Adding Virtualization Support in MIPS 4Kc-based MPSoCs

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

***Abstract*—Virtualization has emerged as a feasible technique for Embedded Systems, providing safer platforms, improving design quality and reducing manufacturing costs. However, its inherit overhead still prevent its wide adoption. Most of the current attempts use the para-virtualization technique that imposes the cost of performing comprehensive changes in the guest OS. We propose the adoption of full-virtualization for MPSoCs, where no guest OS changes are required and, in order to reduce known virtualization overheads, we propose some hardware modifications to a MIPS-based architecture. We conducted experiments that demonstrate our proposal by comparing its processing and communication overheads against a non-virtualized solution.**

I. Introduction

Embedded Systems (ES) count each more on powerful platforms to provide an increasing number of functionalities required by consumers. However, common ESs constraints, such as silicon area, memory size, and design complexity still remain. In this context, virtualization has arisen as a possible solution for embedded applications, since it can **(i)** reduce manufacturing costs; **(ii)** offer greater security levels; **(iii)** enable more efficient processor usage, and; **(iv)** provide better software design quality, since legacy systems can be reused as a virtual machine that coexists with newer projects [1].

Much effort has been spent in order to demonstrate that virtualization is feasible for embedded systems [2], [3], [4], [5], [6]. Most of these focus on providing para-virtualization, aiming to decrease implicit virtualization overheads at the cost of requiring the GuestOS to be changed. Yet, Heiser [7], [8] highlights the need to run unmodified guest OS and applications, besides providing strong spatial isolation to improve security. Armand [9] states that low overhead components are fundamental.

From this conflict of interest between the need to run unmodified guest OSes and the need for low execution overhead, raises the possibility of using *hardware assisted virtualization*. This approach provides several advantages, such as lower overhead, a simpler hypervisor implementation (allowing it to be safer) and more design flexibility, since there is no need to change the guest OS's source code.

Thus, this paper investigates the combination of hardware-assisted virtualization and full-virtualization techniques in a multiprocessed embedded platform. We adopted MIPS-based processors as MIPS is a widely adopted architecture being present in video-games, e-readers, routers, DVD recorders, set-top boxes, among others. Our main contribution is to provide a platform that benefits from known virtualization characteristics, such as increased security and reduced area occupation. Experiments demonstrate the overhead of the proposed virtualization platform, since we use a trap-and-emulate technique for both CPU and I/O bounded applications, compared to a non-virtualized solution. We also evaluate the impact on the overall system performance caused by the amount of virtual and physical CPUs.

The remainder of the paper is organized as follows. Section II discusses some related work. Section III presents the virtualization model and Section IV its implementation concerns. Section V presents the results while Section VI concludes the paper with final remarks and future work.

II. Related Work

A. Academic Hypervisors

Main [10] highlights the possibilities of creating guest-to-guest protocols, aimed for inter-OS communication and signalling through shared-memory and hypervisor's watch. Lin and colleagues [11] propose the SPUMONE architecture, suitable for multi-core virtualization design. In their work, the authors use local memories, such as scratchpads, in order to provide safer domains by physically separating their placement. Their main goal is to prevent a failure in an SMP guest OS from affecting more than one domain. Nakajima et al. [12] extend the SPUMONE architecture to provide temporal and spacial isolation in virtualization of information appliances. These authors use the para-virtualization technique. They achieve temporal isolation with virtual core migration aiming to decrease the dispatch latency of a guest OS application. Finally, [13] introduces real-time resource management into the SPUMONE architecture using a fixed priority preemptive scheduling.

B. Hardware Support

The addition of CPU hardware assists for system virtualization has been key to the practical application of hypervisors in enterprise computing. Intel VT was first released in 2005 and it has been a key factor in the growing adoption of full-virtualization in the enterprise-computing world. Recently, this trend could also be observed in embedded systems [14]. Power.org released in 2009 a specification for the addition of virtualization into the version 2.06 of its ISA. Still, in 2010, ARM also announced the addition of hardware virtualization extensions to its architecture. Finally, MIPS has also announced extensions to its ISA that allow hardware virtualization support in the end of 2012 although at the present time no commercial processor is using such facility.

C. *Analysis*

With the growing hardware support for virtualization in embedded architectures, the use of full-virtualization must be better investigated. Currently, related works focus mainly in para-virtualization approaches (changes in the guest OS are required). However, full-virtualization is desired in embedded devices as long as it has proper hardware support [15]. Thus, the main contribution of our work consists in investigating the behavior of a hardware-assisted fully-virtualized system aiming MIPS-based MPSoCs that require a strong memory separation among virtual machines. In this context, our mechanism allows a secure and transparent environment for guest OSs to cooperate in a multiprocessed environment.

III. Virtualization model

Our virtualization model assumes a bus-based homogeneous MPSoC with a shared memory, so all physical processors share a common address space. Above the CPUs we run the hypervisor, responsible for the creation and management of each virtual machine that we call Application Domain Unit (ADU). Into an ADU, applications can be mapped onto Virtual CPUs (VCPUs) according to their needs. Still, each ADU can count on multiple VCPUs allowing a guest OS with proper multiprocessor support to work[1].

Figure 1 shows the possible flexible mapping and partitioning model we propose. Each ADU contains a guest OS with applications represented as a task-set to be associated with the VCPUs required by this ADU. From a VCPU point of view, a single subset of the entire ADU's task-set is available and is scheduled and managed by the guest OS. It is important to highlight that a single ADU may contain more than one VCPU. However, whenever this occurs, the guest OS becomes responsible for managing these multiple virtual cores.

From the overall system point of view, many VCPUs per ADU can be disposed as if in a matrix arrangement. Each matrix element is independently mapped onto the CPUs. Since we are providing a bus-based virtualization system, CPUs can be seen as an array of available physical processors. Thus, the separation provided by our virtualization model can ease the dynamic mapping of tasks among VCPUs (if supported by the guest OS) and VCPUs among CPUs.

IV. Implementation Description

For the implementation of our proposal we use the OVP [16] platform, which is an instruction-accurate simulator written in C language able to simulate an entire platform, with a given processor core (or many of them) and some peripherals. OVP offers a large open-source database with platform models that support several processor families (such as MIPS, ARM and PowerPC) and many peripherals. The implementation presented in this section is based in one of these models with modifications to the core description in order to support the proposed platform. Hence, we describe some of the original core's characteristics and the main modifications we performed to provide virtualization.

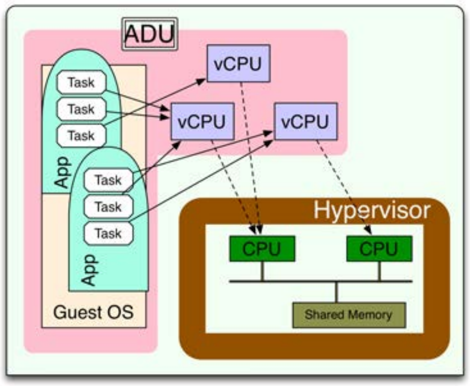

Fig. 1. Proposed virtualized mapping model

A. *Hardware aspects*

Our platform is based on a typical MIPS 4Kc core [17], which we modified in order to provide virtualization facilities. Hereafter, we refer to this modified version of MIPS 4Kc as MIPS 4Kc-VT. We use this core in a multiprocessed arrangement, as each processor has a local scratchpad memory and accesses a shared memory, where the guest OSes are located. Both memories are connected to the processors by a 32-bit wide bus. Still, a slave 32-bit wide bus is dedicated for peripherals, which, for now, are limited to a UART for communication and debug purposes, and a timer for cycle accurate measurements.

Memory Management. Originally, the MMU of the MIPS4Kc processor core is conceived to perform virtual to physical translation for any address before sending requests either to the cache controllers, for tag comparison purposes, or to the bus interface unit, due to an external memory reference. 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). When an address needs to be translated, the appropriate micro TLB (ITLB or DTLB) is accessed first. In cases where the translation is not found in the micro TLB, an access to the JTLB is then performed. Therefore, if a miss occurs in the JTLB, an exception is risen.

The address translation performed by the MMU depends on the processor's execution mode[2]. Part *A* of Figure 2 shows the memory segments that can be accessed on each active execution mode [17]. For instance, kernel mode has several

[1]For a single guest OS of a given Application Domain to manage multiple VCPUs at the same time, there is no model imposed restriction. However, this guest OS must be implemented to support SMP architectures.

[2]Supported modes are: **User mode**, mostly used for application programs; **Kernel mode**, handling exceptions and privileged operating system functions; **Debug mode**, used for software debugging.

segments available (from *kseg0* to *kseg3*, including the *kuseg*), while user mode only accesses *useg* (with virtual addresses equivalent to the *kuseg* segment).

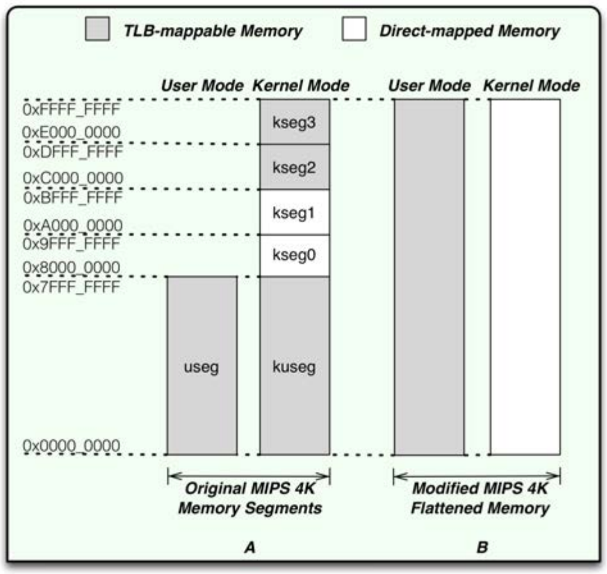

Fig. 2. MIPS 4K memory management for User and Kernel modes of operation

Virtual Memory Segments. Initially, the core enters into the kernel mode during reset, and whenever an exception is raised. In a non-virtualized platform, the virtual memory segments scheme helps the OS to keep user applications isolated from the kernel by running them in different segments. Still, the OS can have privileged access in certain memory areas. Although the segments scheme is strongly recommended to non-virtualized systems, it brings undesirable restrictions to a virtualized platform. In the MIPS4K core only the first 2GB of the virtual memory are available to the virtual machines. Thus, a guest OS running in User mode would not be able to address virtual memory above 2GB. Besides, the fixed-mapping of *kuseg0* and *kuseg1* segments would become a problem for virtualization purposes, since the hypervisor needs to register its exception routine under the Exception Vector address (at 0x8000_0000) and then take control of the execution of guest OSes' privileged instructions. It has also to take control over the hardware interrupts, TLB misses, and other system exceptions.

Yet, in the original implementation of MIPS 4K, a guest OS tries to register its own exception handler routine, causing a conflict with the hypervisor's implementation (and possibly with other guest OSes'). 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 OSes' needs. The same scenario description can be applied to the *kseg1* segment, whenever the hypervisor tries to virtualize a given device. Therefore, to support full-virtualization on a MIPS 4Kc core, we propose two main modifications: (i) removing all virtual memory segments, specially the fixed-address segments (*kseg0* and *kseg1*), and; (ii) disabling the TLB-Translation when kernel mode is active.

The removal of all virtual memory segments implies that virtual memory addresses are only mapped to physical memory when the TLB has a valid entry. However, after enabling TLB translations and executing the TLB flush routine, there is no way to disable the TLB. Thus, a valid entry needs to be kept in the TLB so the hypervisor can map that area into the physical memory. Such scheme is not transparent for a guest OS trying to configure its own TLB entries. In this context, to avoid further conflicts, we have modified the MIPS 4Kc core so the TLB is disabled whenever kernel mode becomes active. In this condition, the modified core MIPS 4Kc-VT translates each single virtual address directly to the matching physical address, 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 needed whenever the guest OS tries to access either the Exception Vector or a memory-mapped device. The new memory map for both user and kernel modes are depicted in Part *B* of Figure 2.

B. Software aspects

Figure 3 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 specific hardware details. It merges drivers interface, along with device drivers implementation, besides handling the VCPUs abstraction; (ii) *Memory-Mapped I/O (MMIO)*, which manages the memory-mapped devices; (iii) *Application Domain Unit (ADU)* described in Section III; (iv) *Scheduler*, responsible for scheduling VCPUs into the CPUs in a round-robin fashion, and; (v) *Dispatcher*, responsible for dispatching the chosen VCPU to the physical CPU.

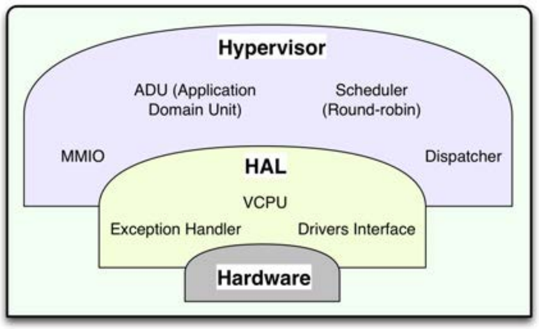

Fig. 3. Hypervisor block diagram

Exception Vector. Originally, MIPS 4K contains a fixed location designated for the Exception Vector starting at 0x8000_0000 (except when the processor is in debug mode). This causes an address conflict between the hypervisor and the guest OSes because both will try to register their exception routine at the same address. To cope with that, we created a virtual mapping for the guest OSes, where the physical address 0x8000_0000 is mapped to a virtual address. So, the hypervisor can register its exception routine at 0x8000_0000 while the guest OSes use a virtual address, which is handled by the hypervisor.

Exception Return. Guest OSes run in user mode and 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 instructions and emulate them. 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 its return address 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 an ERET instruction is executed.

Memory-mapped peripherals. *Direct mapping*: the hypervisor maps the peripheral directly to a guest OS and any requests from other guest OSes are simply denied. In such way, no overhead is added when a 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 unwished guest OS accesses do trap to the hypervisor, generating an exception to be treated accordingly. *Shared peripheral*: 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 OSes. 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. A guest OS that accesses this area causes a trap to the hypervisor that identifies where the origin of the request is, and emulates the peripheral. This means that the hypervisor needs to implement a device driver specifically for each shared peripheral.

C. Multiprocessor concerns

Synchronization Primitive. MIPS II provides two instructions for synchronization purposes: *Load Linked* (LL) and *Store Conditional* (SC). However, in the MIPS 4K core, these instructions are originally intended for single processor architectures. Therefore, to provide synchronization among the many processors intended by our architecture, we developed a lock-based system using a memory-mapped peripheral that guarantees atomicity for the following software layers of the system.

Inter-domain communication. Our proposal is flexible enough to work on monoprocessed and multiprocessed architectures. Either way, we need to provide a communication mechanism between Application Domain Units. The hypervisor is responsible for this support basically performing a copy from the sender domain's memory area into the receiver's domain memory area. For this to work, the guest OS must have a proper driver that understands our communication protocol, implemented to take advantages of the proposed architecture, reducing possible overheads.

D. Guest OS concerns

Our virtualization approach is based especially in some memory-mapping modification, what caused a compatibility break between the original MIPS 4Kc and the modified core (MIPS 4Kc-VT), meaning that a guest OS intended for the platform must be firstly ported to this modified core. Although it can be considered as a disadvantage of our approach, it is important to highlight that once the guest OS is running on this modified core (without hypervisor interference), virtualize it is straightforward due to our full-virtualization approach. Since the modifications for the MIPS 4Kc-VT follow the MIPS R3000 Application Binary Interface (ABI) specifications, the OS modifications also follow this specification.

Case-study: porting HellfireOS to MIPS 4Kc-VT. As a proof-of-concept, we used an academic OS that was not originally compatible with MIPS R3000 specification to illustrate the port to MIPS 4Kc-VT. HellfireOS (HFOS) [18] is a highly customizable operating system, suitable to run on low memory constrained architectures, like some embedded systems. It was primarily designed to run on a Plasma MIPS core. Basically, some modifications were required on the exception handler routine, as we use the ERET instruction to return from exceptions in MIPS 4Kc-VT. HFOS was originally designed to run on MMU-less processors and, since the MMU is managed exclusively by the hypervisor, there's no need for HFOS to be aware of the MMU. On the HFOS implementation level, modifications concerned exclusively its Hardware Abstraction Layer (HAL). Finally, these modifications did not impact significantly neither HFOS's code size nor its data memory usage.

V. Results and discussion

A. Evaluation methodology

To perform our evaluation we've implemented a peripheral responsible only for measuring time in microseconds. This peripheral is placed in the shared bus and each virtual machine accesses it through emulation, with a 600-instruction overhead. This induces some extra timing to each access that is not significant when compared to the algorithm's results. We use HellfireOS as a guest OS, described in Section IV-D. Other configuration details are described at each case study, as needed.

B. Processing Overhead Measurement

This test demonstrates the processing overhead of our proposal by comparing it to a non-virtualized solution. We have implemented a CPU-bounded application that implements the classic Hanoi Tower problem [19] resolution. Results were measured from the average execution time of one hundred iterations using a 16-piece configuration. Figure 4 shows the virtualization overhead compared to the non-virtualized platform. We varied the amount of physical cores (CPUs), even in the native execution. Then, for the virtualized platform, besides varying the amount of CPUs we varied the amount virtual cores (VCPUs) per physical core.

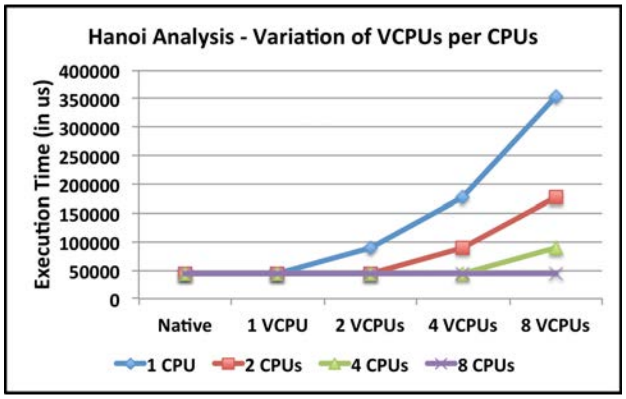

Fig. 4. Virtualization execution overhead for the Hanoi algorithm

Best-case scenario can be considered when we have one VCPU per CPU. In this case, the average overhead is not significant (around 0.32%). However, in the worst case comparison, when we have a single CPU and 8 VCPUs, the overhead grows dramatically to more than 700% since we need to share a single physical resource between a bigger amount of VCPUs. Still, there are cases when the total amount of VCPUs is inferior to the amount of CPUs. In these cases the average overhead is around 0.33%, meaning that there is no significant interference of the scheduling scheme in the overall system performance. The compromise of the VCPU per CPU ratio must be carefully analysed by the designer in order to balance the benefits and the cost induced by the platform.

We also analysed the execution and emulation of shared peripherals. To stimulate such problem, we analysed the relationship between the virtualization overhead growth and the amount of UART accesses (per 100000 instructions), depicted in Figure 5. It is possible to see that the more UART accesses a program performs, the more its execution overhead grows. However, we believe that depending on the application's behavior our virtualization overhead can still be acceptable. It is also important to highlight that, if a given virtual machine uses extensively a given peripheral, it could be considered to use the direct-mapping strategy, which eliminates the emulation overhead.

C. Communication Overhead Measurement

Figure 6 shows the results of two communication-bounded application. Firstly, the Ping Test application is responsible only for performing message exchanges. Secondly, the Bitcount application contains a master processor that is responsible for counting the amount of set bits in a given array of

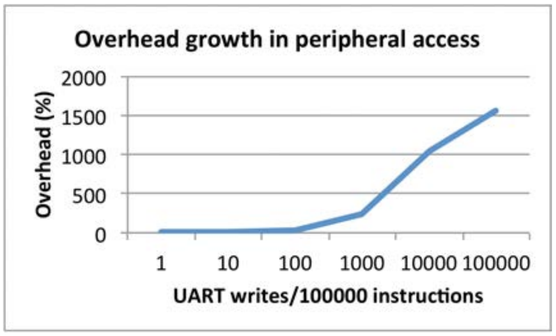

Fig. 5. Virtualization overhead at UART accesses

bits using a given number of slaves to help complete the task. Both applications are executed in virtualized and non-virtualized (native) scenarios. Since these are communicating applications, native execution requires at least two physical cores, varying from 2, 4 and 8 CPUs. In the virtualized environment, we've performed an analysis based in the VCPU per CPU ratio (VCPU:CPU), varying from 1:1, 2:1, 3:1 and 4:1. Still, for the ping test we varied the size of each message, since the larger the message size, the more packets are needed to send it through the network. Results were taken as an average of execution time needed to send one thousand messages for each test set. For the Bitcount test we use an array size of 4096 and each VCPU as a slave and one thousand executions for the average results. Since this test involves a

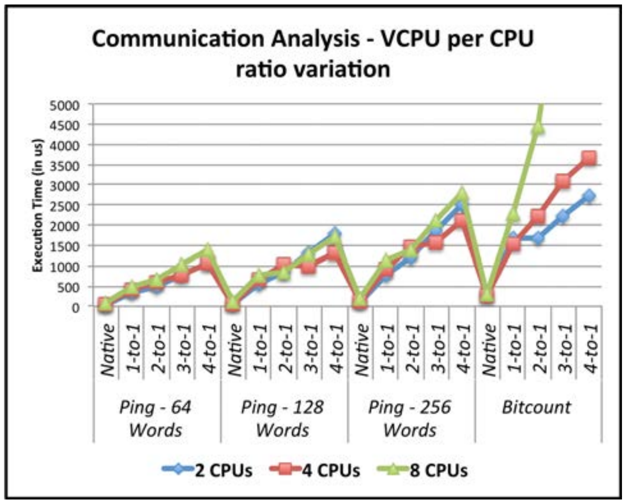

Fig. 6. Communication Test with Synthetic Application (Ping Test)

lot of privileged instructions to perform the communication, we observed that the overhead is equally high. Even in the best cases, when the VCPU per CPU ratio was set in 1:1, the communication overhead was in average 800%, due to the trap-and-emulate strategy we use. This occurs mainly due to our trap-and-emulate strategy. We believe that some refinement in the way full virtualization is provided, such as adding different execution modes (as similar to what Intel, AMD, and ARM have done) would definitively decrease the overall execution overhead.

D. Mixed Scenario Measurement

Finally, the third scenario mixes the processing and communication bounded applications to allow an analysis of the interference they cause in each other. We use the Hanoi as the CPU-bounded application and the Bitcount communicating algorithm. Figure 7 depicts the execution times in two cases: in the graph on the right side of the figure, we present the Bitcount execution times. On the left side of the figure, we present the Hanoi execution times. For both tests, each VCPU contains one test (1 Hanoi or 1 Bitcount) as we varied the amount of Hanois and Bitcounts per test, also varying the amount of CPUs. Native scenario only explores one CPU per Hanoi or Bitcount employed.

It is possible to see that the Bitcount execution suffers more interference from the amount of CPUs employed, with a single CPU being the worst case scenario and eight CPUs being the best-case scenario. The Hanoi execution time is also clearly affected by the increase of Bitcounts in the system, although the processing time itself is affected by the amount of Hanois in the system (more VCPUs per CPU ratio) and by the time spent by the hypervisor treating the privileged communication.

E. Discussion

These results were taken aiming to investigate the behavior of the proposed platform. We use a hardware-assisted virtualization based in the trap-and-emulate technique. Therefore, we were able to achieve low execution overheads depending on the VCPU per CPU ratio. Obviously, if we are using a single CPU to execute two vCPUs, we'd expect the execution time of the application running on the vCPU to double, at least. However, the virtualized approach would need half the area occupied by the non-virtualized approach, and that's a tradeoff that needs to be analysed. Also, if we are talking about applications that could share a physical CPU (non-prohibitive overhead) but could not coexist due to security reasons, the strong separation between Application Domains could be a solution.

Our platform allows inter-domain communication both in mono- and multi-processed environments. For that, each Application Domain uses a network peripheral, which accesses need to be treated by the hypervisor. It is the hypervisor the module responsible for performing the actual communication, by accessing the memory and copying the desired data. Therefore, tests with communication present higher overhead. We believe that adding extra hardware support, such as extra execution modes (like virtualization mode, proposed by Intel), duplicating certain structures (such as the register bank, to decrease context-switch overhead), and an improving lock system (to provide synchronization) can dramatically decrease these overheads.

Finally, by analysing these results we believe our platform is suitable for monoprocessed and small multiprocessed environments with less than 8 CPUs (due to the shared memory strategy), where applications with high idle time coexist with medium-intensive processing applications. This scenario would benefit from the consolidation offered by virtualization with acceptable penalty due to the low amount of emulations.

VI. Final remarks and future work

In this paper we presented a virtualization model intended for multiprocessed embedded systems where no change in the guest OS is desired and hardware support for virtualization is possible as we suggest some modification to the MIPS 4Kc core in order to provide full-virtualization. The main advantages of our approach are: (i) no guest OS change required (as opposed to para-virtualization approaches); (ii) the strong secure separation between virtual machines, and; (iii) an environment with low execution overhead in some cases when compared to a non-virtualized system. Results were taken aiming to measure the overhead of our approach, since we adopt a classic trap and emulate technique for full-virtualization. Future work includes the support of real-time applications into the platform and other hardware modifications in order to reduce the overhead of privileged instructions' execution.

References

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361073

[2] M. Ito and S. Oikawa, "Mesovirtualization: lightweight virtualization technique for embedded systems," in *Proceedings of the 5th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, ser. SEUS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 496–505. [Online]. Available: http://dl.acm.org/citation.cfm?id=1778978.1779039

[3] J. Brakensiek, A. Drge, M. Botteck, H. Hrtig, and A. Lackorzynski, "Virtualization as an enabler for security in mobile devices," vol. ILES 08, 2008, pp. 17–22.

[4] D. Su, W. Chen, W. Huang, H. Shan, and Y. Jiang, "Smartvisor: towards an efficient and compatible virtualization platform for embedded system," in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, ser. IIES '09. New York, NY, USA: ACM, 2009, pp. 37–41. [Online]. Available: http://doi.acm.org/10.1145/1519130.1519137

[5] A. Cohen and E. Rohou, "Processor virtualization and split compilation for heterogeneous multicore embedded systems," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 102–107. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837303

[6] M. Åsberg, N. Forsberg, T. Nolte, and S. Kato, "Towards real-time scheduling of virtual machines without kernel modifications," in *16th IEEE International Conference on Emerging Technology and Factory Automation (ETFA'11), Work-in-Progress (WiP) session*, September 2011. [Online]. Available: http://www.ipr.mdh.se/index.php?choice=publications&id=2556

[7] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, ser. IIES '08. New York, NY, USA: ACM, 2008, pp. 11–16. [Online]. Available: http://doi.acm.org/10.1145/1435458.1435461

[8] ——, "Virtualizing embedded systems: why bother?" in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 901–905. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024925

[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*. IEEE, Jan. 2009, pp. 1–7. [Online]. Available: http://dx.doi.org/10.1109/CCNC.2009.4784874

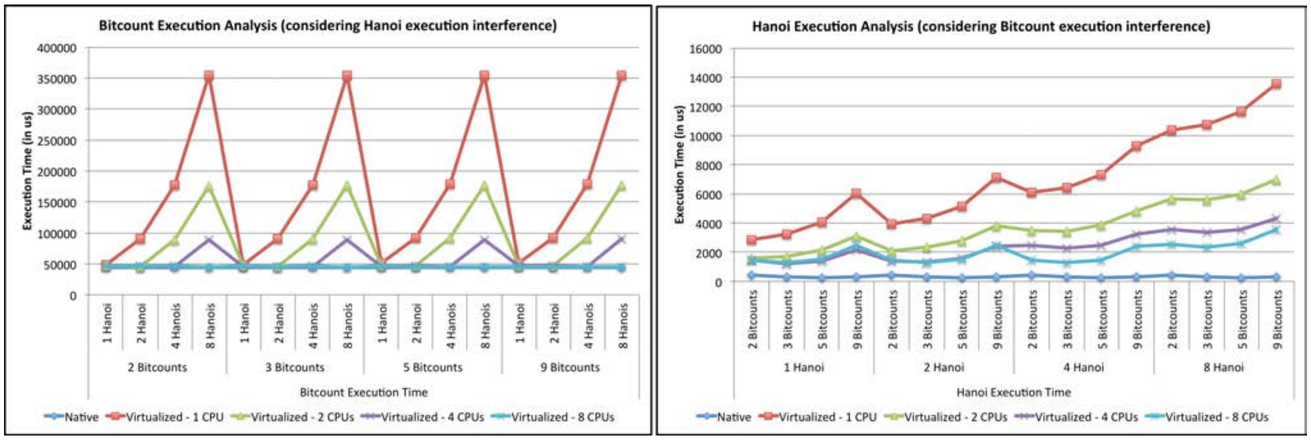

Fig. 7. Mixed Scenario With Bitcount and Hanoi application's interference in each other's execution time

[10] C. Main, "Virtualization on multicore for industrial real-time operating systems [from mind to market]," *Industrial Electronics Magazine, IEEE*, vol. 4, no. 3, pp. 4 –6, sept. 2010.

[11] T.-H. Lin, Y. Kinebuchi, A. Courbot, H. Shimada, T. Morita, H. Mitake, C.-Y. Lee, and T. Nakajima, "Hardware-assisted reliability enhancement for embedded multi-core virtualization design." in *ISORC*. IEEE Computer Society, 2011, pp. 241–249. [Online]. Available: http://dblp.uni-trier.de/db/conf/isorc/isorc2011.html#LinKCSMMLN11

[12] T. Nakajima, Y. Kinebuchi, H. Shimada, A. Courbot, and T.-H. Lin, "Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 645–652. [Online]. Available: http://dl.acm.org/citation.cfm?id=1950815.1950942

[13] N. Li, Y. Kinebuchi, H. Mitake, H. Shimada, T.-H. Lin, and T. Nakajima, "A light-weighted virtualization layer for multicore processor-based rich functional embedded systems." in *ISORC*, C. Hu, G. Karsai, J. Xu, A. Polze, J. Wang, and A. J. Wellings, Eds. IEEE, 2012, pp. 144–153. [Online]. Available: http://dblp.uni-trier.de/db/conf/isorc/isorc2012.html#LiKMSLN12

[14] D. Kleidermacher and M. Kleidermacher, *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Elsevier Science, 2012.

[15] C. Bertin, C. Guillon, and K. De Bosschere, "Compilation and virtualization in the hipeac vision," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 96–101. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837302

[16] OVP, "Open virtual platforms," http://www.ovpworld.org/, Accessed, November 2012, 2012.

[17] MIPS, "MIPS 32 4K - Processor Core Family Software User's Manual," http://www.usrmodem.ru/files/adsl/mips.pdf, Accessed, November 2012, 2012.

[18] A. Aguiar, S. J. Filho, F. G. Magalhaes, T. D. Casagrande, and F. Hessel, in *ISQED*. IEEE, pp. 730–737.

[19] P. Hayes, "A note on the towers of hanoi problem," *The Computer Journal*, vol. 20, pp. 282–285, 1977.