by the entire hypervisor: around 2KLoC written in both C and Assembly languages. In this case, around 500 lines are described in Assembly language and represent the Hardware Abstraction Layer (HAL). The rest, entirely written in C Language, is mainly divided in around 150 LoC dedicated to the round-robin schedule algorithm, about 600 lines to implement the VCPU concept, Timer and IRQ emulation as the rest is responsible for other routines.

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.

VI. CONCLUSION

In this paper we presented a virtualization-aware architecture intended for embedded systems. We present the virtualization model we used, which is based in key concepts, such as the Application Virtual Domain and the Virtual Processing Unit. We also presented detailed implementation information regarding the modifications needed in a MIPS 4Kc core to provide support to full virtualization where no change in the Guest OS is required. The hypervisor we developed is responsible for some memory management aiming to keep itself hidden from the Guest OSs. Our approach combines the concepts of full virtualization and native execution. Therefore, the Guest OS must be ported to the MIPS-based processor with virtualization support we used, since the execution of non-privileged instruction executions are ran, directly, onto the physical processor. However, whenever privileged instructions are intended, the hypervisor assumes the control and emulates it. The main advantages of our approach are: (i) the small hypervisor size (around 2KLoC); (ii) the absence of required Guest OS's changes, and; (iii) the strong secure domain offered when hiding the hypervisor's memory from virtual machines and the virtual domains' memories among themselves. Results were taken aiming to measure some critical hypervisor's overhead when (i) executing the emulation of privileged instructions; (ii) performing context switch among virtual machines, and; (iii) when used shared peripherals. Limitations and future work were covered back in Section III-G.

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] L. Lavagno and C. Passerone, "Design of embedded systems," in *Embedded Systems Handbook*, R. Zurawski, Ed. CRC press, 2005, ch. 3.

[3] G. Heiser, "Virtualizing embedded systems - why bother?" *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, 2011.

[4] "Mesovirtualization: lightweight virtualization technique for embedded systems," *Software Technologies for Embedded and Ubiquitous ...*, 2007.

[5] 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.

[6] 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.

[7] A. Cohen and E. Rohou, "Processor virtualization and split compilation for heterogeneous multicore embedded systems," *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010.

[8] 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.

[9] G. Heiser, "The role of virtualization in embedded systems," ... *on Isolation and integration in embedded systems*, 2008.

[10] 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.

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

[12] M. Technologies, "Processor core family software user's manual," http://www.usrmodem.ru/files/adsl/mips.pdf, Accessed, June 2012, 2012.

[13] C. Bertin, C. Guillon, and K. De Bosschere, "Compilation and virtualization in the HiPEAC vision," *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010.

[14] A. A. Group, "Virtualization Extensions Architecture Specification," Web, 2011.

[15] "Embedded virtualization for the next generation of cluster-based MP-SoCs," *Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on*, 2011.

[16] Y. Park and S. Yoo, "Real-time operating system virtualization for xen-arm." Web, Available at http://os.korea.ac.kr/publication_papers/inter_confer/ParkMiri_ISET_2009.pdf. Accessed at 10 ago., 2010.

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

[18] W. Kanda, Y. Yumura, and e. al, "Spumone: Lightweight cpu virtualization layer for embedded systems," *Embedded and ...*, 2008.

[19] O. OVP, "Open virtual platforms," http://www.ovpworld.org/, Accessed, June 2012, 2012.