Full-Virtualization on MIPS-based MPSOCs embedded platforms with real-time support

Carlos Moratelli
Faculty of Informatics
PUCRS
Av. Ipiranga 6681
Porto Alegre, Brazil
carlos.moratelli@pucrs.br

Samir Zampiva
Faculty of Informatics
PUCRS
Av. Ipiranga 6681
Porto Alegre, Brazil
samir.zampiva@acad.pucrs.br

Fabiano Hessel
Faculty of Informatics
PUCRS
Av. Ipiranga 6681
Porto Alegre, Brazil
fabiano.hessel@pucrs.br

ABSTRACT

Virtualization has emerged also as a feasible technique for Embedded Systems, as it provides more secure platforms, improves software design quality and reduces costs. However, real-time and memory constraints requires the development of different techniques from that widely applied to enterprise computing. Still, embedded processors area and power consumption constraints has limited the large adoption of hardware support for full-virtualization. In this paper we present an embedded hypervisor designed to provide full-virtualization and real-time execution of applications. The hypervisor is running on a lightweight MIPS-based MPSOC platform improved to provide hardware-based virtualization. Discussions about key aspects of the multiprocessor hypervisor, such scheduling policies, real-time and overall overhead are presented.

Categories and Subject Descriptors

C.3 [**Special Purpose and Application-based Systems**]: Real-time and embedded systems

Keywords

Embedded Systems, Virtualization, Hypervisor.

1. INTRODUCTION

Virtualization has been widely adopted in enterprise computing providing the integration of multiple systems on a shared platform breaking the one-to-one correspondence between logical and physical systems [20]. Different software systems coexisting on the same hardware platform allows better software design quality, since, the systems can be designed separately, better processor usage, greater security levels while still reducing manufacturing costs. However, virtualization does not fit directly on the Embedded System (ES) field. These requirements are the same for ES design, and a question arises: May I use virtualization techniques in ES design? Of course the answer is yes, why not? Modern ESs are increasingly taking characteristics of general-purpose systems [8], however, they must support real-time applications which the current virtualization technology in enterprise field does not support [19].

In despite of the challenges to bring virtualization to ESs, some characteristics in its adoption are highly desirable. However, typical embedded constraints still prevent its wide adoption and much effort has been spent in order to demonstrate that its usage can indeed be feasible in ESs [1], [6], [16], [7], [5]. Heiser [9] highlights the need to run unmodified guest operating system (guestOS) and applications, besides providing strong spatial isolation to improve security. Armand [4] states that low overhead components are fundamental. Therefore, the biggest challenge lies in achieving all of these characteristics at once, while respecting typical ES's constraints.

Moreover, as these conflicts has a strong relationship with the hypervisor's [1] implementation, which in turn depends on the underlying hardware, the challenge itself grows. Thus, the architecture, and the characteristics of its Instruction-Set Architecture (ISA), can make the implementation either easier or harder [15]. Hardware-assisted virtualization is widely adopted in interprise solutions using technologies like Intel VT-x [11] aiming to keep the hypervisor simpler and more efficient. In embedded systems, although different architecture options are available, hardware support for virtualization is rarely adopted. The embedded ARM architecture specifies virtualization extensions [3], but, just a few manufacturers have implemented it in their cores.

Thus, this paper focus on a multiprocessor embedded hypervisor that provides full-virtualization and real-time execution of bare-metal applications. We are running the hypervisor on a MIPS-based MPSOC platform composed by modified MIPS4K cores [18] with proper hardware support to provide virtualization [2]. Our main contribution is to provide virtualization in embedded multiprocessor environments where: (i) no GuestOS modifications are desired, and; (ii) non-real-time GuestOSs need to coexist with real-time applications.

SBCCI '14, September 01 - 05 2014, Aracaju, Brazil

ACM 978-1-4503-3156-2/14/09 ...$15.00.
http://dx.doi.org/10.1145/2660540.2661012 .

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

2. RELATED WORK

Several studies discuss the use of virtualization in ESs [6], [16], [7], [5]. However, we consider that virtualization on ESs is an emergent technology that brings great benefits and will be continually spread in the near future. Thus, this section is focused on virtualized platforms with real-time support.

Enabling real-time support on virtualized platforms creates a 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 VCPU level, it is difficult to ensure real-time execution to the tasks over the RTOS. Aiming to deal with hierarchical scheduling problem Jin [19] suggests a mechanism based on messages which returns the scheduling information to the hypervisor level. The hypervisor support two kinds of VCPUs: the normal VCPU and the RT-VCPU. The hypervisor's scheduler uses the information obtained from the RTOS to map real-time tasks into RT-VCPUs. The RT-VCPUs are scheduled in a fashion to keep the real-time constraints. They implemented two ways to feed the hypervisor scheduler with real-time information: before task creation (at system design) and during task creation (with use of hypercalls). The main drawback of this approach is the requirement of a strong cooperation between the hypervisor and the RTOS, which means, both software layers must be implemented or modified in order to attend the message mechanism.

Kinebuchi et-al [12] proposed a lightweight virtualization layer called SPUMONE. Using a para-virtualization approach, the guest OSs are executed in privileged mode minimizing the virtualization overhead and the amount of modifications on the OS's kernel. On the other hand, such approach compromises the system's dependability, since, in privileged mode, a guest OS would interfere another guest OS or hypervisor. SPUMONE allows a General Purpose OS (GPOS) and a RTOS share the same physical CPU. In order to cope with real-time constraints, the RTOS is bounded to a VCPU with higher priority than the VCPU bounded to the GPOS. Interrupt virtualization is a key feature of SPUMONE's approach to deal with the hierarchical scheduling problem. When the RTOS is idle, the GPOS is executed. The RTOS restarts its execution when it receives an interrupt. An interrupt for RTOS preempts the GPOS immediately.

Lin [13] proposed a new architecture for SPUMONE, suitable for multi-core virtualization design. This new approach still execute the guest OS and the hypervisor in privileged mode. However, the authors use local memories such as scratchpad to provide isolation among VMs, by physically separating instances among physical cores. Their main intent is to avoid that a failure in an SMP Guest OS affects more than one VM. The main concern about this approach is that the isolation among VMs is only achieved when they do not share the same physical core.

To the best of our knowledge, all related works implement para-virtualization approaches (changes are required to be performed in the GuestOS). As we described, the main contribution of our work consists in offering a hypervisor full-virtualized with acceptable overhead even for real-time constraints. To do so, we use the concept of different types of VCPU, similar to [19], and a hypervisor capable of dealing real-time constraints through the EDF [10] scheduling algorithm. The two level hierarchical scheduling problem is avoided instantiating real-time applications which can be scheduled directly by the hypervisor.

3. VIRTUALIZATION MODEL

Our virtualization model is depicted in Figure 1. At the physical level, we assume a bus-based homogeneous MPSoC along with a shared memory. On top of the CPUs, we execute our distributed hypervisor, responsible for the creation and management of each Application Domain Unit (ADU). The ADU is a logic arrangement responsible to associate the guestOS with its VCPUs, besides, each ADU has its own memory space controlled by the hypervisor providing memory isolation. 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 ADU can use heterogeneous multiprocessors 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 GPOS with proper multiprocessor support[2] can be used for the best-effort tasks. Besides, it is responsible for instantiating an RT-VCPU for each real-time application it desires to execute.

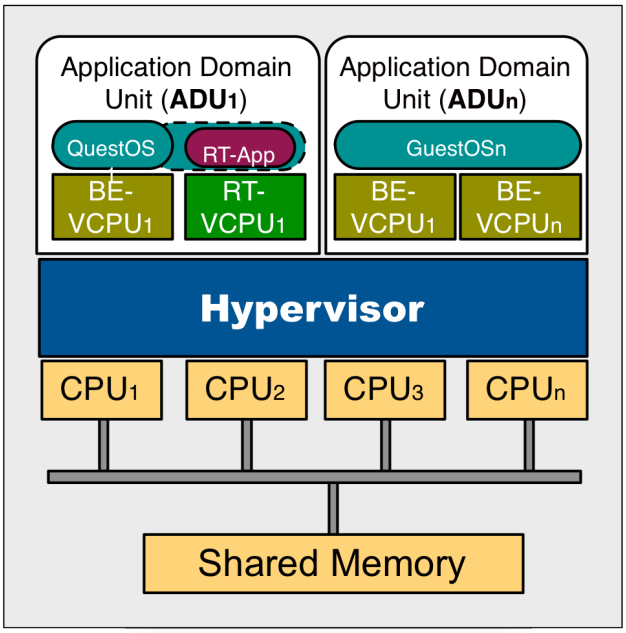

Figure 1: Virtualization model and Application Domain Unit for multiprocessor embedded systems

Figure 2 shows the possible flexible mapping and partitioning model for virtualized architectures based in our model. Each ADU has a given task-set associated with its VCPUs. 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 deal with a dynamic mapping of tasks among VCPUs (if supported by

[2]For a single GuestOS of a given ADU to manage multiple VCPUs, there is no model imposed restriction. However, this GuestOS must be implemented to support multiple processing units.

the GuestOS), VCPUs among CPUs, and even tasks among CPUs. Our approach to enable RT-VCPUs and meet timing constraints is further described in Section 4.

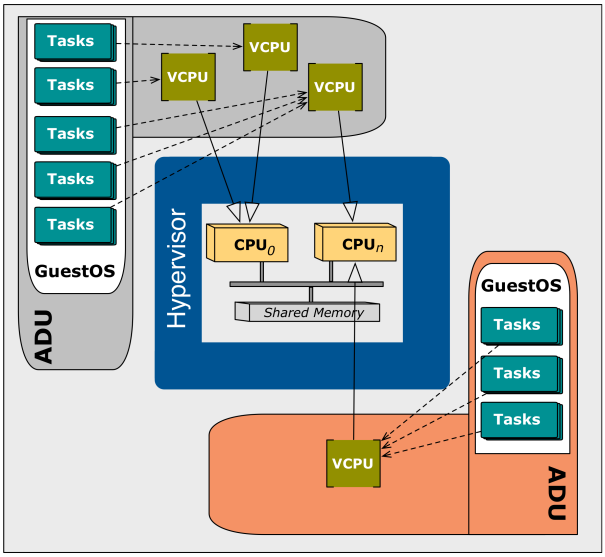

Figure 2: Flexible Mapping model for multiprocessor embedded systems with real-time support

4. MODEL IMPLEMENTATION

This Section discusses the implementation strategies for the model described in Section 3 in terms of hardware and software aspects of the platform.

4.1 Hardware aspects

Our platform is based on a typical MIPS 4Kc core modified to provide a lightweight virtualization support [2]. In this work we use a multiprocessor arrangement, where each processor has a local scratchpad memory and access to a shared memory, where the GuestOSs are located. Both memories are connected to the processors by a 32-bit wide bus. Still, a slave 32-bit wide bus is dedicated to peripherals, which are limited to a UART for communication purposes, and a Timer for time measurement. The UART is an IBM PC 16550 peripheral, whilst the Timer is a Xilinx IP XPS Timer Counter peripheral configured to the resolution of $1\mu s$.

Hardware support for virtualization. Since the MIPS 4Kc instruction set attend to the classic virtualization model proposed by Popek [15], a hypervisor can be constructed under it. However, the processor core contains fixed virtual memory segments, which are differently used depending on the mode of operation [3]. The part A for the Figure 3 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. During reset or when an exception is thrown the core enters into the *Kernel* mode where 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 ($0x00000000$ to $0x7FFFFFFF$) 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.

[3]The processor can operate in both Kernel or User modes, which gives different privilege levels.

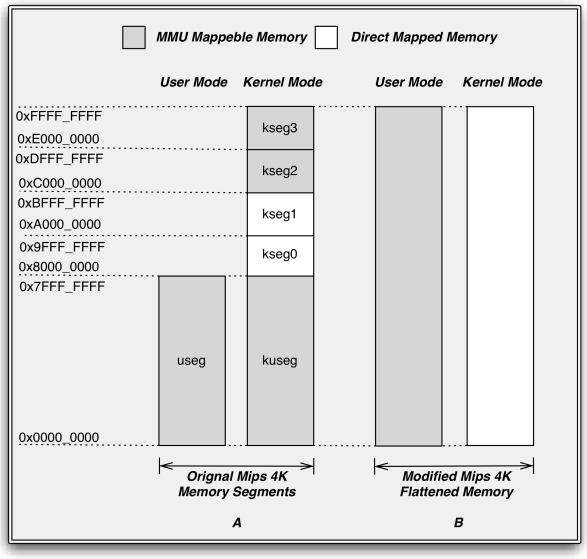

Figure 3: MIPS 4K memory management for User and Kernel modes of operation.

Such fixed memory map 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 the 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 interrupts 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 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 para-virtualization. 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 proposed 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 *Memory Management Unit* MMU when the Kernel mode is active.

Part B of the Figure 3 shows the resulting flattened memory. Now, the hypervisor has total control of the memory

when using the MMU. However, once the MMU has been turned on there is no way to turn it off again. This is necessary to return to the hypervisor code after a VM execution. Thus, we modified the MIPS 4Kc core in order to turn off the MMU when the Kernel mode is active. More details about the hardware support for virtualization on the MIPS 4kc can be found in [2].

4.2 Software aspects

Figure 4 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 optimal performance; (ii) *Memory-Mapped I/O (MMIO)*, which manages the memory-mapped devices, including the hypercalls' subsystem; (iii) *ADU Control Layer* dealing with the ADUs; (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.

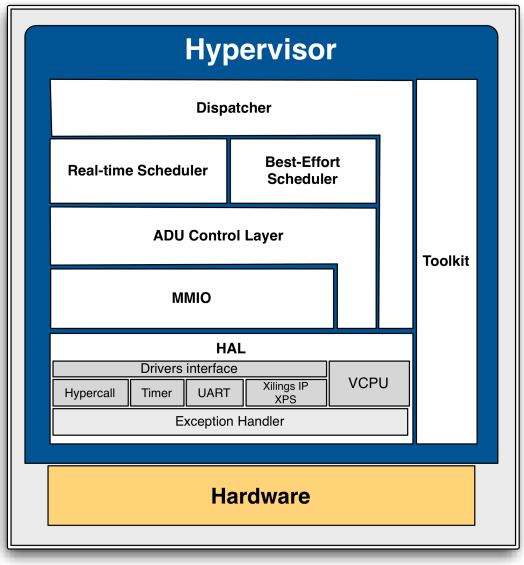

Figure 4: Hypervisor block diagram

Hypercalls. The hypervisor implements the hypercall concept to allow an ADU to instantiate RT-VCPUs. Hypercalls are widely used in para-virtualization based approaches, where the GuestOS needs to be modified to invoke hypercalls instead of use privileged instructions. In our case (full-virtualization) we use the hypercalls concept exclusively to perform proper 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, if the system designers wish to enable real-time support for a certain guestOS, they may choose to implement our hypercall subsystem, giving to the guestOS the ability to start real-time tasks in a bare-metal setup. We choose do not support dynamic instantiation of BE-VCPU, since, its behaviour is analogue to a task instantiated by a multi-processed guestOS in the sense both are best-effort.

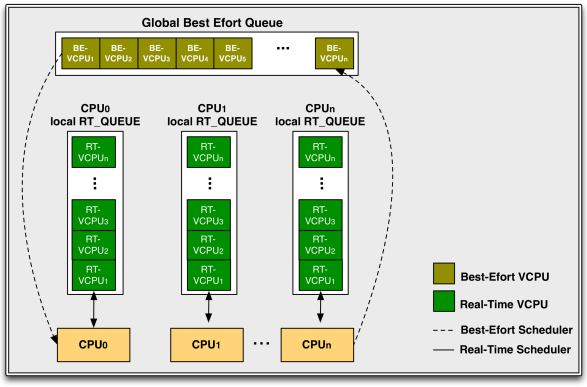

Figure 5: Real-time and best-effort multiprocessor scheduler.

Scheduler. The hypervisor implements both Earliest Deadline First (EDF) [10] and best-effort (round-robin) policies to deal with RT-VCPUs and BE-VCPUs, respectively. For the BE-VCPUs implementation, a single global best-effort queue is managed, while RT-VCPUs are kept in local individual queues per processor. The EDF is the main hypervisor's scheduler and it has higher execution priority over the best-effort scheduler, which will not suffer from starvation since we use time reservation for it. Figure 5 presents this two-level scheduling scheme, where RT-VCPUs and BE-VCPUs are placed in different positions (global and local individual queues). The EDF scheduler requires deadline, period, and capacity real-time parameters which are received from the guestOS during the hypercall invocation for a RT-VCPU instantiation.

Scheduling strategy. This sample will illustrate our scheduling strategy where both EDF and best-effort schedulers cooperate on a multiprocessor platform aiming to increase the CPUs utilization. Assuming two physical CPUs for the execution of 2 BE-VCPUs (kept in the global BE queue) and 4 RT-VCPUs (*0* and *1* assigned to CPU *0* and *2* and *3* assigned to CPU *1*), the following real-time parameters[4]: (period p and capacity c): (i) RT-VCPU *0*: p = 5 and c = 3; (ii) RT-VCPU *1*: p = 4 and c = 1; (iii) RT-VCPU *2*: p = 5 and c = 2; (iv) RT-VCPU *3*: p = 4 and c = 1; and the following best-effort parameters: (v) BE-VCPU *0*: priority 1, and; (vi) BE-VCPU *1*: priority 4. Figure 6 depicts this scheduling scenario during *20* time slices (*ticks*). Each RT-VCPU is scheduled according to the EDF policy on the physical CPU where it was previously designated. However, the BE-VCPUs can migrate among CPUs aiming to increase the utilization of the idle CPUs. The circle 1 in the figure indicates the BE-VCPU 0 executing on CPU 1, since, the RT-VCPU 2 and 3 are waiting for their release time. For the circle 2 we have the same scenario, but, the circle 3 shows the BE-VCPU 0 executing on CPU 0. Finally, in the circle 4 and 5 the BE-VCPU 0 turns to CPU 1. Similarly, the BE-VCPU 1 migrate among the physical CPUs.

[4]For sake of simplicity we are assuming period and deadline values to be the same.

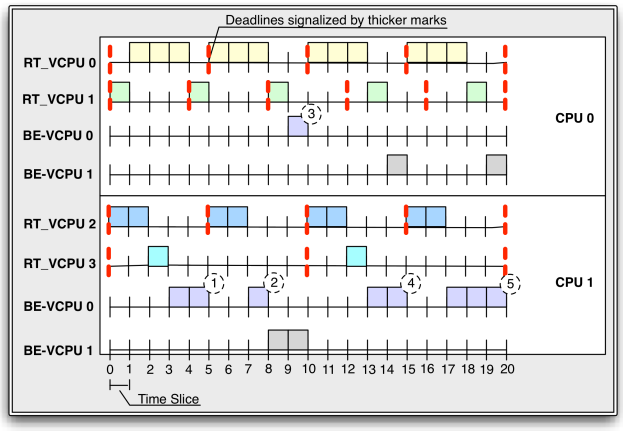

Figure 6: Real-time and best-effort multiprocessor scheduling example.

Time Reservation. To avoid starvation of BE-VCPUs, we adopted a time reservation strategy where the designers must indicate the system load capacity for real-time purposes desired for each ADU. For example, on a system with 1 CPU and 2 ADUs, the designers may attribute upto 30% [5] of the CPU time for each ADU to use for real-time purposes, resulting in 40% of CPU time available to BE-VCPUs. From the ADU point of view, this share of the system's entire capacity is seen as its own entire real-time capacity. All the RT-VCPUs created by this ADU share its slice of the entire system's capacity. If an ADU tries to allocate more than its maximum real-time capacity, new RT-VCPUs will fail to be created as they will not succeed through the admission control algorithm. In this case, the guestOS will receive an error message from the hypercall subsystem.

Admission Control. Whenever an ADU requests the creation of a new RT-VCPU, the admission control algorithm checks if there are enough available physical resources that can satisfy the RT-VCPU requirements. In the particular case of multiprocessor systems, even if there is enough free capacity in the entire system, this may not guarantee the new RT-VCPUs allocation since this capacity is actually fragmented among several physical CPUs. Thus, there is indeed an efficiency issue regarding the local CPU queues in the real-time scheduler. However, our model utilizes a shared memory where is possible to move RT-VCPUs among CPUs aiming to reduce this problem.

Inter-domain communication. Direct communication among ADUs is not possible since our model imposes memory isolation between them. However, our proposal is flexible enough to support typical sockets style [17] communication among ADUs. Thus, the hypervisor may implement a network layer in order to route Internet Protocol (IP) datagrams [17] between ADUs. Still, when communication with the external world is desired the hypervisor may virtualize a network adapter providing network capabilities to the platform. Nevertheless, in the embedded system context, the sockets communication scheme can be expensive in terms of system overhead. Aiming to provide a lightweight communication subsystem among ADUs we implemented a message passing scheme where the hypervisor is responsible for performing a copy from the sender ADU's memory area into the receiver's ADU's area. Whenever a guestOS desires take advantages of the proposed architecture it must implement a proper driver that conforms to our communication protocol.

[5]The designers must determine the amount of CPU time needed to attend the ADU's real-time constraints, in this example, 30% for each ADU.

5. RESULTS AND DISCUSSION

In order to validate our platform we used OVP [14], which is a hardware simulator for MPSoCs, instruction-accurate and able to simulate an entire platform. OVP offers a large open-source model database of IP cores, supporting several processor families (such as 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 knowledge, the scheduler jitter and the standard deviation are the most important time aspect to measure the hypervisor overhead. The scheduler jitter is the undesired deviation time from the exact moment that a VCPU should execute, caused by the new software layers imposed by the hypervisor, resulting in an execution delay. This delay affects the total execution time available to the VCPU, consequently, impacting in the system performance. For real-time purposes is important to keep the jitter as small as possible. Still, for a good predictability of the system behaviour the jitter must be constant as well.

Our experiment consists in measurement of the jitter for RT-VCPUs scheduling in 3 different system configurations composed by 1, 2 and 4 physical CPUs. For each configuration, we varied the number of RT-VCPUs in execution, for 1, 2 and 4. For all tests, a ADU, executing onto a BE-VCPU, is responsible to start the RT-VCPUs remaining idle (without affecting the behaviour of other VCPUs). The OS running in the ADU is a simple OS, called BareOS, which was designed by us in order to validate the hypervisor. The BareOs is able to initialize the CPU (VCPU in case it is virtualized) and implements several facilities to the applications, in special, communication subsystem, device drivers and the hypercall for instantiation of RT-VCPUs. Each instantiated RT-VCPU execute the same application, called RT-App, which is responsible to measure the jitter. It is important highlight that in our virtualization model, a RT-app does not suffer influences from the best-effort OS (in this case the BareOS), since, it is running onto a RT-VCPU wich will be scheduled directly by the hypervisor's scheduler. OVP was configured to simulate the rate of 100 MIPS (Millions of Instructions Per Second). Since the 4Kc core implements a load/store architecture with single-cycle ALU operations, it is possible to approximate the core frequency to 100MHz.

The jitter was obtained calculating the time difference from the exact moment the RT-App should start to execute to the moment it effectively started. When scheduled, the RT-App reads the current time from the hardware timer determining the jitter and outputing the result through the serial port for analysis. The average jitter was calculated in microseconds from 1000 scheduling rounds.

The Tables 1, 2 and 3 show the individual average jitter (A.J.) and the standard deviation (S.D.) for each one of the RT-VCPUs for the system configuration of 1, 2 and 4 CPUs, respectively. For the Table 1, we can identify that the jitter slightly increased associated with the the number of RT-

Table 1: Hypervisor versus Number of RT-VCPUs for 1 CPU.

#RT-VCPUs	RT-VCPU ID	A.J. (μs)	S.D. (μs)
1	1	163.99	1.00
2	1	165.01	1.05
	2	164.98	0.57
4	1	166.94	0.57
	2	166.95	1.93
	3	166.97	0.98
	4	166.98	0.60

VCPUs. With 1 RT-VCPU the average jitter is 163.99μs and with 4 the smallest average jitter is 166.94μs to the RT-VCPU 1. This small increase associated to the number of RT-VCPUs is expected because more RT-VPCUs add more complexity to the EDF scheduler. However, all RT-VCPUs executing on the same CPU are likewise affected, since, for 4 RT-VCPUs the average jitter is about 166.9μs. Considering that our scheduler *quantum* is 10ms a jitter of 166.9μs represents a overhead of 1.67% introduced by the hypervisor. Still, the standard deviation help us to understand how predictable our real-time scheduler is. The worst stantard deviation was 1.93 μs to the RT-VCPU 2 in the 4 RT-VCPUs scenary. Such small standard deviation shows that our system is very predictable for 1 CPU even when the number of RT-VCPUs increase.

Table 2: Hypervisor versus Number of RT-VCPUs for 2 CPUs.

#RT-VCPUs	RT-VCPU ID	A.J. (μs)	S.D. (μs)
1	1	316.51	19.47
2	1	378.85	30.53
	2	378.60	24.66
4	1	323.92	27.00
	2	323.97	30.90
	3	324.02	33.25
	4	323.88	24.99

Results for 2 CPUs presented on Table 2 shows that the average jitter increase. This happens due to data contention in the hypervisor shared structures. Although the hypervisor uses a local memory to keep data for each processor, there are common subsystems that are shared among all CPUs. However, similar to the results in the Table 1, the RT-VCPUs are likewise affected, for example, whith 4 RT-VCPUs the smallest average jitter is 323.92μs to the RT-VCPU 1 and the worst is 324.02μs to the RT-VCPU 3. The worst average jitter of 378.85μs represents a hypervisor overhead of 3.78%, which we consider very optimistic. Still, the addition of a second CPU increased the standard deviation substantially being 33.25μs in the worst case suffered by the RT-VCPU 3 on the 4 RT-VCPUs scenary. A decrease in the system predictability is the onus to obtain more processing power.

Finally, the results for 4 CPUs showed in Table 3 presents

Table 3: Hypervisor versus Number of RT-VCPUs for 4 CPUs.

#RT-VCPUs	RT-VCPU ID	A.J. (μs)	S.D. (μs)
1	1	314.51	42.41
2	1	299.54	28.27
	2	299.72	28.33
4	1	297.57	6.03
	2	297.58	4.36
	3	297.94	5.99
	4	297.78	6.24

an average jitter close to the Table 2 for 2 CPUs system, but, lightly smaller. In this case the worst average Jitter is 314.51μs to the 1 RT-VCPU configuration and the smaller is 297.57μs to the RT-VCPU 1 in the 4 RT-VCPUs scenary. The standard deviation in almost all cases is close or smaller then in the 2 CPUs system, except in the case of 1 RT-VCPU which achieved 42.41μs.

From the overall results we can conclude two main points about our hypervisor: i) it is highly predictable for monoprocessor platforms and presents a low overhead. ii) it is scalable in multiprocessor systems, although, the system will suffer a penalty in terms of overhead and predictability. However, we believe that our hypervisor's scheduler can be optimized for multiprocessor systems in order to decrease the overhead and improve the predictability.

6. FINAL REMARKS AND FUTURE WORK

In this paper we presented a virtualization model intended for multiprocessor embedded systems with real-time constraints where no change in the GuestOS is required and hardware support for virtualization is possible. The main advantages of our approach are: (i) the absence of required GuestOS's changes; (ii) the strong secure domain offered when hiding the hypervisor's memory from virtual machines and the virtual domains' memories among themselves; and (iii) real-time support through a typical real-time scheduling policy (EDF). Results were taken aiming to measure the hypervisor's real-time responsiveness and predictability as well the overhead introduced by the hypervisor itself. Future works include the study of techniques to decrease the average jitter and the standard deviation in the hypervisor's scheduler for multiprocessor systems.

7. REFERENCES

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

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

[3] ARM. Virtualization extensions - arm. http://www.arm.com/products/processors/technologies/virtualization-extensions.php, Accessed, August 2013, 2013.

[4] 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.

[5] 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.

[6] 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.

[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] G. Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM.

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

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

[11] Intel. Hardware-assisted virtualization technology. http://www.intel.com/content/www/us/en/ virtualization/ virtualization-technology/hardware-assist-virtualization-technology.html, Accessed, August 2013, 2013.

[12] Y. Kinebuchi, W. Kanda, Y. Yumura, K. Makijima, and T. Nakajima. A hardware abstraction layer for integrating real-time and general-purpose with minimal kernel modification. In *Future Dependable Distributed Systems, 2009 Software Technologies for*, pages 112–116, 2009.

[13] T.-H. Lin, Y. Kinebuchi, H. Shimada, H. Mitake, C.-Y. Lee, and T. Nakajima. Hardware-Assisted Reliability Enhancement for Embedded Multi-core Virtualization Design. *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, 2011.

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

[15] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[16] 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.

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

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

[19] Y. Wang, J. Zhang, L. Shang, X. Long, and H. Jin. Research of real-time task in xen virtualization environment. In *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*, volume 1, pages 496–500, 2010.

[20] S. Xi, J. Wilson, C. Lu, and C. Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48, 2011.