Adapting Embedded Systems' Framework to Provide Virtualization: the Hellfire Case Study

Alexandra Aguiar, Sérgio J. Filho, Felipe G. Magalhães, Fabiano Hessel
Faculty of Informatics – PUCRS – Av. Ipiranga 6681, Porto Alegre, Brazil
E-mail: alexandra.aguiar@pucrs.br, {sergio.johann, felipe.magalhaes}@acad.pucrs.br,
fabiano.hessel@pucrs.br

Abstract— In this paper, we present how it is possible to adapt existing development solutions to provide embedded virtualization advantages. To do so, we use the Hellfire Framework, which offers an integrated tool-flow in which design space exploration (DSE), OS customization and static and dynamic application mapping are highly automated. In this case, the designer can develop embedded sequential and parallel applications while evaluating how design decisions impact in the overall system behavior including analysis regarding whether virtualization usage presents benefits towards the non-virtualized solution.

Keywords— MPSoC, Embedded Systems Design, Embedded Virtualization

I. Introduction

Embedded systems, critical and non-critical, have increasingly become part of people's lives being present in day-by-day life issues, like health-care electronics, automotive industry and entertainment devices. Over the years, Multiprocessors System-on-Chip (MPSoC) have become the most viable choice to implement them [1]. This trend combined with the growing complexity of *real-time* (RT) constrained embedded systems, has a major impact in the entire design and implementation of embedded systems.

Additionally, characteristics present earlier only in general-purpose systems are each more common in embedded systems, which affects directly their design. In this context, we can highlight their growing functionality as the greatest change, affecting dramatically their design and increasing the complexity of their software. Nowadays, it is even common to run general purpose applications in some embedded systems as well as to use applications written by developers that have little or no knowledge at all about the embedded systems constraints [2].

On the other hand, some typical embedded systems' constraints persist and need to be considered. For example, timing constrained MPSoCs with short energy consumption budgets and strict time-to-market need software deployment and test on the target hardware architecture to be performed but this usually is a difficult and time consuming approach. Ergo, it is needed alternative methodologies where the final hardware prototype has equivalence of the code with system's hardware and software simulator and emulator.

Aiming to improve the software design quality, we propose the use of embedded virtualization, which has gained attention over the last few years [3], [4], [5].

Thus, this paper presents adaption possibilities of the Hellfire Framework (HellfireFW) to allow embedded virtualization to be better explored. HellfireFW is an environment for the design of embedded systems' applications which follows its own design flow where an MPSoC emulator is employed and over which a specific RTOS solution - the HellfireOS - is executed [6]. Still, HellfireFW improves the overall system quality, since it helps the designer to perform design space exploration through the customization and simulation of different HW/SW scenarios.

The remainder of the paper is organized as it follows. The next section shows some related work. Section **3** shows virtualization concepts. Section **4** presents the Hellfire Framework followed by Section **5** where virtualization is considered in the framework. Then, Section **6** concludes the paper and presents some future work.

II. Related Work

Design Related. Design space exploration can be achieved in several ways, including co-design based solutions which aim to create a suitable platform architecture [7]. It usually starts at system level, in which the final system division into hardware and software has not been completed yet. After the initial steps, the functional verification can be made through co-simulation mechanisms which help in both system's validation and refinement.

Common embedded systems' constraints, such as a tight time-to-market and the high complexity of current designs, lead to simulators in different abstraction levels, from Instruction Set Simulators (ISS) to system-level descriptions, usually implemented in SystemC or SpecC. In these simulators, usually both hardware and software models are executed in order to analyze the system behavior at an earlier development stage.

In this context, several works [8], [9], [10] and [11] show some effort in modeling the RTOS at a higher level of abstraction. The main issue with this approach is that when the design is refined, although the RTOS model can be translated automatically into software services, usually a widely adopted RTOS is preferred, which prevents the results taken during the modeling to be accurate when the real hardware implementation is finished.

[12] present the *code equivalence* problem, that is, if the code executed by the simulated system is different from the target hardware code, the system's behavior can be different from what was simulated. Still wrong design choices can be made. Moreover, when using high level simulation approaches, usually the system timing behavior is either not taken into account or is not completely accurate [13]. Simulators where timing of the target architecture is not accurately replicated are not allowed to precisely model critical and parallel embedded systems, as MPSoCs.

These researches are valid since they help the designer to refine the HW/SW partition. Unfortunately, this validation is limited due to the code equivalence problem and restrictions of the assessment of the system timing properties. As opposed to some works shown previously, we use ISSs in our study. Although it represents a lower simulation speed, it allows the use of the same application code both in the simulation level and in the target architecture, thus dealing with the code equivalence problem.

Virtualization Related. EmbeddedXEN Project. EmbeddedXEN is an academic project of the XEN.org research group where the main target concerns embedded real-time applications. Its hypervisor is executed in ARM cores and the EmbeddedXEN project provides to ARM developers a single multikernel binary image which includes XEN, Linux, miniOS and XenomaiRT extension adapted to run onto embedded systems. Virtualization and isolation mechanisms are fully relied on the XEN hypervisor for general purpose computers. The main goal of this project is to provide viability and performance evaluation of the embedded virtualization. It is an open-source project and it can be used in any device, although the Server Xen version is not open-source which can restrict its use [14].

OKL4. Implemented by OK Labs (Open Kernel Labs), it is an L4 family microkernel commercially distributed hypervisor with low overhead rates [3]. It has a high performance Inter-Process Communication (IPC) message exchange mechanism, which helps the low overheaded virtualization. A system call that causes a trap triggered by any virtual machine, calls the microkernel exception manager, converting this event into an IPC message to the guest OS. The client deals with this process as a normal system call and the answer is returned through another IPC message [3].

Wind River Hypervisor. It focuses on high performance, small footprint, determinism, low latency and high reliability. It is highly optimized for and integrated with VxWorks and Wind River Linux although it supports other operating systems. In terms of processor, supports single and multicore processors based on Intel and PowerPC architectures and its solution integrates with VxWorks and Wind River Linux. It also enables devices to be assigned to virtual boards as it provides device and memory protection between virtual boards [15].

VirtualLogix VLX. Hypervisor that decouples hardware management (intended for ARM and Intel architectures) and application environments (Android, Linux, proprietary, Symbian, Windows), thus enabling separation of design and functionality concerns. This allows OS/device independence and fault tolerance with minimal overhead, as well as improved performance for multimedia and gaming and enhanced device security through isolation [16].

Trango. It provides a thin layer of code that allows system designers greater flexibility when extending the functionality of an existing system or using multiple OSs. Only a single CPU is then needed to keep the OS and multiple environments separate, so the designer can create trusted areas where secure processes (such as key management or secure boot) can run without adding another CPU [17].

XtratuM. XtratuM is an open source hypervisor specially designed for embedded real-time systems available for x86, PowerPC, MIPS and recently for LEON2 (SPARC v8) processors. It is a hypervisor designed for embedded systems to meet safety critical real-time requirements and provides a framework to run several operating systems in a robust partitioned environment. XtratuM can be used to build a MILS (Multiple Independent Levels of Security) architecture [18].

Analysis. Many embedded hypervisors have emerged within the last few years. Although they have several qualities, our proposal aims to integrate a hypervisor intended for real-time MPSoCs in a well-structured design tool (Hellfire Framework) and it respects a design flow that helps to improve software quality. Also, it allows the use of simpler RISC processors, such as MIPS-based ones.

III. Virtualization basic concepts

Virtualization allows a single physical computer to host multiple virtual machines, each being isolated from one another and it dates back more than 30 years [19]. Several advantages arise from its use, such as the possibility of running different operating systems in the same physical hardware. Still, if a virtual machine fails, the other ones can be kept safe at a reasonable cost [3].

Although it causes a single point of failure, since many servers can be placed at a unique hardware machine, one of the main uses of virtualization lies in enterprise IT. Still, virtualization is considered safer because most of service interrupts are caused by software failures, usually, by the operating system which tends to be big enough so that maintenance is harder [20]. So, the idea is to use simpler and more efficient kernel level software to avoid safety problems. The hypervisor - the main virtualization component - usually is at least two orders of magnitude smaller than general purpose OSs, therefore, it is less likely to have failures [20].

Commonly, two approaches are used to implement the hypervisor which is also known as Virtual Machine Monitor (VMM): In *hypervisor type 1*, also known as *hardware level virtualization*, the hypervisor itself can be considered as an operating system, since it is the only piece of software that works in kernel mode, like depicted in Figure 1. Its main task is to manage multiple copies of the real hardware - the virtual boards (virtual machines or domains) - just like an OS manages multitasking.

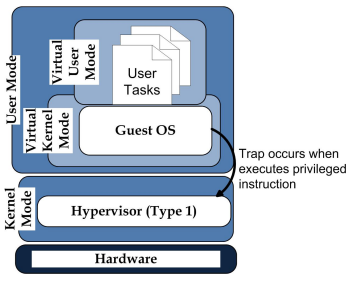

Fig. 1. Hypervisor Type 1

Type 2 hypervisors, also known as *operating system level virtualization*, depicted in Figure 2, are implemented such that the hypervisor itself can be compared to another user application that simply "interprets" the guest machine ISS.

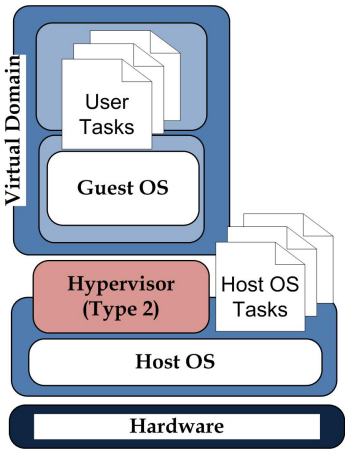

Fig. 2. Hypervisor Type 2

In several aspects, the hypervisor can be considered similar to an operating system. So, some concepts regarding OSs' implementation need to be highlighted. Classic studies of Popek and Goldberg [21] introduce a classification for the instructions of an ISA (Instruction Set Architecture) into three different groups:

1. *privileged instructions*: those that trap when used in user mode and do not trap if used in kernel mode;
2. *control sensitive instructions*: those that attempt to change the configuration of resources in the system, and;
3. *behavior sensitive instructions*: those whose behavior or result depends on the configuration of resources (the content of the relocation register or the processor's mode).

Also, Popek and Goldberg declared that in order to virtualize a given machine, sensitive instructions must be a subset of the privileged instructions. Unfortunately, this is not a reality in many processors, such as Intel's x86 family and the common solution in this case, is to adopt processor's hardware support. Intel's support is named as VT (Virtualization Technology) and AMD's named as SVM (Secure Virtual Machine). In spite of it, hardware support is hard to be achieved for embedded systems yet, so other options have to be considered at the present time. In the remainder of the paper only hardware level (hypervisor type 1) is considered.

In this type of virtualization, when no hardware support is available, the hypervisor is at charge of translating instructions whenever the virtual domain attempts to execute a privileged instruction (I/O request, memory write etc), which causes a trap into the hypervisor, being known as pure virtualization. This is often a very expensive way of dealing with virtual machines [22].

Still, an alternative known as impure virtualization can be used as it requires that sensitive instructions (those that require a trap into the hypervisor) are removed from the code executing in the virtual machine. This can be done either at compile time, by a technique called *pre-virtualization* or by *binary code rewriting*, where the executable code is scanned in order to replace such instructions. The main issue is that both approaches can cause huge performance losses.

Finally, *para-virtualization* it a technique that replaces sensitive instructions of the original kernel code by explicit hypervisor calls (also known as *hypercalls*). The goal of para-virtualization is to reduce the problems encountered when dealing with different privilege levels. Usually, a scheme referred to as *protection rings* is used and it guarantees that the lower level rings (Ring 0, for instance) holds the highest privileges. So, most of OSs are executed in Ring 0, thus being able to interact directly with the physical hardware.

In this case, when the hypervisor is adopted, it becomes the only piece of software to be executed in Ring 0, bringing severe consequences for the guest OSs: they are no longer executed in Ring 0, instead, run in Ring 1, with fewer privileges. This problem, known as ring de-privileging is depicted in Figure 3.

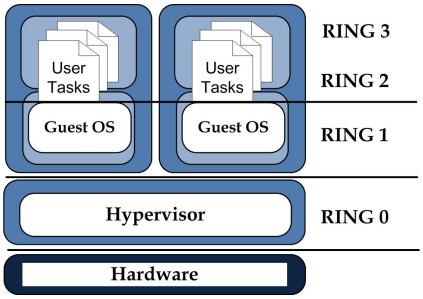

Fig. 3. Ring de-privileging caused by the hypervisor

IV. HELLFIRE SYSTEM

The Hellfire System (HFS) is an academic project born at the Embedded Systems Group (GSE) of the Pontifical Catholic University (PUCRS), in Brazil. The project provides its own design flow that comprehends different abstraction levels, from C application development to FPGA prototyping. This design flow is supported by several tools and modules that compose the Hellfire Framework (HFFW).

From a single processor point of view, the designer can develop the application C code and run it over the Hellfire Operating System (HellfireOS), which is a highly configurable real-time modular micro-kernel based OS. From the platform point of view, the designer can add up to 128 processors to the system, configure each one of them and, by using the HellfireOS API, develop parallel embedded applications which are able to exchange data and even able to migrate tasks. Currently, only MIPS based processors are allowed.

A. HellfireOS

The HellfireOS (HFOS) [6] is a real-time operating system (RTOS) developed in order to ensure maximum flexibility in its configuration and allow a high level platform customization. In order to allow such feature, HFOS was implemented in a modular way, where each module corresponds to some specific functionality.

Figure 4 depicts the kernel modular organization. All hardware-specific functions are defined in the first layer, known as HAL (Hardware Abstraction Layer) and the uKernel lies just above it. The communication, migration, memory management and mutual exclusion drivers, as well the API are placed over the uKernel layer. The user applications belong to the top layer.

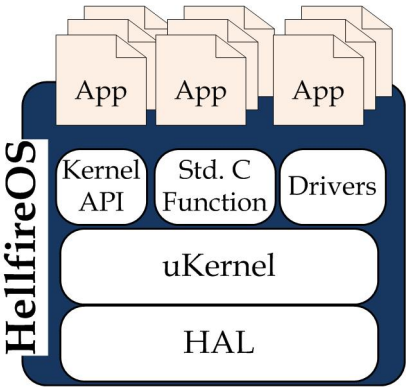

Fig. 4. HellfireOS Structure

Due to its modular implementation, HellfireOS is easily portable to others architectures, requiring only the rewrite of hardware-dependent functions, such as the interrupt service, its initialization routine and the context switch. In order to decrease the kernel final size, allowing the HFOS usage even in architectures with severe memory limitations, parameters such as users tasks maximum number, stack and heap size and drivers usage, are configurable.

The users' applications are written using basic *C* programming language and the HellfireOS API. In Figure 5 a user task example that prints the CPU usage in the standard output of the system is shown.

```
void cpu_usage_test(void) {

    while (1) {
        printf ("\nMemory Usage %d", OS_GetCpuUsage());
    }
}
```

Fig. 5. Task Example

Another configurable parameter is the activation bit used by the timing register, which determines the system tick size. The tick size corresponds to the minimum periodic timing unit of the system. This unit varies from 0.32ms to 83.88ms. Figure 6 shows the relation between the processor clock frequency rate and the activation bit.

Core Frequency	Activation bit	Tick time (ms)	Ticks/second
25 MHz	15	1.31	763.36
	16	2.62	381.68
	17	5.24	190.84
	18	10.48	95.42
	19	20.97	47.69
	20	41.94	23.84
	21	83.88	11.92
33 MHz	15	0.99	1010.10
	16	1.98	505.05
	17	3.97	251.89
	18	7.94	125.94
	19	15.88	62.97
	20	31.77	31.48
	21	63.55	15.74
50 MHz	15	0.65	1538.46
	16	1.31	763.36
	17	2.62	381.68
	18	5.24	190.84
	19	10.48	95.42
	20	20.97	47.69
	21	41.94	23.84
66 MHz	15	0.49	2040.82
	16	0.99	1010.10
	17	1.98	505.05
	18	3.97	251.89
	19	7.94	125.94
	20	15.88	62.97
	21	31.77	31.48
100 MHz	15	0.32	3125.00
	16	0.65	1538.46
	17	1.31	763.36
	18	2.62	381.68
	19	5.24	190.84
	20	10.48	95.42
	21	20.97	47.69

Fig. 6. Tick Time x Core Frequency

B. Hellfire Framework

The Hellfire Framework (HFFW) [6] allows a complete deployment and test of parallel embedded applications, defining the HW/SW architecture to be employed by the designer. The HFFW is divided in three modules as it follows, which are discussed throughout this section.

- HellfireOS, discussed previously in Section **4**-A;
- N-MIPS MPSoC Simulator, and;
- architecture builder.

The N-MIPS MPSoC Simulator is an Instruction-Set Simulator (ISS) based on MIPS-like [23] cores. It was also written in *C* and provides simulation of up to 128 processing cores.

A very important feature is the possibility to prototype in FPGA the same resulting image file used during the simulation, which increases the chances of prototyping success after the simulation.

The last highlighted module, named *architecture builder* is used to specify the target architecture and to configure all the HellfireOS parameters.

The HellfireFW design flow is basically divided in three steps: the application design, the project creation and simulation/refinement, explained in the remainder of this section.

Application Design. The first step of the design flow is the development of an application implemented in *C* language and manually divided into a set of tasks. These tasks are defined as n-uples $(id_i, r_i, WCET_i, D_i, P_i)$ and the parameters stand for identification, release time, worst case execution time, deadline and period of each task respectively.

HellfireFW Project Creation. After the application design, the designer must create a project in HFFW. This step corresponds to the architecture design and configuration. The designer must define the initial hardware architecture, specifying the number of processors and their frequencies and configuring the HellfireOS parameters, as presented in Section **4**-A. After that, the mapping of the tasks along the processors must be performed. This mapping is done manually and is based on the designer's experience.

As the output of this step, an MPSoC platform is expected, having a given number of processors, a personalized instance of HellfireOS on each processor and a static task mapping.

Simulation and Refinement. When the designer triggers a simulation, the MPSoC ISS Simulator is activated. After the simulation is completed, several reports are presented to the designer both in textual and graphical way. The textual reports are the following:

- standard output of each processor;
- report with all Plasma instructions used, the number of times that each was used and an usage percentage of each group (logical, arithmetic, etc) of instructions;
- an energy consumption summary, based on [24];
- report containing the main characteristics of the system, such as deadlines misses and CPU load, and;
- individual report of the cycle to cycle operation of each processors. All information contained on the system stack is shown in this report.

V. Adapting HellfireFW to Virtualization

This section depicts the main changes in the Hellfire Framework design flow and the main advantages to use virtualization. We use the Virtual-Hellfire Hypervisor (VHH) architecture [5], based in the HellfireOS structure. The main advantages of VHH are:

- temporal and spatial isolation among domains (each domain contains its own OS);
- real-time scheduling policy for domain scheduling;
- deterministic hypervisor system calls (hypercalls).

Figure 7 depicts the Virtual-Hellfire Hypervisor structure. In this figure, the hardware continues to provide the basic services as timer and interrupt but they are managed by the hypervisor, which provides hypercalls for the different domains, allowing them to perform privileged instructions.

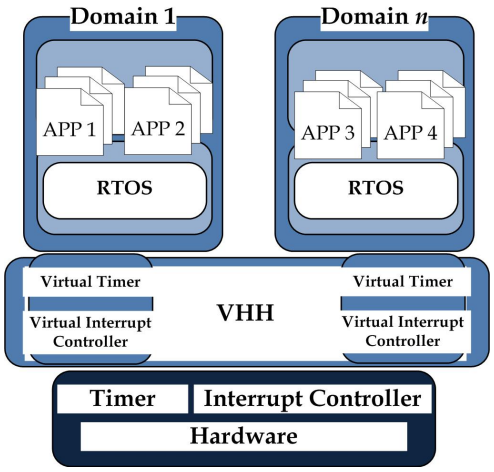

Fig. 7. Virtual-Hellfire Hypervisor Domain structure

In terms of memory management, in MMU-less processors a possible choice is to implement a software virtual memory management, as proposed in [25]. Another viable strategy is to use a fixed partition memory scheme. In this case, the required amount of memory is allocated to each domain at boot (or load) time, meaning that its size cannot grow or shrink at run time. If the application code of the guest OS of a given domain requires dynamic memory (such as with *malloc* or *free* C primitives), the heap needs to be managed by the domain's code itself. For now, VHH uses this second option (fixed partition memory), as we can see in Figure 8, where each processor (Processing Element - PE, in the figure) of the MPSoC has its own Local Memory (LM). This memory is divided according to the amount of partitions that this processor will hold.

The internal architecture of HellfireOS had to be modified to guarantee the use of virtualization. As a matter of fact, we kept some of the original features and took advantage of its highly modular implementation by adding the necessary modules to provide virtualization. Thus, Virtual-Hellfire Hypervisor is implemented based on the following layers:

- **Hardware Abstraction Layer - HAL**, responsible for implementing the set of drivers that manage the mandatory hardware, like processor, interrupts, clock, timers etc;

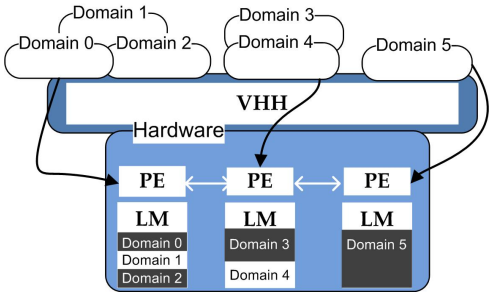

Fig. 8. Virtual-Hellfire Hypervisor Memory Management

- **Kernel API and Standard C Functions**, which are not available to the partitions;
- **Virtualization layer**, which provides the services required to support virtualization and para-virtualization services. The hypercalls are implemented in this layer.

In this new layer responsible for allowing virtualization to be used, there are some mandatory modules, such as:

- *domain manager*, responsible for domain creation, deletion, suspension etc;
- *domain scheduler*, responsible for scheduling domains in a single processor;
- *interrupt manager*, that handles hardware interrupts and traps. It is also in charge of triggering virtual interrupt and traps to domains;
- *hypercall manager*, responsible for handling calls made from domains, being analogous to the use of system calls in conventional operating systems;
- *system clock provider*, in which two clocks per domain are implemented: one that only advances while the domain is being executed (virtual) and a real, counted from the boot time.
- *timer provider*, similar to clock implementation, provides virtual and real timers, both accessible by hypercalls;
- *memory manager*, divided in virtual and physical management, according to the underlying hardware;
- *system output facility*, where all messages are queued and can be redirected to hardware peripherals, such as a serial port.

A very interesting point of the VHH is the use of an MPSoC as underlying hardware. We assume the use of a Symmetric MultiProcessor (SMP) and the hypervisor acts as a MultiProcessor RTOS (MP-RTOS). The hypervisor is aware of the several domains and respects their own scheduling policies.

Each processor has its ready task queue, which can contain tasks from different virtual domains. For each processor, the highest-priority task in the ready queue is executed. To avoid starvation of non real-time tasks (when allocated in the same processor of real-time tasks), it is possible to adopt a scheduling policy that guarantees the execution of best-effort tasks, such as R-EDF [26].

The mapping of virtual domains onto real processors is done at design time. For now, it is the designer's responsibility to associate virtual domains and real processors. In the future, we intend to use virtualization even as a load balancing solution, where, dynamically, virtual domains can migrate among the several processors of the MPSoC to improve a given measure, as performance or energy consumption. When more than one do-

main is mapped for a single processor, the scheduling among domains occurs according to a fixed priority scheduling. Then, domains are scheduled by the hypervisor considering its priority level.

Since HellfireOS is integrated in the Hellfire Framework with several simulation facilities, VHH is also integrated in it and requires the designer to choose whether virtualization is enabled. In this case, although user-transparent, the design flow presented by Figure 9 is employed. This flow starts with the configuration of the VHH, where the number of domains is informed and the VHH core is generated. Following, each of the desired domains is configured in a very similar way Hellfire Framework used to do with non-virtual HellfireOS edition: application tasks were added and put together with the OS image. Finally, all system is assembled and executed by an ISS-like (Instruction Set Simulator) simulator.

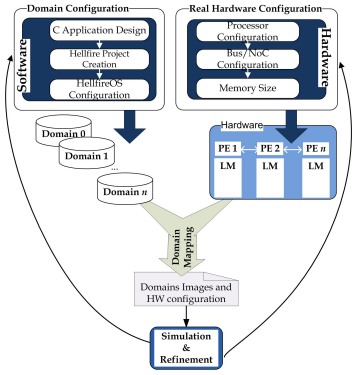

Fig. 9. VHH Integrated in the Hellfire Framework

VI. Concluding Remarks and Future Work

Embedded systems are becoming more complex and new design solutions need to be investigated. In this paper the HellFire framework was presented. This framework helps to develop, validate and test embedded systems' applications. The solution provides an easy way to develop and validate embedded software in a given platform and also to perform HW/SW design space exploration allowing the designer to combine different SW mapping possibilities onto several HW platform architectures, as well as several OS configurations to analyze different aspects of the system. Still, we discuss the use of a virtualization methodology. We demonstrate the usefulness of virtualization in embedded systems' design and how it can be applied in current design flows.

Future work includes measuring and getting comparison results for performance, area and energy consumption with non-virtualized systems. Still, we want to measure precisely the overhead of the proposal besides improving the methodology itself, especially regarding memory and I/O management as well as to improve these features' implementation quality.

Acknowledgment

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

References

[1] Ahmed Jerraya, Hannu Tenhunen, and Wayne Wolf, "Multiprocessor systems-on-chips," *Computer*, vol. 38, no. Issue 7, pp. 36– 40, July 2005.

[2] Y. Zorian and E. Marinissen, "System chip test - how will it impact your design," in *DAC'2000 - Design Automation Conference*, Las Vegas, EUA, Jun 2000, ACM Press.

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

[4] A. Aguiar and F. Hessel, "Embedded systems' virtualization: The next challenge?," in *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*, 2010, pp. 1 –7.

[5] A. Aguiar and F. Hessel, "Virtual hellfire hypervisor: Extending hellfire framework for embedded virtualization support," in *To appear in Quality Electronic Design (ISQED), 2011 12th International Symposium on*, 2011.

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

[7] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, April 2003.

[8] A. Gerstlauer, Haobo Yu, and D.D. Gajski, "Rtos modeling for system level design," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 130–135.

[9] R. Le Moigne, O. Pasquier, and J.-P. Calvez, "A generic rtos model for real-time systems simulation with systemc," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, Feb. 2004, vol. 3, pp. 82–87 Vol.3.

[10] H. Posadas, J. Adamez, P. Sanchez, E. Villar, and F. Blasco, "Posix modeling in systemc," in *Design Automation, 2006. Asia and South Pacific Conference on*, Jan. 2006, pp. 6 pp.–.

[11] A. Shaout, K. Mattar, and A. Elkateeb, "An ideal api for rtos modeling at the system abstraction level," in *Mechatronics and Its Applications, 2008. ISMA 2008. 5th International Symposium on*, May 2008, pp. 1–6.

[12] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya, "Automatic generation of fast timed simulation models for operating systems in soc design," in *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, Washington, DC, USA, 2002, p. 620, IEEE Computer Society.

[13] Gunar Schirner and Rainer Dömer, "Introducing preemptive scheduling in abstract rtos models using result oriented modeling," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, New York, NY, USA, 2008, pp. 122–127, ACM.

[14] XEN.org, "Embedded xen project.," Web, Available at http://www.xen.org/community/projects.html. Accessed at 10 ago., 2010.

[15] Wind River, "Wind river," Web, Available at http://www.windriver.com/. Accessed at 2 oct., 2010.

[16] VirtualLogix VLX, "Real-time virtualization for connected devices," Web, Available at http://www.virtuallogix.com/. Accessed at 2 oct., 2010.

[17] Trango, "Trango hypervisor," Web, Available at http://www.trango.com/. Accessed at 2 oct., 2010.

[18] XtratuM, "Trango hypervisor," Web, Available at http://www.trango.com/. Accessed at 2 oct., 2010.

[19] Robert P. Goldberg, "Survey of virtual machine research," *Computer*, pp. 34–35, 1974.

[20] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

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

[22] Carl A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[23] Open Cores, "Plasma most mips i(tm) opcodes," http://www.opencores.org.uk/projects.cgi/web/mips/, Accessed, September 2009, 2007.

[24] Sergio Johann Filho, Alexandra Aguiar, César Augusto Missio Marcon, and Fabiano Passuelo Hessel, "High-level estimation of execution time and energy consumption for fast homogeneous mpsocs prototyping," in *RSP '08: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, Washington, DC, USA, 2008, pp. 27–33, IEEE Computer Society.

[25] Siddharth Choudhuri and Tony Givargis, "Software virtual memory management for mmu-less embedded systems," Tech. Rep., 2005.

[26] Wanghong Yuan, K. Nahrstedt, and Kihun Kim, "R-edf: a reservation-based edf scheduling algorithm for multiple multimedia task classes," 2001, pp. 149 –154.