Virtual Hellfire Hypervisor: Extending Hellfire Framework for Embedded Virtualization Support

Alexandra Aguiar, Fabiano Hessel
Faculty of Informatics – PUCRS – Av. Ipiranga 6681, Porto Alegre, Brazil
E-mail: alexandra.aguiar@pucrs.br, fabiano.hessel@pucrs.br

Abstract— Virtualization of embedded systems has recently been in the spotlight especially because of the numerous advantages it can bring. Among these, the improvement of software design quality can be highlighted, since legacy software can be reused along with newer applications, easing newer and older systems' integration. Also, security concerned systems can enjoy the gains of virtualization: two Operating Systems (OS) can be used, namely an application OS and a security certified OS, both running on the same machine. Though virtualization can offer so many benefits, its use in embedded systems is still not as wide as it should or could be. The numerous constraints of embedded systems combined with suspicious thoughts whether virtualization overheads are prohibitive or not prevent its wide adoption. Thus, we present in this paper a methodology for an extension of the Hellfire Framework Project and the creation of the Virtual Hellfire Hypervisor - VHH. The Hellfire Framework already offers an integrated tool-flow in which Design Space Exploration (DSE), OS customization and static and dynamic application mapping are highly automated. Therefore, we show the potential benefits of integrating existing embedded systems tools, like the Hellfire Framework, to virtualization facilities and how this can impact in the overall system design quality.

Keywords—Virtualization, MPSoC, Embedded Systems Design, HW/SW Co-design

I. Introduction

In the past years, the use of Embedded Systems (ES) has changed from an optional and non essential good to become a reality for the overwhelming majority of the population. This includes several fields of applications, which can be from [1]:
• the entertainment world (smart phones, cell phones, video cameras, digital cameras, games toys etc);
• the medical world (dialysis machines, infusion pumps, cardiac monitors etc);
• the automotive business (entertainment centers, engine controls, security, ABS etc);
• aerospace and defense (flight management, smart weaponry, jet engine control etc), and;
• several other fields (industrial automation, office automation, industrial control etc).

This wide range of applications impacts directly on the design of embedded devices. It also implies that not all ESs have the same constraints or goals. Whereas in some systems a given failure can cause serious damage to the environment or even be the major cause of millions of human lives' losses (Critical Embedded Systems), in others these failures only cause performance degradation and, in spite of being accepted, are not desired (Non-Critical Embedded Systems).

Increasingly, embedded systems are bringing typical characteristics of general-purpose systems to their design. The main change is their growing functionality which dramatically affects and increases the complexity of their software. It is also very 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].

Within this context, the classical model of embedded systems' design [1] where software and application layers were considered *optional* is losing space to current trends. Now, designers tend to implement critical and non-critical tasks in software and applications layers, since it allows a higher flexibility, easier debug and higher reuse rates.

Still, some of the traditional differences between general purpose and embedded systems still remain [3]. Even on high-end multimedia entertainment-driven embedded systems, some real-time constraints last. For a great share of the embedded devices, energy consumption is still a matter of concern, which impacts on the processor frequency choice: usually lower frequency rates are mandatory in order to accomplish energy consumption goals. Another common restriction regards the memory use, since modern embedded devices are desired to be cost effective which conflicts with an excessive use of memory. It is important to highlight that, besides being a high energy consuming device, memory is frequently a cost factor issue [4].

In a contradictory way, while some ESs are more concerned in area and energy consumption reduction, such as cell phones, others need the most predictable and deterministic behavior in spite of pure performance levels, such as some avionic systems. This peculiarity directly affects the processor choice, thus, justifying that, in embedded systems, so many predominant architectures such as ARM, MIPS, PowerPC and even some Intel Atom versions are used, whereas general purpose systems are mainly implemented onto the x86 architecture.

Among these (sometimes contradictory) implementation characteristics, a common way to arrange ESs is the use of multiprocessed platforms, where Multiprocessors System-on-Chip (MPSoC) have become a viable choice [5]. In terms of software design, before the rise of MPSoCs, embedded systems used to have a well defined and successful programming model, which includes the use of Real-Time Operating Systems (RTOS) and critical functions of a given device. Instead, with the wide adoption of multicore hardware and the incessant increase of desired functionalities in embedded devices, especially multimedia ones, a true change in the way embedded developers are designing their systems is required.

Several solutions arise including the use of virtualization, a successful General Purpose (GP) computing technique, which can increase ESs' performance and software design quality while reducing its manufacturing costs. Nevertheless, there are several key differences in the way that ES developers face the use of multicore processors and virtualization techniques when compared to those of GP computing [6].

While virtualization provides advantages such as the capability of running multiple instances of operating systems on a single processor (mono- or multi-core), embedded systems are far different from enterprise systems [7]. This means that in order to take the advantages offered by virtualization, much effort must be spent in understanding how to better adapt them to embedded's special needs.

In this paper, we present a methodology to extend an embedded system design tool - the Hellfire Framework - in order to aggregate virtualization facilities to it, known as Virtual Hellfire Hypervisor - VHH. We show how to do it and the expected advantages and issues of our approach. The main contribution of this paper is to provide a novel methodology to allow virtualization to be incorporated in current embedded systems tools.

The remainder of the paper is organized as it follows. The next section shows the basic concepts of virtualization. Section **3** shows some related work, followed by Section **4**, that presents a motivational example to use it in embedded systems. Section **5** briefly presents the Hellfire framework. Section **6**, discusses the possibilities to extend the Hellfire Framework to allow virtualization to be achieved whereas Section **7** concludes the paper besides presenting some future work.

II. Classic virtualization concepts

Virtualization is an old technique that dates back more than 30 years [8]. It allows a single physical computer to host multiple virtual machines, each being isolated from one another. 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 [9].

This approach is very common in enterprise IT, although it causes a single point of failure, since many servers can be placed at a unique hardware machine. Even so, 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 [10]. 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 [10].

To implement the hypervisor, also known as Virtual Machine Monitor (VMM), commonly two approaches are used. 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.

Type 2 hypervisors, also known as *operating system level virtualization*, depicted in Figure 2, are implemented such that the

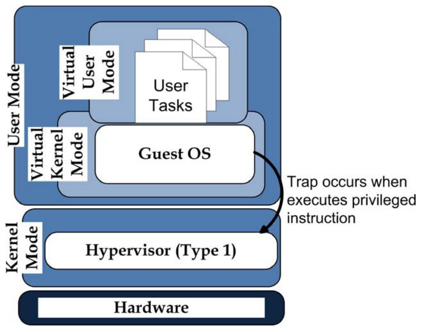

Fig. 1. Hypervisor Type 1

hypervisor itself can be compared to another user application that simply "interprets" the guest machine ISS.

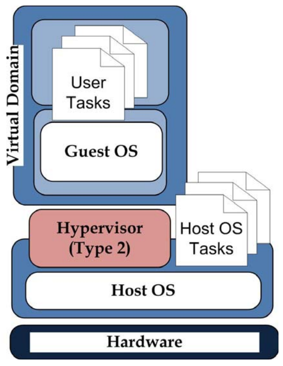

Fig. 2. Hypervisor Type 2

Since the hypervisor can be considered similar to an operating system in many aspects, concepts regarding OSs' implementation are important to be highlighted. Thus, classic studies of Popek and Goldberg [11] 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).

Moreover, those researches first declared that in order to virtualize a given machine, sensitive (control and behavior) instructions must be a subset of the privileged instructions. This is not a reality in many processors, as Intel's x86 family and the common solution in this case, is to adopt processor's hardware support. To name, Intel's support is named as VT (Virtualization Technology) and AMD's named as SVM (Secure Virtual Machine). Hardware support is not an option for embedded systems yet, so other options have to be considered in the present time.

Considering the options to virtualize systems - OS and hardware level - it is important to highlight that at hardware level we may need some support from the processor and at OS level the virtual boards share both the hardware and the host's operating system. Since one of the most promising advantages of using virtualization is to allow several operating systems in a single hardware, OS level virtualization will no longer be considered in the remainder of the paper.

In this case, we will detail some concepts regarding hardware level virtualization without hardware support. Here, the hypervisor is at charge of translating instructions whenever the virtual board 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 [12].

Another option at hardware level is known as impure virtualization and 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.

Alternatively, *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.

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.

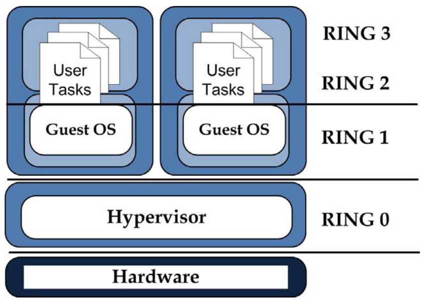

Fig. 3. Ring de-privileging caused by the hypervisor

When para-virtualization is adopted, the hypervisor must define an interface composed by system calls allowed to be used by the guest OS. Besides working on an unsuitable hardware for pure virtualization, it can also bring performance boost.

The difference between pure virtualization and para-virtualization is depicted in Figure 4. In part A of the figure, pure virtualization is showed. In this case, whenever the guest OS calls a sensitive instruction, a trap is caused to the hypervisor, which emulates the instruction behavior and returns the proper results. In part B, para-virtualization is showed. The guest OS has been modified in order to make hypercalls instead of containing sensitive instructions. In this case, the trap is similar to the one that occurs in non virtualized systems, whenever a user application makes a system call on its OS.

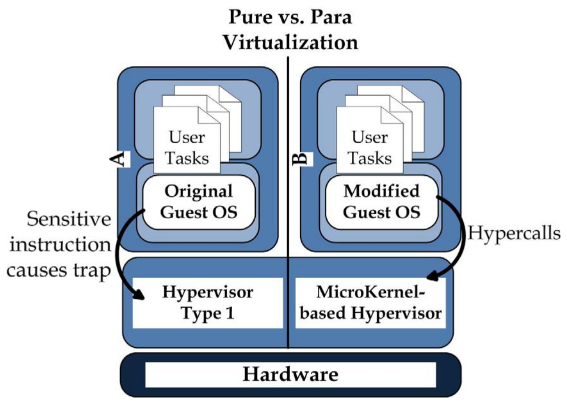

Fig. 4. Hypervisor control of pure virtualization (part A) and para-virtualization (part B)

III. Related Work

In this section, we discuss the existing hypervisors strategies that allow embedded virtualization.

EmbeddedXEN Project. EmbeddedXEN is an academic project of the XEN.org research group where the main target are embedded real-time applications. Its hypervisor is executed in ARM cores and the EmbeddedXEN project provides to ARM developers a single multi-kernel 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 [13].

OKL4. Implemented by OK Labs (Open Kernel Labs), it is an L4 family microkernel commercially distributed hypervisor with low overhead rates [9]. 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 [9].

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 [14].

VirtualLogix VLX. Hypervisor that decouples hardware management (intended for ARM and Intel architectures) and application environments (Android, Linux, proprietary, Symbian, Windows), 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 [15].

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 [16].

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 [17].

Analysis. Many embedded hypervisors have emerged within the last few years. Although they have several qualities, to the best of our knowledge, our proposal is the first that aims to integrate the following features:

- it is a hypervisor intended for real-time MPSoCs;
- it allows several virtual machines on a single processor of a given MPSoC;
- it is integrated with a well structured design tool (Hellfire Framework) and it respects a design flow that helps to improve software quality;
- it allows the use of simpler RISC processors, such as MIPS-based ones.

IV. Motivational Use Cases

Virtualization can be applied in a wide variety style for embedded systems. Here, we highlight some motivational examples for its use [18]. The first case for virtualization on embedded systems consists of enabling several operating systems to be executed concurrently, allowing:

1. legacy software to co-exist with current and incompatible applications, and;
2. real-time software and user interface applications separation, by using different OSs.

In this case, virtualization can strongly increase software development quality, since it allows the designer to choose among several OSs, the most suitable for the application or even the one that presents the best cost/performance ratio. Moreover, the time required to develop an application can be reduced, since legacy applications can simply be reused in virtual machines.

Furthermore, unified software architecture for multiple hardware platforms can be achieved, since the software architecture is developed for the virtual machines and virtualization deals with the several hardware platforms. In this case, a current issue in embedded systems - software portability - could be widely affected and developers would be able to faster satisfy the increasingly restricted time-to-market. The combination of real-time, legacy and user application operating systems in the same device is achieved by virtualization, as depicted in Figure 5.

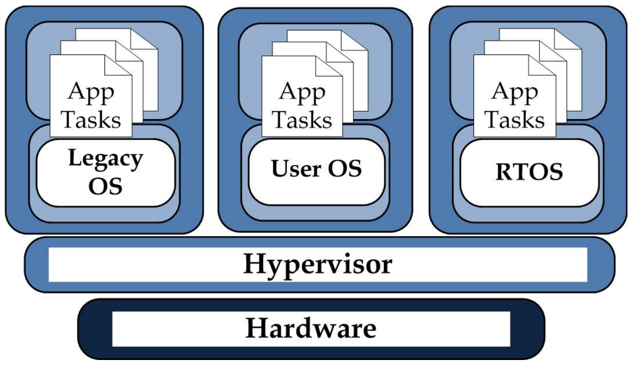

Fig. 5. Legacy software use along with new user applications

Besides, security levels can be increased since virtualization provides a protective environment that encapsulates embedded operating systems and other crucial software components. This approach, where an application specific operating system is kept apart from the RTOS as a way of avoiding attacks, is depicted in Figure 6. However, in order to actually guarantee this improvement of security, the underlying hypervisor *has* to be significantly more secure that the guest OS. According to Tanembaum [10], the most suitable way of achieving it, is to keep the hypervisor as small as possible.

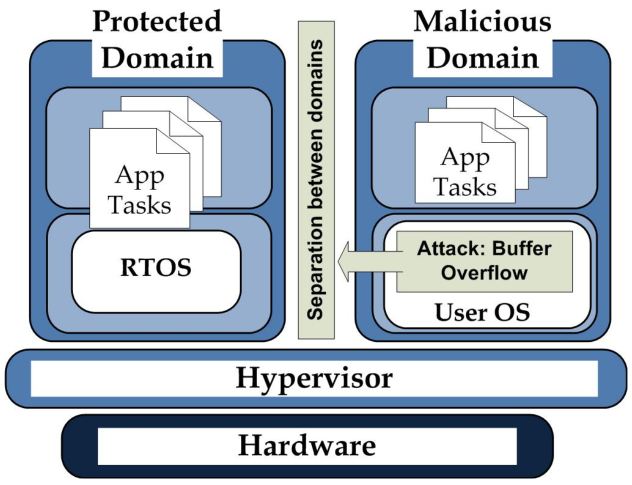

Fig. 6. User attack blocked by virtual machines' isolation

Another advantage of using virtualization when MPSoCs are employed is to ease the workload balance management. The parallelism can be extracted at the application level, that is, entire applications can be migrated through different virtual machines since they have the same OS, as depicted in Figure 7. The advantages of migration in embedded systems have been widely proved throughout the years [19], [20], [21].

Also, in addition to these cases, virtualization has been considered for some researchers to be used in critical embedded systems, such as in avionic matters [22]. Usually, security sensitive or mission critical parts need a protected environment [23]. Then, the sensitive parts have their own OS and the hypervisor separates them from non-trusted OSs and applications.

The separation showed previously allows creative and useful arrangements, such as when some parts of the system are

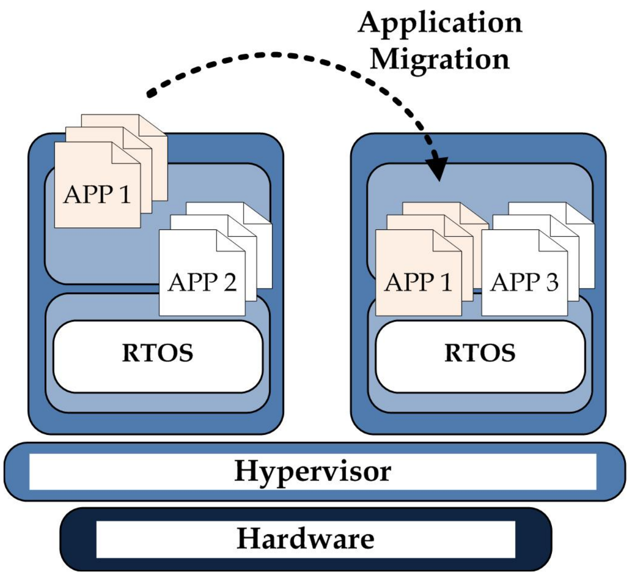

Fig. 7. Migration of applications between virtual machines

required to boot up faster than others. For instance, a car or a camera must have some of their features available at a really fast pace (tens of milliseconds after power on). A general purpose OS will take much longer, therefore, virtualization can separate the functions to be ran in exclusive virtual machines, boosting their boot time.

Separation also allows license protection to be achieved, since proprietary application can be completely isolated from GPL OS. Intellectual Property (IP) protection can rely on virtualization's inherit separation, since private modules are safe from user's inappropriate handling. Firmware over the air (FOTA) upgrades could also be easier to be made with virtualization thus allowing that only a given part of the system reboots after the upgrade [9].

Finally, easier application migration would allow extensive use cases for pervasive computers, as virtual machines could migrate among different devices, leading to a whole new level of remote device usage [24].

V. Hellfire Framework

The present work is an extension of the Hellfire Framework (HellfireFW) [25] which allows a complete deployment and test of parallel critical and non-critical embedded applications, defining the HW/SW architecture to be employed by the designer. The HellfireFW follows a design flow where several steps can be performed aiming to develop the HW/SW solution for a given application. This design flow is presented in Figure 8.

In terms of application design, the entry point in C language, where an application is manually divided into a set of tasks. Each task τ_i is defined as a n-uple $(id_i, r_i, WCET_i, D_i, P_i)$ and the parameters stand for identification, release time, worst case execution time, deadline and period of task τ_i, respectively. They can communicate either through shared memory (in the same processor) or message passing (in different processors).

After designing the application, the HellfireFW project must

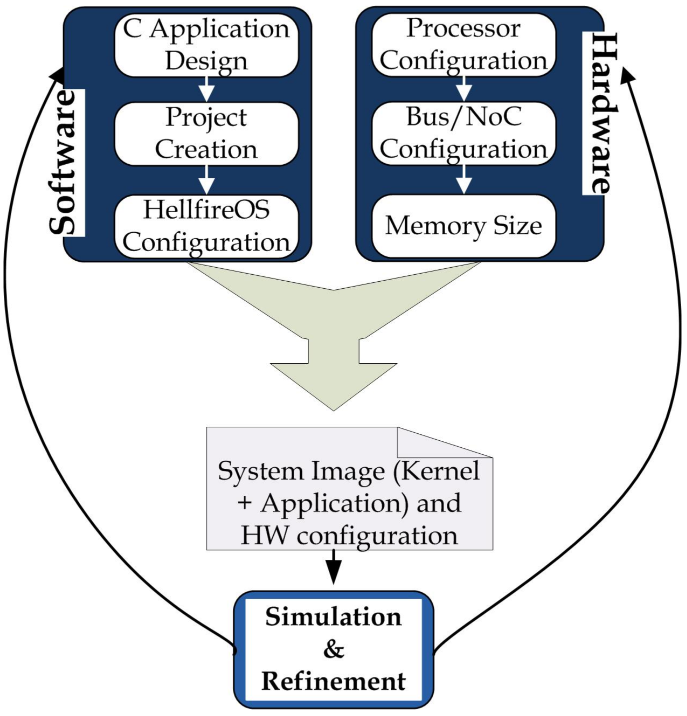

Fig. 8. Hellfire Framework Design Flow

be created. This is the step where the initial HW/SW platform configuration is defined. The C application is executed on the top of the HellfireOS stack. HellfireOS [25] is a micro-kernel based Real-time Operating System - RTOS, highly configurable and easily portable. To ease the OS port to other architectures, HellfireOS uses a modular structure as depicted in Figure 9.

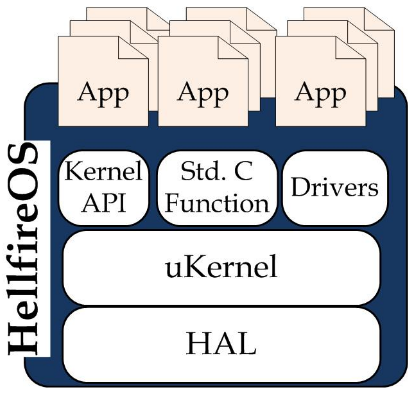

Fig. 9. HellfireOS Structure Stack

All hardware specific functions and definitions are implemented on the Hardware Abstraction Layer (HAL), which is unique for a specific hardware platform solution, simplifying the port of the kernel onto different platforms. The micro-kernel itself is implemented on top of this layer. Features like standard C functions and the kernel Application Programming Interface (API) are implemented on top of the micro-kernel. Communication and migration drivers, memory management and mutual exclusion facilities are implemented on top of the kernel API and the user application is the highest level in the HellfireOS development stack.

After following these steps an MPSoC platform configuration is expected, with a given number of processors, a personalized instance of HellfireOS on each processor and a static task mapping. The user must then trigger the simulation of the system which runs for a given time window and then generates several graphical results for the designer to analyze. If the results are satisfactory, the SW part of the platform can be easily ported to a prototype, such as an FPGA. Otherwise, the designer can change the HW/SW settings and rerun the simulation as refinements are needed.

VI. Virtual-Hellfire Hypervisor (VHH), a HellfireOS based Hypervisor

In this section we describe the Virtual-Hellfire Hypervisor (VHH) architecture, based in the HellfireOS structure. The main advantages of VHH are:

- temporal and spatial isolation among domains (each domain contains its own OS);
- resource virtualization: clock, timers, interrupts, memory;
- efficient context switch for domains;
- real-time scheduling policy for domain scheduling;
- deterministic hypervisor system calls (hypercalls).

VHH considers a **domain** as an execution environment where a guest OS can be executed and it offers the virtualized services of the real hardware to it. As detailed in Section **2**, for embedded systems where no hardware support is offered, para-virtualization tends to present the best performance results. Therefore, in VHH, domains need to be modified before being executed on top of it. As a result, they do not manage hardware interrupts directly. Instead, the guest OS must be modified to allow the use of virtualized operations provided by the VHH (hypercalls).

Figure 10 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.

In terms of memory management, in MMU-less processors a possible choice is to implement a software virtual memory management, as proposed in [26]. 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 11, 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.

Another point of concern is the dealing of I/O peripherals. Xen [27] is one of the most successful para-virtualized implementations for desktop systems and it uses the concept of a specific I/O domain (known as *Domain 0*). This is needed because most peripherals must be managed by a single software driver which is aware of its current status.

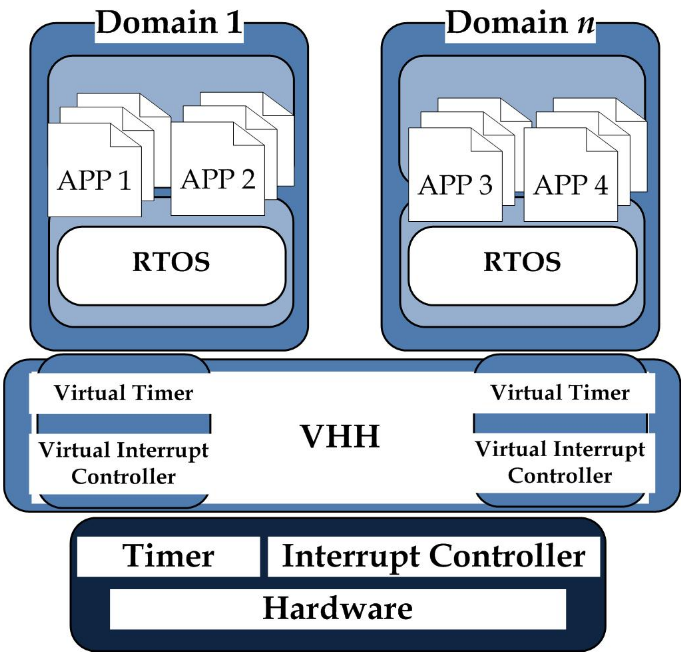

Fig. 10. Virtual-Hellfire Hypervisor Domain structure

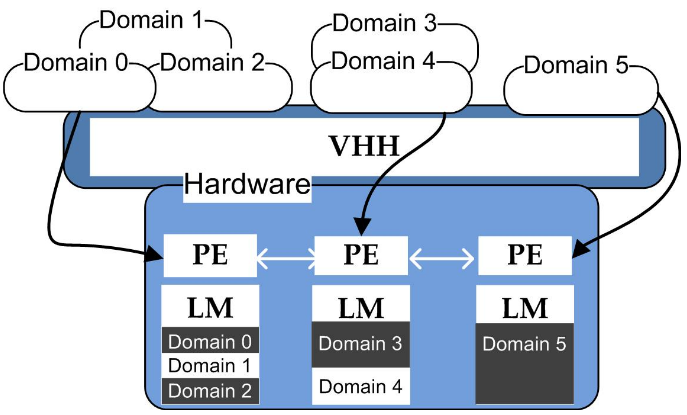

Fig. 11. Virtual-Hellfire Hypervisor Memory Management

We also use this concept, so I/O ports and interrupt lines of peripherals are managed by a specific domain, named *I/O Domain*. In the future, we intend to extend this model, allowing that other partitions also use the peripheral (one at a time). This approach is depicted in Figure 12, where the highlighted domain is responsible for handling I/O issues. Currently, this limits the use of peripherals to the processor that holds the I/O Domain. In the future, when other domains will be able to handle it, any processor will be able to have its own I/O peripheral.

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;

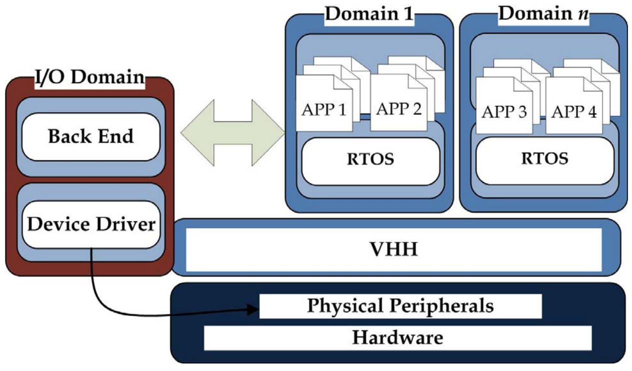

Fig. 12. Virtual-Hellfire Hypervisor I/O Handling

- **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.

The described architecture of VHH is depicted in Figure 13.

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 [28].

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 domain 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.

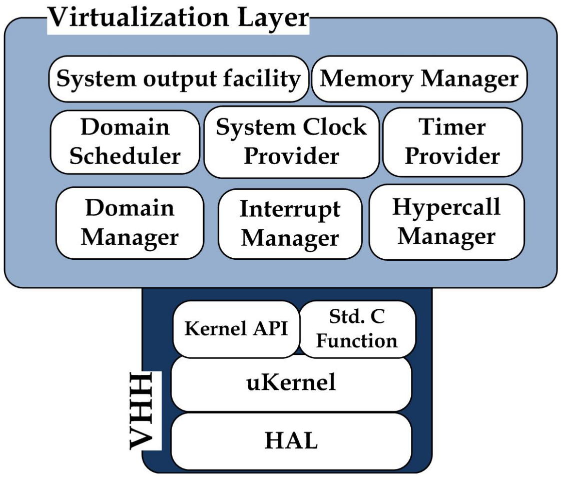

Fig. 13. VHH System Architecture

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 14 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.

VII. Concluding Remarks and Future Work

This paper presented an extension of the HellFire framework in order to incorporate a virtualization methodology. We demonstrated the usefulness of virtualization in embedded systems' design and how it can be applied in current design flows. A new

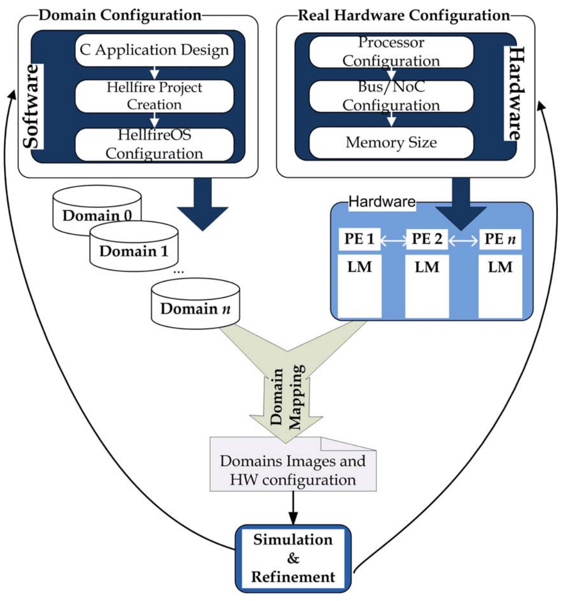

Fig. 14. VHH Integrated in the Hellfire Framework

design flow with virtualization was proposed as an extension of the Hellfire methodology.

As a future work we intend to get 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 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] Tammy Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*, Newnes, 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] Luciano Lavagno and Claudio Passerone, "Design of embedded systems," in *Embedded Systems Handbook*, Richard Zurawski, Ed., chapter 3. CRC press, 2005.

[4] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro, "Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors," in *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, New York, NY, USA, 2004, p. 22, ACM.

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

[6] Grant Martin, "Overview of the mpsoc design challenge," in *DAC '06: Proceedings of the 43rd annual conference on Design automation*, New York, NY, USA, 2006, pp. 274–279, ACM Press.

[7] Wayne Wolf, *Computers as components: principles of embedded computing system design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

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

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

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

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

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

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

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

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

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

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

[18] 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.

[19] Hao Shen and F. Petrot, "Novel task migration framework on configurable heterogeneous mpsoc platforms," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, Jan. 2009, pp. 733–738.

[20] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali, "Supporting task migration in multi-processor systems-on-chip: A feasibility study," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, 2006, pp. 1–6.

[21] V. Nollet, P. Avasare, J-Y. Mignolet, and D. Verkest, "Low cost task migration initiation in a heterogeneous mp-soc," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, Washington, DC, USA, 2005, pp. 252–253, IEEE Computer Society.

[22] D. Kleidermacher and M. Wolf, "Mils virtualization for integrated modular avionics," oct. 2008, pp. 1.C.3–1 –1.C.3–8.

[23] Johan Fornaeus, "Device hypervisors," jun. 2010, pp. 114 –119.

[24] L. Rudolph, "A virtualization infrastructure that supports pervasive computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 8 –13, oct. 2009.

[25] 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.

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

[27] XEN.org, "Xen.org," Web, Available at http://www.xen.org/. Accessed at 10 ago., 2010.

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