Virtualizing Mixed-Criticality Operating Systems

Rodrigo Schmitt Meurer   Mateus Krepsky Ludwich   Antônio Augusto Fröhlich
Software/Hardware Integration Lab (LISHA)
Federal University of Santa Catarina (UFSC)
P.O.Box 476, 880400900 - Florianópolis - SC - Brazil
Email: {rmeurer,mateus,guto}@lisha.ufsc.br

***Abstract*—The forever growing number of embedded control units in some applications such as cars or airplanes are increasing system complexity and making harder to coordinate all this hardware. The increasing capacity of embedded hardware and the advances in virtualization technology make it possible to deal with this problem. There is a current trend to bring many of this control systems to a single platform, thus making it possible for a single hardware platform to manage a whole system with the same isolation guarantees that the use of multiple Microcontroller Units (MCUs) provides, which also implies in a reduction of cost. However, all these different applications have very different purposes and requirements, so the platform should be able to handle all of them, from human interaction to hard real-time control. In our paper, we explore Linux paravirtualization interface, paravirt_ops, to make it run on a confined Virtual Machine (VM) on top of the HyperEPOS real-time hypervisor. The virtualized Linux provides embedded systems with all the functionalities of a general purpose operating system, including human interaction and connectivity, while other VMs define a realm for the proper operation of safe-critical tasks.**

I. Introduction

Constructability and operability of modern embedded systems are becoming more complicated. This increasing complexity directly affect the traditional metrics for developing these systems, such as cost, reliability, real-time operation, security and time-to-market. During the recent years, the performance of embedded systems had a steady growth making it possible to add more functionalities to the systems that in the past were single-purpose. In other words, tasks that previously had to be done by many separated platforms and MCUs now can be joint into one. It is currently possible to take advantage of this potential to make the system's hardware easier to build and manage. Take a car, for example; nowadays a vehicle has numerous embedded systems running in parallel to provide all the functionalities a modern car has. Joining all of them on a single platform would imply in a less sophisticated hardware and therefore a better system reliability. While doing this could create a single-point-of-failure scenario, it can be easily dealt with by creating redundancy, and not only would it be easier but also cheaper. While this is in a very early stage in the automotive industry, there is already development in the avionic industry [1].

In this case, we would have a lot of systems with different purposes, interfaces and requirements. To deal with this variety of systems, we can use platform virtualization techniques. By using virtualization, it is possible to create virtual machines that can either run a real-time operating system able to control time-constrained tasks (e.g. the ABS brake system of the car) or a general purpose to interface with the user or the Internet (e.g. to update the navigation system on the panel).

With the addition of a general purpose operating system to such embedded systems, we can make the manageability of all these joint systems yet easier. Also, it expands the capabilities of such systems by providing an infrastructure to develop any application that somebody may imagine.

In this paper, we present important concepts involving virtualization technologies, embedded systems and how can they benefit from this technology from extending their capabilities with general purpose operating systems. We use the Linux paravirtualization interface (paravirt_ops) to provide HyperEPOS, a real-time hypervisor, with the required infrastructure for integration with the Linux kernel, thus allowing the interoperability of the systems executed in a virtualized manner. Our main contributions are the possibility of parting real-time software from user interface or connectivity applications with the guarantee of no I/O interference provided by HyperEPOS, the increase in software development quality and the reduction of time required to develop applications.

The rest of this paper is organized as follows: Section II provides a quick background on Virtualization, the available techniques, and Linux paravirtualization interface; Section III gives an overview of the related work in the fields of embedded system virtualization and mixed-criticality systems; Section IV describes the main design decisions, the adaptation of the paravirtualization interface and; Lastly, we close our paper in Section V with our key findings and future work.

II. Background

This section provides a quick background on Virtualization, its techniques and Linux paravirtualization interface.

A. *Virtualization*

In the context of this paper, we understand virtualization as platform virtualization. In this case, virtualization is used for hiding the physical machine hardware resources, such as CPU, RAM, and I/O devices and allowing it to host multiple virtual machines.

A virtual machine is the emulation of a particular computer system. Usually, the virtual machines operate based on the computer architecture and functions of a real or hypothetical computer. The implementations may involve specialized hardware, software, or a combination of both. There are two main

techniques to implement virtual machines, which solely depend on the type of the hypervisor that they use.

B. Hypervisors

Hypervisors or Virtual Machine Monitors (VMMs) is the software component that implements the virtualization. The work of Popek et al. states the formal requirements for a virtualizable computer architecture [2]. In their paper, they expect architectures that support memory addressing using relocation register, supervisor and user modes, and trap mechanisms that change the control of the program to a specific point up to the execution of particular instructions.

The authors point out three main types of instructions: i) a Privileged instruction is an instruction that causes a trap and is executed by the hypervisor; ii) a Sensitive instruction if it affects the memory or processor state without going through a trap; iii) an Innocuous instruction is the one that relies on the value of the registers or the processor mode.

They also state tree properties that a hypervisor should be able to achieve: efficiency, resource control, and equivalence. By efficiency, the authors intend that all the innocuous instructions should be executed directly on the physical hardware. The resource control property states that an arbitrary program should not be able to affect the system's resources. Lastly, the equivalence property states that any program executed in a VM should perform in a manner indistinguishable from the one implemented on a native machine.

There are two main types of hypervisors. In a bare-metal or native hypervisor (a.k.a. Type 1), the hypervisor itself can be considered an operating system since it is the piece of software that works in privileged mode. It is also known as hardware level virtualization. This type of hypervisor manages the guest operating systems similarly to the way that an operating system executes its tasks. The most known hypervisors of this kind nowadays are the Oracle VM Server, Xen, Microsoft's HyperV and VMware ESX/ESXi.

The other type of hypervisor (a.k.a Type 2) is called Hosted hypervisor. The hypervisor is an application running on an operating system that can create an environment able to support the execution of another operating system. This type is also referred to as operating system level virtualization. Known hosted hypervisors are Virtual box and QEMU.

C. Paravirtualization

The most traditional type of Hypervisor exposes a hardware functionally identical to the underlying machine. Although this approach has its advantages (e.g. running unmodified operating systems), sometimes it has some drawbacks, especially when there is no hardware support. It is possible to overcome this disadvantage with the cost of reduced performance and increased software complexity. There are more arguments against this virtualization approach. There are situations in which the guest operating systems can benefit from being aware that they are being executed on a host and distinguishing the real from the virtual resources. Providing a guest operating system with both real and virtual time, for example, allows it to provide better support for time-sensitive tasks. This approach was presented in [3] and is named Paravirtualization. It can overcome most of the drawbacks from prior virtualization techniques, but the guest operating system requires modifications.

Barham et al. present Xen, a hypervisor that can provide high performance, support for unmodified application binaries and full multi-application operating systems without the need of sacrificing performance of functionality [4]. This hypervisor, however, requires modifications to the guest operating system source code.

D. Linux and Paravirtualization Support

Linux is a general purpose operating system assembled under the model of free and open-source software development and distribution. At the USENIX conference in 2006 in Boston, Massachusetts, many Linux development vendors (including IBM, VMware, Xen, and Red Hat) collaborated on an alternative form of paravirtualization, initially developed by the Xen group, called "paravirt_ops." The paravirt_ops code (often shortened to pv_ops) was included in the mainline Linux kernel as of the 2.6.23 version and provides a hypervisor-agnostic interface between the hypervisor and guest kernels. This interface made simpler to port Linux to developed hypervisors.

The infrastructure allows you to compile a single kernel binary which will either boot native on bare hardware or boot fully paravirtualized in any of the environments you've enabled in the kernel configuration. It uses various techniques, such as binary patching, to make sure that the performance impact when running on bare hardware is effectively unmeasurable when compared to a non-paravirt_ops kernel. At present, paravirt_ops offers hooks for over one hundred sensitive functions and is available for x86_32, x86_64, and IA64 architectures.

III. Related Work

Even though the virtualization field has been very developed in the recent years, little is being done regarding embedded systems virtualization. The most well-known virtualization solutions, such as Xen [5], KVM [6] and VMware ESXi [7], may support many operating systems, but they do not address some of the issues that embedded systems face. Therefore the choice of the piece of software that virtualizes the system is of foremost importance. Kaiser identifies the main virtualization approaches concerning their applicability to embedded systems, presents possible shortcomings that this approaches may encounter, and proposes methods which could remedy these deficiencies [8].

Bruns et al. provide in their paper [9] an evaluation of the interference imposed by applications without time constraints which run in parallel to a real-time subsystem by comparing the L4 [10] / Fiasco [11] microkernels and FreeRTOS [12]. Their work states that the run-time overhead on the microkernel has almost no effect on the real-time applications. However, the resulting system requires more cache resources to achieve the same level of performance of a non-virtualized operating system.

In the work of Sandström et al., the authors identify the key solutions among the state-of-the-art of server and embedded virtualization. They discuss what are the hardware requirements for the virtualization technologies as well as the level of service provided by each one. Most importantly, the authors provide development directions for supporting virtualized embedded real-time systems [13].

The efforts made to support real-time tasks in the work from Gerum, focus on enhancing Linux's capability of running real-time tasks [14]. This approach is different from ours. In our work we provide a secure real-time hypervisor that can support any Linux in an isolated environment, therefore not interfering with each other, and yet run real-time tasks in other VMs.

The work presented by Aguiar and Hessel, discuss the most suitable ways for employing virtualization in embedded systems and the primary goals that such techniques can achieve [15]. They also point out that bringing together general purpose operating systems and embedded systems can increase the quality of software development since it allows the designer to choose among the available operating systems more suitable for the target application. Besides, the time required to develop an application can be reduced, because applications already available for specific operating systems do not need to be rewritten.

Running Linux on a real-time hypervisor has already been done by Legout et al. [16]. They build a hypervisor inside of the Anaxagoros microkernel that does not provide high performance but yet gives correct responses and use the Linux paravirt_ops to port the Linux kernel to their hypervisor. While the authors were able to port Linux to their hypervisor, their hypervisor does not achieve real-time. In our work, we use HyperEPOS to achieve this.

IV. Linux @ HyperEPOS

Linux is the best-known and most-used open source operating system in the world [17]. It runs on a highly diverse range of computer architectures. From smartphones to cars, super-computers and home appliances, the Linux operating system is everywhere. Linux distributions usually are specialized for different purposes including computer architecture support, embedded systems, stability, security, localization to a particular region or language, targeting of specific user groups, support for real-time applications, or commitment to a given desktop environment. These are the main reasons that make Linux such an interesting choice of general purpose operating system to employ on embedded systems. Furthermore, as of kernel version 2.6.23, Linux comes with a paravirtualization interface. Paravirt_ops provides a uniform interface, that can be used by many distinct hypervisors, thus, standardizing the port of Linux to new Hypervisors.

Since we wish to have real-time capabilities in our system, we chose a bare-metal real-time hypervisor. The main reason for this choice is the fact that it doesn't matter if we are running a real-time hypervisor if the operating system underneath it does not provide us with strict timing policies. Meanwhile, the use of a bare-metal ensures that the timing constraints are met.

A. *HyperEPOS*

Most of the hypervisors that target real-time do not deal with the challenges imposed by modern multicore architectures such as memory and I/O hierarchies. Instead, they either employ a dedicated hardware approach (e.g. CoMik) [18], do not address hard real-time (e.g. OKL4 and uC/OS-MMU) [19], [20], or focus mostly on multicore scheduling issues (e.g. RT Xen) not performing a further investigation on memory neither I/O [21]. HyperEPOS is a real-time hypervisor that takes into account in its design and implementation such architectural aspects, handling memory and I/O hierarchy, interrupt handling, and real-time scheduling.

HyperEPOS virtualizes execution with Virtual CPU (VCPU)s, which are grouped together on a *domain* (i.e. a virtual machine (VM)) where lies a guest OS. Each VCPU is in turn, assigned to a Physical CPU (PCPU) (e.g. a core in a multicore processor). Domains are classified either as critical or as best-effort. Critical ones are meant to run embedded real-time operating systems (RTOS) and must not suffer interference that could disrupt their temporal behavior. Best-effort (BE) domains are expected to run general-purpose operation systems on which human-interaction and multimedia applications will be executed. On HyperEPOS, a VCPU is implemented as a periodic server: it is scheduled to run, and it runs up to the completion of its period, in which the guest OS can execute its tasks. Additionally, the VCPUs themselves are scheduled by the hypervisor according to real-time scheduling algorithms such as PEDF.

In multicore architectures, one core can cause temporal interference on another due to the sharing of the last-level cache (LLC). HyperEPOS deals with such interference by using page coloring, a strategy of memory partitioning where a *color* is given to a set of pages in a way that using pages of a color will not evict cache lines of another color. Therefore enabling time predictability on memory accesses [22], [23]. On HyperEPOS a color is assigned to each VCPU that is intended to be used in a critical domain. Thus critical domains do not suffer nor cause interference on other domains. Furthermore, VCPUs of BE domains can share the same color since it is not critical that they cause temporal interference on each other. Memory management in HyperEPOS is built using the MMU hardware mediator, responsible for allocating physical memory to the system and by *heap* abstractions, responsible for allocating logical memory to the domain tasks. The implementation of page coloring used in HyperEPOS relies on the EPOS MMU hardware mediator family. While employing colors, the MMU provides for multiple lists for free-frames management, one list for each color. Similarly, there are multiple heaps, one for each color, to provide dynamic memory allocation for the domain tasks [24].

To solve the problem of temporal interference caused by I/O operations, HyperEPOS proposes a strategy for monitoring the traffic on shared buses and powering down peripherals being

used by non-critical tasks to mitigate the interference suffered by the PCPUs that are assigned to critical domains. Powering down peripherals is performed in a speculative fashion based on data from the Performance Monitoring Unit (PMU) and also from peripheral's performance registers (e.g. the register that counts the number of bytes sent in a network card). The hypervisor can choose to shut down a BE domain that uses I/O devices whenever the insertion rate related to I/O trespasses a given threshold or based on statistical analysis of peripheral usage. Such evaluation of which domain should be powered down is performed at runtime by the hypervisor, in its idle time (when all VCPUs have completed their work and are waiting for the next period). EPOS power management interface, implemented by the HyperEPOS, allows changing operating modes of individual components, including the ability to turn them on and off [25]. It also keeps track of the relation between system components, ensuring consistency of operating mode transitions.

One of the key ideas to achieve bounded interrupt handling, needed to fulfill real-time requirements, is to decouple interrupt reception and acknowledgment from interrupt servicing. HyperEPOS achieves that by employing interrupt servicing threads (IST)s that are modeled and implemented using the *Concurrent_Observer* design pattern [26]. In such design pattern, a hardware interrupt is handled by a short interrupt servicing routines (ISR) that are limited to receive, acknowledge, and notify in a semaphore the occurrence of the interrupt. An IST waits on the same semaphore for the interrupt to occur and once notified proceeds with the interrupt servicing. On HyperEPOS devices interrupts are routed to a single PCPU whose VCPU belongs to a dedicated domain, named Domain 0, which is responsible for handling I/O interrupts. Once a hardware interrupt occurs, it is received by a short ISR that notifies the semaphore if the *Concurrent_Observer* design pattern. Upon the invocation of a hypercall, a task running on a guest OS of a domain will make the VCPU assigned to it to play the role of the IST of the *Concurrent_Observer* design pattern by handling the interrupt that can implies, for example, on copying data from a buffer inside the hypervisor to a buffer of the guest OS domain. It should be noted that the use of semaphores in Concurrent Observer does not introduce priority inversion problems since they are not being used to guard a critical section. They are used only to synchronize the ISR with the corresponding IST, much in a Producer/Consumer way.

B. Linux and HyperEPOS Integration

Figure 1 shows how we have integrated Linux and HyperEPOS. Our hypervisor interface, called *phepos*, assigns functions to the paravirt_ops hooks (function pointers), implementing paravirt_ops interface. We must observe that not all of these functions must be implemented through a direct hypercall. Some of them just need to have the same behavior. Before describing how each paravirt_ops hooks are implemented by phepos, we describe the kernel configuration used and how does HyperEPOS jumps to the Linux domain (start-of-day).

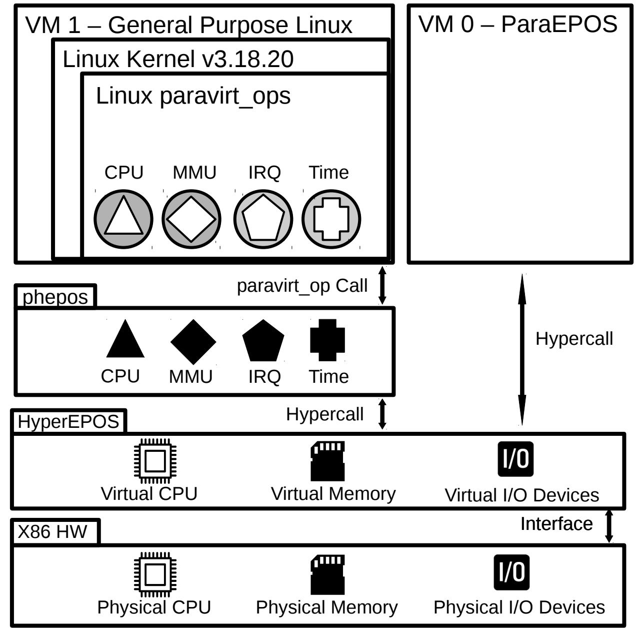

Figure 1. Linux and HyperEPOS integration.

C. Kernel Configuration

In this work, we use the Linux kernel version 3.18.20 with custom settings that are mainly responsible for building a 32-bit kernel without PAE support, enabling paravirt_ops and relocating the image. These settings were generated based on Linux preset configurations. We used *i386_defconfig* to create a default configuration file for the kernel build. From this configuration, we added the *tinyconfig* parameters, which alters the configuration file to create a functional kernel with minimum size and minimal functionalities.

HyperEPOS assumes that all applications begin at the address 0x00000000, and the data segment is separated by a page table (0x400000). Since Linux runs as an application under HyperEPOS, we changed *PHYSICAL_START* (physical address where the kernel is loaded), *PAGE_OFFSET* to the required value to make the ELF compatible with the hypervisor. The Linux linker script was also changed so that the data segment begins at the specified address. Lastly, we enabled the paravirtualization interface. Other than that, we do not make any changes in Linux source code other than the ones necessary to add HyperEPOS as a Host.

D. Start-of-Day

In the usual Linux boot flow after jumping into protected mode, the kernel enables paging and resume the boot reaching *i386_start_kernel* and *start_kernel*. When running paravirtualized, the hypervisor is already running under the operating system, so we must assign function hooks for the sensitive functions that are patched through the paravirtualization interface before we reach their first calls. To do that when we enter into protected mode and before we reach any of these functions, we jump to *phepos_start_kernel*. It is responsible for

the whole hypervisor setup inside the kernel. It mostly assigns hooks for the functions of the paravirt_ops, but also perform some extra functions that the typical setup would (reserving the top of the memory), or the hypervisor requires (indicating where the page structures are).

E. Hypercalls

The guest operating systems running under a hypervisor are not able to execute any privileged instruction. In the same way, that applications use system calls to ask the operating system's kernel to execute a privileged instruction; a virtual machine uses hypercalls to ask the hypervisor. Table I lists the hypercalls used by phepos to implement the paravirt_ops hooks related to virtualization of CPU, MMU and IRQ. In Figure 2 we have an example where the kernel issues a write_cr3 inside the load_cr3 function call. With the kernel configured for paravirtualization, the write_cr3 is implemented through pv_mmu_ops, that has a hook assigned to phepos, which implements this instruction through a hypercall.

For x86_32 targets, paravirt_ops uses the `regparm (3)` GCC calling convention. This places the first three arguments in `%eax`, `%edx`, `%ecx` (in that order), and the remaining arguments are placed on the stack when a function from the interface is called. The hypercalls can take up to 4 arguments, which is the case for the cpuid instruction. Our hypervisor, on the other hand, takes all the function arguments from the stack. To deal with this, we wrap our functions with asm instructions that put the arguments from the registers on the stack issue the function call and place the return values from the stack to the registers again. This is the only overhead that is brought by our hypervisor interface.

In the following sections we describe the main considerations when assigning the adequate function hooks to the paravirt_ops interface.

F. CPU Hooks

These functions are responsible for intercepting a series of CPU privileged instructions and if necessary asking the hypervisor to execute them through a hypercall. Most of the privileged instructions related to x86 CPU make use of the Control Registers (CR).

The control register CR0 keeps several flags that control the basic operation of the processor. We maintain the value of the CR0 register locally. The hypervisor (Host) does not at any moment make changes. Sometimes Linux makes reads in this register, these however do not need to be implemented via a hypercall because that would cause an unnecessary overhead to the host.

The CR4 control register, among other functions is mainly used in protected mode to control operations such as virtual-8086 support, enabling I/O breakpoints, page size extension and machine check exceptions. Since we do not wish to support page size extension or any other of the features that CR4 controls, our write_cr4 function does nothing and the read_cr4 always returns zero.

Table I
TABLE OF HYPERCALLS

Hypercall	Description
hyper_epos_cpuid	Returns processor identification and feature information
hyper_epos_read_cr2	Returns the value of the CR2 register for the current VCPU
hyper_epos_read_cr3	Returns the value of the CR3 register for the current VCPU
hyper_epos_write_cr3	Writes the give value on the CR3 register of the current VCPU
hyper_epos_int_disable	Disable current VCPU interrupts
hyper_epos_int_enable	Enable current VCPU interrupts.
hyper_epos_load_idt_entry	Update shadow IDT
hyper_epos_new_page_table	Creates new page table
hyper_epos_new_page_table_at	Create new page table at given page frame number
hyper_epos_attach_page_table	Attaches the given page table to the page directory of the current VCPU
hyper_epos_detach_page_table	Detaches the given page table to the page directory of the current VCPU
hyper_epos_update_page_table_entry	Update a given page table entry
hyper_epos_flush_tlb	Flushes the TLB of the current VCPU

The kernel issues the cpuid instruction to query the processor for a list of its available features. HyperEPOS implementation for this instruction masks all of the inconvenient features out, so Linux does not try to activate them.

G. MMU Hooks

Most of our hypercalls are related to memory management. To simplify our port, we do not implement PAE. We have only the paging structure of a 32-bit kernel without PAE support. Therefore we only need to deal with two-level paging (PGD and PTE).

The control register CR3 points to the current page directory. The page directory changes whenever there is a change in the current process or domain. Both read and write hypercalls were developed to cope with virtual memory addressing.

The CR2 register contains the address of the last page fault, so there is no need for the guest to write on this register, only read. When the page fault occurs, the hypervisor writes the address on the register.

Four paravirt_ops handle TLB flushes. They are respectively user, kernel, single and others. In our experiment, we implement all of the in the same way.

H. IRQ Hooks

IRQ hooks are pretty simple, they are done directly through hypercalls. The only point that we need to consider is that these functions are assumed to save their own registers if they need to. Normally C fucntions assume that they can trash the `%eax` register. For this reason we use the PV_CALLEE_SAVE_REGS_THUNK macro provided by

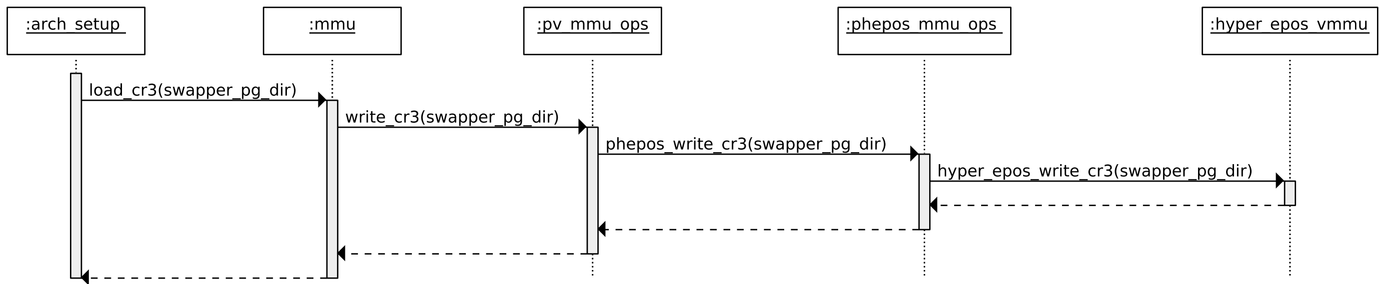

Figure 2. Example of Linux Call sequence using paravirt_ops.

Linux, that creates a wrapper arround the function that saves the register values in the stack.

V. CONCLUSION

Extending the capabilities of an embedded system has demonstrated to be very beneficial in cases such as: improving connectivity, human-computer interaction, security or allowing it to make better usage of its resources. The employment of current virtualization technologies indicates to be an excellent solution to do this, yet we must handle critical aspects like parting real-time software from user interface or connectivity applications with care.

In this paper we proposed a combination of the Linux kernel with a real-time hypervisor, HyperEPOS, to cope with the increasing complexity of distributed embedded control units used in modern systems. We described the most important aspects that must be taken into account when aiming at bringing together these systems, such as the need to execute some instructions through hypercalls and providing the guest operating system with the necessary infrastructure that it expects. Moreover, we explained how the port of Linux to HyperEPOS is done.

In our future work, we plan to test this system in a real hard real-time scenario such as the feed control of an inverted pendulum. In this scenario the real-time operating system will be responsible for taking care of the physical plant control and in another non-real-time virtual machine Linux will display a user interface with the current state of the pendulum. With this, we hope to identify any possible shortcomings of our implementation and focus on optimizations to make this even better.

REFERENCES

[1] M. A. Sánchez-Puebla and J. Carretero, “A new approach for distributed computing in avionics systems,” in *Proceedings of the 1st International Symposium on Information and Communication Technologies*, ser. ISICT ’03. Trinity College Dublin, 2003, pp. 579–584.

[2] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” in *Proceedings of the Fourth ACM Symposium on Operating System Principles*, ser. SOSP ’73. New York, NY, USA: ACM, 1973, pp. 121–.

[3] A. Whitaker, M. Shaw, and S. D. Gribble, “Scale and performance in the denali isolation kernel,” in *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, ser. OSDI ’02. Berkeley, CA, USA: USENIX Association, 2002, pp. 195–209.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

[5] Xen Project, “Xen Project,” "http://www.xenproject.org/", accessed: June 2016.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “Kvm: the linux virtual machine monitor,” in *In Proceedings of the 2007 Ottawa Linux Symposium (OLS’-07)*, 2007.

[7] VMware, “VMware ESXi,” "https://www.vmware.com/products/vsphere-hypervisor", accessed: June 2016.

[8] R. Kaiser, “Complex embedded systems - a case for virtualization,” in *Intelligent solutions in Embedded Systems, 2009 Seventh Workshop on*, June 2009, pp. 135–140.

[9] F. Bruns, S. Traboulsi, D. Szczesny, E. Gonzalez, Y. Xu, and A. Bilgic, “An evaluation of microkernel-based virtualization for embedded real-time systems,” *Proceedings - Euromicro Conference on Real-Time Systems*, pp. 57–65, 2010.

[10] “L4Linux,” "http://os.inf.tu-dresden.de/L4", accessed: June 2016.

[11] “Fiasco,” "http://os.inf.tu-dresden.de/fiasco", accessed: June 2016.

[12] “FreeRTOS,” "http://www.freertos.org", accessed: June 2016.

[13] K. Sandström, A. Vulgarakis, M. Lindgren, and T. Nolte, “Virtualization technologies in embedded real-time systems,” *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2013.

[14] P. Gerum, “Xenomai - Implementing a RTOS emulation framework on GNU/Linux,” 2004.

[15] A. Aguiar and F. Hessel, “Embedded systems’ virtualization: The next challenge?” in *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, June 2010, pp. 1–7.

[16] V. Legout and M. Lemerre, “Paravirtualizing linux in a real-time hypervisor,” *SIGBED Rev.*, vol. 9, no. 2, pp. 33–37, Jun. 2012.

[17] “About Linux,” "http://www.linuxfoundation.org/about/about-linux", accessed: June 2016.

[18] A. Nelson, A. B. Nejad, A. Molnos, M. Koedam, and K. Goossens, “Comik: A predictable and cycle-accurately composable real-time microkernel,” 3001 Leuven, Belgium, Belgium, pp. 222:1–222:4, 2014.

[19] G. Heiser and B. Leslie, “The okl4 microvisor: Convergence point of microkernels and hypervisors,” New York, NY, USA, pp. 19–24, 2010.

[20] M. Beckert, M. Neukirchner, R. Ernst, and S. M. Petters, “Sufficient temporal independence and improved interrupt latencies in a real-time hypervisor,” New York, NY, USA, pp. 86:1–86:6, 2014.

[21] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee, “Real-time multi-core virtual machine scheduling in xen,” New York, NY, USA, pp. 27:1–27:10, 2014.

[22] J. Liedtke, H. Haertig, and M. Hohmuth, “Os-controlled cache predictability for real-time systems,” in *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS ’97)*, ser. RTAS ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 213–.

[23] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi, “Building timing predictable embedded systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, pp. 82:1–82:37, Mar. 2014.

[24] G. Gracioli and A. A. Fröhlich, “An experimental evaluation of the cache partitioning impact on multicore real-time schedulers,” in *19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Taipei, Taiwan, Aug 2013.

[25] A. S. H. Junior, L. F. Wanner, and A. A. Fröhlich, “A Hierarchical Approach For Power Management on Mobile Embedded Systems,” in *5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, Braga, Portugal, Oct. 2006, pp. 265–274.

[26] M. K. Ludwich and A. A. Fröhlich, “Proper Handling of Interrupts in Cyber-Physical Systems,” in *26th IEEE International Symposium on Rapid System Prototyping*, Amsterdam, The Netherlands, Oct. 2015, pp. 83–89.