Proteus Hypervisor: Full Virtualization and Paravirtualization for Multi-core Embedded Systems

Katharina Gilles, Stefan Groesbrink, Daniel Baldin, Timo Kerstan

Proteus Hypervisor: Full Virtualization and Paravirtualization for Multi-Core Embedded Systems

Katharina Gilles[1], Stefan Groesbrink[1], Daniel Baldin[1], and Timo Kerstan[2]

[1] Design of Distributed Embedded Systems, Heinz Nixdorf Institute, Fuerstenallee 11, 33102 Paderborn, Germany
s.groesbrink@upb.de,dbaldin@upb.de, www.hni.uni-paderborn.de/
[2] dSPACE GmbH, Rathenaustrasse 26, 33102 Paderborn, Germany
tkerstan@dspace.de, www.dspace.de

Abstract. System virtualization's integration of multiple software stacks with maintained isolation on multi-core architectures has the potential to meet high functionality and reliability requirements in a resource efficient manner. Paravirtualization is the prevailing approach in the embedded domain. Its applicability is however limited, since not all operating systems can be ported to the paravirtualization application programming interface. Proteus is a multi-core hypervisor for PowerPC-based embedded systems, which supports both full virtualization and paravirtualization without relying on special hardware support. The hypervisor ensures spatial and temporal separation of the guest systems. The evaluation indicates a low memory footprint of 15 kilobytes and the configurability allows for an application-specific inclusion of components. The interrupt latencies and the execution times for hypercall handlers, emulation routines, and virtual machine context switches are analyzed.

1 Introduction & Related Work

System virtualization refers to the division of the hardware resources into multiple execution environments [21]. The hypervisor separates operating system (OS) and hardware in order to share the hardware among multiple OS instances. Each guest runs within a virtual machine (VM)—an isolated duplicate of the real machine (also referred to as partition). The consolidation of multiple systems with maintained separation is well-suited to build a system-of-systems. Independently developed software such as third party components, trusted (and potentially certified) legacy software, and newly developed application-specific software can be combined to implement the required functionality. The reusability of software components is increased, time-to-market and development costs can be reduced, the lifetime of certified software can be extended. The rise of multi-core processors is a major enabler for virtualization. The replacement of multiple hardware units by a single multi-core system has the potential to reduce size, weight, and power [19]. Virtualization's architectural abstraction eases the migration from

single-core to multi-core platforms [11] and supports the creation of an unified software architecture for multiple hardware platforms.

Primary use cases for this technology are security for open systems and OS heterogeneity. First, if a system allows the user to add software, the isolation of potentially faulty or malicious software in a VM ensures against risks for the critical parts of the system. Second, multiple different OSs can be hosted to provide each subsystem a suitable interface. Industrial automation, medical, or mobile systems, for example, require often both a real-time operating system (RTOS) and a general purpose operating system (GPOS) [11]. The deterministic and highly efficient RTOS executes critical tasks such as the control of actuators or the cellular communication of a mobile device. The feature-rich GPOS supports the development of the graphical user interface. The integration of a legacy component may require a third OS.

Since system virtualization gained significant interest in the embedded real-time world, multiple vendors developed multi-core hypervisors for this domain, for example, Wind River's *Embedded Hypervisor*, LynuxWorks' *LynxSecure Hypervisor*, or Green Hills' *Integrity Multivisor*. See [7] for a recently published survey of both commercial and academic real-time virtualization solutions. In the academic world, Xi et al. developed a real-time scheduling framework for the hypervisor Xen [23], which supports PowerPC multi-core architectures. Xen relies on either paravirtualization [2] or on hardware assistance. Xen is not available for PowerPC without an additional hypervisor mode, although this is on the project's roadmap since 2006 [3]. XtratuM by Masmano et al. is a paravirtualization hypervisor implemented on PowerPC [17]. SParK by Ghaisas et al. is a hypervisor for PowerPC platforms without hardware assistance for virtualization [6]. However, their solution requires paravirtualization and does not support multi-core platforms. Closest to our work, Tavares et al. presented an embedded hypervisor for PowerPC 405, which supports full virtualization, but no multi-core architectures [22].

None of these hypervisors provides full virtualization on multi-core PowerPC platforms without hardware assistance. They rely on either paravirtualization or processor virtualization extensions. Examples for processors with hardware assistance for virtualization are Intel VT-x or AMD-V for x86 architectures. Virtualization support was added to the PowerPC architecture with instruction set architecture Power ISA Version 2.06 [9], is however only available for high performance processors. Typical platforms for embedded systems do not feature hardware assistance and many OSs cannot be paravirtualized for legal or technical reasons. By consequence, the applicability of existing PowerPC hypervisors is limited significantly.

We present the first real-time hypervisor for multi-core PowerPC platforms, which features both paravirtualization and full virtualization without relying on explicit hardware assistance for virtualization. Proteus ensures VM separation and is characterized by a bare-metal approach, a symmetric use of the processor cores, and a synchronization mechanism that does not rely on special hardware support. The evaluation shows a low memory and execution time overhead.

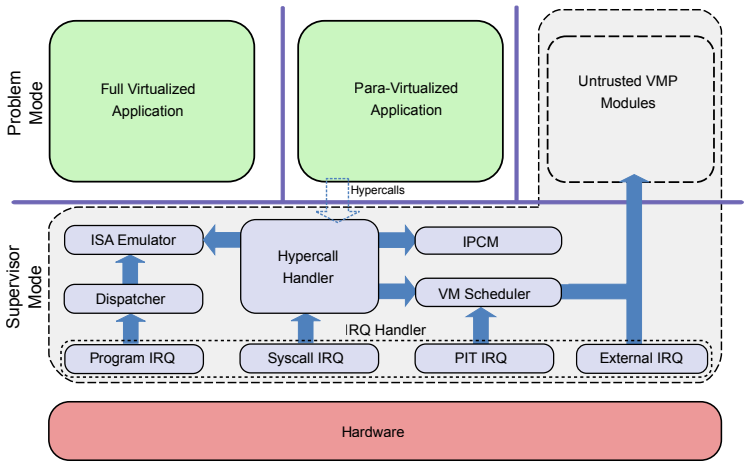

Fig. 1: Design of the Proteus Hypervisor [1]

2 Approach

In previous work, we developed a predecessor with the same name *Proteus*[1], a hypervisor for 32-bit single-core PowerPC architectures. In this work, we present a redesign for multi-core platforms.

2.1 Design

A *hosted* hypervisor runs on top of a host OS [21], which leaves resource management and scheduling at the mercy of this OS. Moreover, the entire system is exposed to the safety and security vulnerabilities of the underlying OS. A *bare-metal* hypervisor runs directly on top of the hardware, facilitating a more efficient virtualization solution. The amount of code executed in privileged mode is smaller compared to a hosted hypervisor, since only a (preferably thin) hypervisor and no OS is incorporated in the trusted computing base. The attack surface is reduced, both the overall security and the certifiability of functional safety are increased. Due to those performance and robustness advantages as well as the clearer and more scalable separation, the bare-metal approach is more appropriate for embedded systems and followed by our design.

The design of the Proteus hypervisor is depicted in Fig. 1. The PowerPC 405 [8] features two execution modes. In the *problem mode* for applications only a subset of the instruction set can be executed. In the more privileged *supervisor mode* for system software, full access to machine state and I/O devices is available via *privileged instructions*. Only the minimal set of components is executed in supervisor mode: interrupt and hypercall handlers, VM scheduler, and inter-partition communication manager (IPCM). All other components such as I/O device drivers are placed inside a separate partition (untrusted VMP modules) and executed in problem mode.

Any occurring interrupt is delegated to the hypervisor. The hypervisor saves the context of the running VM and forwards the interrupt internally to the appropriate component or back to the OS. If the execution of a privileged instruction caused the interrupt, it is forwarded to the dispatcher to identify the corresponding emulation routine. In case of a hypercall, the hypercall handler invokes either the emulator, the inter-partition communication manager, or the VM scheduler. An external interrupt is forwarded to the responsible device driver.

Proteus is a symmetric hypervisor: all cores have the same role and execute guest systems. When the guest traps or calls for a service, the hypervisor takes over control and its own code is executed on that core. Different guests on different cores can perform this context switch from guest to hypervisor at the same time. An alternative design is the *sidecore* approach with one dedicated core to exclusively execute the hypervisor [14]. When an interrupt occurs, the hypervisor on the sidecore handles it and no context switch is invoked. The hypervisor may either be informed via an interprocessor interrupt (not featured by the PowerPC 405) or a notification by the guest OS, which requires paravirtualization. To reconcile sidecore approach and full virtualization, a small fraction of the hypervisor could be executed on each core to forward interrupts. The guest OS could run unmodified, but each exception would involve a context switch and thereby a loss of the major benefit. If the sidecore is already serving the request of a guest, other guests have to wait, resulting in varying interrupt processing time, which is inappropriate for real-time systems. For these reasons, we decided in favor of a symmetric design.

2.2 Multi-core Processor Virtualization

The virtualization of the processing unit is the crucial part of a hypervisor. An instruction is called *sensitive* if it depends on or modifies the configuration of resources. According to a criteria defined by Popek and Goldberg, an instruction set is efficiently virtualizable, if the set of sensitive instructions is a subset of the set of privileged instructions [18]. The PowerPC fulfills this criteria and is fully virtualizable. In contrast for example to the x86 architecture, all sensitive instructions cause an exception (trap), if executed in problem mode.

Solely the hypervisor is executed in supervisor mode and the guests are executed in problem mode with no direct access to the machine state. This limitation of the guests' hardware access is mandatory in order to retain the hypervisor's control over the hardware and guarantee the separation between VMs. The PowerPC 405 does not provide explicit hardware support for virtualization such as an additional hypervisor execution mode. However, guest OSs rely themselves on an execution-mode differentiation. Therefore, the problem mode has to be subdivided into two logical execution modes: VM's privileged mode and VM's problem mode. By virtualizing the machine state register, the hypervisor creates the illusion that a guest OS is executed in supervisor mode, but runs it actually in problem mode. When a guest OS executes a privileged instruction in problem mode (e.g. an access to the machine state register) a trap is caused and the hypervisor executes the responsible emulation routine.

In a multi-core system, access to shared resources must be synchronized. A common solution are semaphores, accessed under mutual exclusion and assigned exclusively to one core at any time. The PowerPC 405 does not feature any hardware support to realize mutual exclusion in a multi-core architecture. Its instructions *lwarx* (load locked) and *stwcx* (store conditional) for atomic memory access do not work across multiple processor cores. Since interrupt disabling is as well not feasible for multi-core systems, Proteus implements a software semaphore solution: Leslie Lamport's Bakery Algorithm [15]. It does not require atomic operations such as test-and-set, satisfies FIFO fairness and excludes starvation, an advantage over Dijkstra's algorithm [4].

2.3 Full Virtualization and Paravirtualization

The capability to host unmodified OSs classifies hypervisors. In terms of *full virtualization*, unmodified guests can be hosted, whereas *paravirtualization* requires a porting of the guest OS to the hypervisor's paravirtualization application programming interface (API) [2]. The guest is aware of being executed within a VM and uses hypercalls to request hypervisor services, what can often be exploited to increase the performance [13]. The major drawback is the need to port an OS, which involves modifications of critical kernel parts. If legal or technical issues preclude this for an OS, it is not possible to host it. A specific advantage of paravirtualization for real-time systems is the possibility to apply dynamic real-time scheduling algorithms, which in general require a passing of scheduling information such as deadlines from guest OS to hypervisor.

Proteus supports both kinds because of those characteristics of the two approaches—paravirtualization's efficiency, but limited applicability on the one hand, full virtualization's support of non-modifiable guests on the other hand. If the modification of an OS is possible, the system designer decides whether the effort of paravirtualization is justified. The concurrent hosting of both paravirtualized and fully virtualized guests is possible without restriction. Proteus is designed for the co-hosting of GPOS and RTOS, and the natural approach is to host a paravirtualized RTOS and a fully virtualized GPOS. In addition, bare-metal applications without underlying OS can be hosted.

Each privileged instruction is associated with an emulating hypercall. Hypercalls are realized as system calls. A system call is identified as a hypercall, if it is executed in the VM's logical privileged mode. A paravirtualized OS can use hypercalls to communicate with other guests, call I/O functionality, pass scheduling information to the hypervisor, or yield the CPU.

2.4 Spatial and Temporal Separation

System virtualization for embedded real-time systems requires the guarantee of spatial and temporal separation of the guest systems. *Spatial separation* refers to the protection of the integrity of the memory space of both the hypervisor and the guests. Any possibility of a harmful activity going beyond the boundaries of a VM has to be eliminated. To achieve this, each VM operates in its

own address space, which is statically mapped to a region of the shared memory. It is protected by the memory management unit (MMU) of the PowerPC 405. Communication between VMs is controlled by the IPCM. If the hypervisor authorizes the communication, it creates a shared-memory tunnel. Communication between VMs is mandatory, if formerly physically distributed systems that have to communicate with each other are consolidated.

Temporal separation is fulfilled, if all guest systems are executed in compliance with their timing requirements. A predictable, deterministic behavior of every single real-time guest has to be guaranteed. The worst-case execution times (WCET) of all routines are bounded and were analyzed (see section 3). These results make it possible to determine the WCET of a program that is executed on top of Proteus. System virtualization implies scheduling decisions on two levels. The hypervisor schedules the VMs and the guest OSs schedule their tasks according to their own scheduling policies. Proteus manages a global VM scheduling queue and each VM can be executed on each core. If this is undesired, a VM can be bound to one specific core or a subset of cores, for example to assign a core exclusively to a safety-critical guest [11]. If the number of VMs n_{guests} exceeds the number of processor cores n_{cores}, at each point in time, $n_{guests} - n_{cores}$ VMs are not executed. The cores have to be shared in a time-division multiplexing manner and the VM scheduling is implemented as a fixed time slice based approach. The guests' task sets have to be analyzed and execution time windows within a repetitive major cycle are assigned to the VMs based on the required utilization and execution frequency. This static scheduling approach is for example applied in the aerospace domain and part of the software specification *ARINC 653 (Avionics Application Standard Software Interface)*[20] for space and time partitioning in avionics real-time systems in the context of *Integrated Modular Avionics* [5]. See [12] for guidance of designing a schedule that allows all guests to meet their timing constraints. The scheduler can be replaced by implementing an interface.

3 Experimental Results

3.1 Evaluation Platform: IBM PowerPC 405

Target architecture of our implementation are platforms with multiple IBM PowerPC 405 cores [8], a 32-bit RISC core providing up to 400 MHz. It is designed for low-cost and low-power embedded systems and features separate instruction and data caches as well as a MMU with a software-managed TLB. Specifications and register-transfer level description are freely available to the research community. Due to the API compatibility within the PowerPC family, porting the results to other PowerPC processors should be fairly simple. In order to be able to evaluate the software with low effort on different hardware configurations, the evaluation platform is a software simulator for PowerPC multi-cores [10]. The *IBM PowerPC Multi-core Instruction Set Simulator* can optionally include peripheral devices (e.g. an UART) and provides an interface for external

simulation environments. Many components of the simulated hardware can be configured, for example, the number of cores or cache sizes.

3.2 Memory Footprint

Dependent on the requirements of the actual system, Proteus can be configured. The workflow is based on the modification of a configuration file by the system designer. According to these specifications, the preprocessor manipulates the implementation files and removes unneeded code.

Figure 2 lists code and data size for the base functionality and the additionally required memory for different components, also depicted in a figure with a differentiation between text segment (executable instructions) and data segment (static variables). The hypervisor is written in C and assembly language. The efficiency of a hypervisor is highly dependent on the execution times of the interrupt handling. For this reason, most of the components called by those handlers and the handlers themselves are written in assembly language. All executables are generated with compiler optimization level 2 (option -O2 for the GNU C compiler), which focuses on the performance of the generated code and not primarily on the code size. The solely full virtualization supporting base requires a total of about 11 kilobytes. The addition of paravirtualization support accounts for less than 1 kilobyte.

The system designer can decide on enabling TLB virtualization (TLB V), device driver support, and inter-partition communication. *Innocuous register file mapping* (IRFM) is a performance boost for paravirtualized guests. By mapping a specific set of privileged registers into VM's memory space, no trap to the hypervisor is required to access these registers. *Previrtualization* (Pre V) is an approach to paravirtualize guests automatically [16]. The source code is analyzed at compile time in order to identify privileged instructions. At load time, the hypervisor replaces privileged instructions by hypercalls. If all features are enabled, the memory requirement of the hypervisor sums up to about 15 kilobytes.

Feature	**Memory Footprint [bytes]**			
	Assembler	C code	Data	Total
Base	2492	5732	2980	11204
ParaV	252	0	148	400
IRFM	292	476	0	768
PreV	0	256	0	256
TLB V	812	264	656	1732
Driver	0	648	12	660
IPCM	0	500	0	500
Total	3848	7876	3796	15520

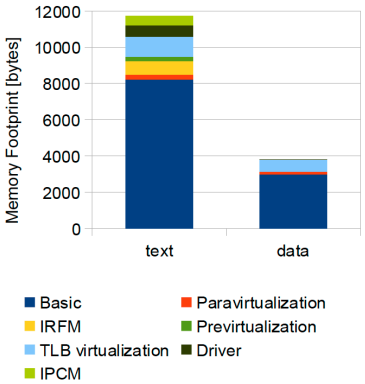

Fig. 2: Impact of Individual Components on Memory Footprint

3.3 Execution Time Overhead

The following performance figures denote the worst-case execution time in case of enabled and hot instruction cache, resulting in a duration for each instruction fetch of one processor cycle, and a clock speed of 300 MHz.

Virtual Machine Context Switch If multiple VMs share a core, switching between them involves the saving of the context of the preempted VM, the selection of the next VM and the resume of this VM, including the restoring of its context. Table 1 lists the execution times for a VM context switch. The overhead of accessing the semaphore that protects the ready queue accounts for a large part of the scheduling execution time.

Table 1: Execution Time of a Virtual Machine Context Switch (4 cores)

Routine	**Execution time in ns (processor cycles)**
VM Context Saving	450 (135)
VM Scheduling	2270 (681)
VM Resume	800 (240)
Total	3520 (1056)

Synchronized Shared Resource Access Routines Figure 3 depicts the execution time of the subroutines of Bakery's Algorithm for synchronized shared resource access (semaphore operations *wait*() and *signal*()). The execution time increases linearly with the number of cores, since the included execution of the function *mutex_start*() has to iterate over an array of length equal to the number of cores. The operation *wait*() causes a blocking of the calling process, if the resource is not available. In case of four cores, the worst case occurs if the calling process is blocked by a process on each of the three other cores, as depicted in Fig. 4. The following formula calculates the worst-case waiting time. *wait_short* (14 cycles), *wait_long* (46 cycles), *signal_short* (15 cycles) and *signal_long* (54 cycles) refer to the shortest and longest paths through the routines wait() and signal(). The critical section is equal to $3 \cdot (wait\_long + mutex\_stop)$, which is the minimum influence of the critical section, since core 1 cannot perform the *signal*() before core 4 completed the try to acquire the semaphore.

$$\begin{aligned} x = 3(&mutex\_start + wait\_short + mutex\_stop + 3(wait\_long + mutex\_stop) \\ &+ mutex\_start + signal\_long + mutex\_stop) \\ &= 3(169 + 14 + 11 + 3(46 + 11) + 169 + 54 + 11) = 1797. \end{aligned}$$

As a result, the worst-case waiting time for synchronized shared resource access sums up to 1797 processor cycles or 5990 ns.

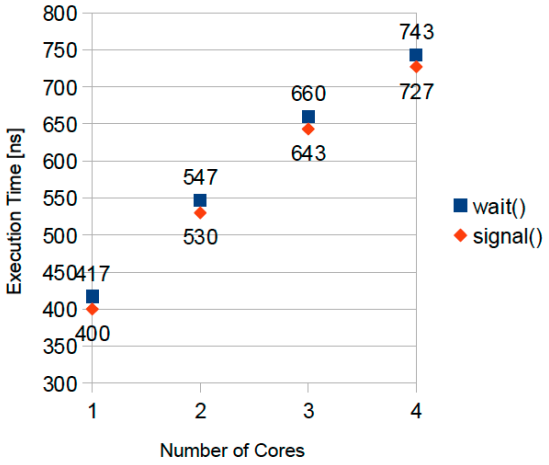

Fig. 3: Linear Dependency of Execution Time of Routines for Exclusive Resource Access on Number of Cores

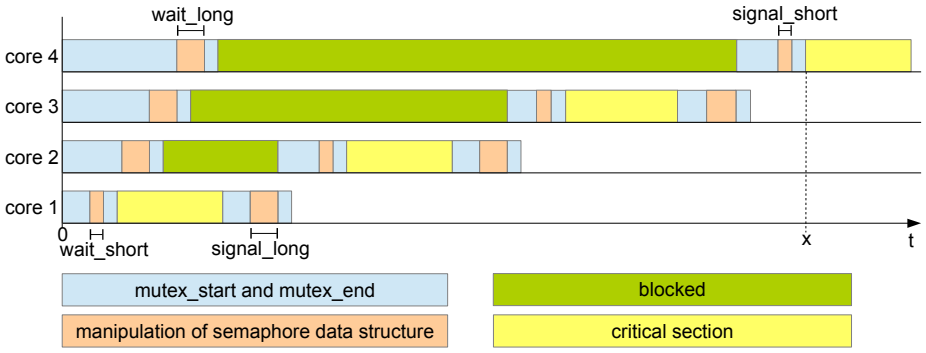

Fig. 4: Worst-case Waiting Time for Synchronized Shared Resource Access for 4 Cores

Interrupt Latency Virtualization increases the interrupt latency. Any interrupt is first delivered to the hypervisor, analyzed and potentially forwarded back to the guest. For example, the additional latency of a programmable timer interrupt is 497 ns (149 processor cycles) and 337 ns (101 processor cycles) for a system call interrupt. To obtain the total interrupt latency, one has to add the interrupt latency of the guest OS. Timer interrupt handling takes longer, since the virtual interrupt timer has to be updated. Proteus omits the effort of saving the complete VM context by saving only the registers that are needed by the emulation routine. The implementation in assembly language uses the fewest possible number of registers.

Emulation of Privileged Instructions The emulation of privileged instructions is the core functionality of the hypervisor. The emulation service is re-

Table 2: Execution Time of Emulation Routines

Privileged instruction	**Execution time in ns (processor cycles)** Full virtualization	Paravirtualization	Speedup
rfi	527 (158)	410 (123)	28 %
wrteei	447 (134)	393 (118)	14 %
mtmsr	517 (155)	363 (109)	42 %
mtevpr	503 (151)	347 (104)	45 %
mtzpr	547 (164)	353 (106)	55 %
mfmsr	453 (136)	363 (109)	25 %
mfevpr	477 (143)	363 (109)	31 %

quested via interrupt (full virtualization) or hypercall (paravirtualization). Table 2 lists the execution times of some exemplary emulation routines. Compared to full virtualization, paravirtualization speeds up the execution by 14% to 55%. The average speedup for all privileged instructions, not just the ones listed in this paper, is 39.25%. An analysis of the steps of an emulation routine helps to understand why paravirtualization can achieve such a significant speedup:

1. Reenabling of the data translation and saving of the contents of those registers that are needed to execute the emulation routine.
2. Analysis of the exception in order to identify the correct emulation subroutine and jump to it (dispatching).
3. Actual emulation of the instruction.
4. Restoring of the register contents.

Table 3 lists the execution time of those steps exemplary for the instruction *mtevpr*. The actual emulation accounts for the smallest fraction. Register saving and restoring are expensive, however likewise for both full virtualization and paravirtualization. The performance gain of paravirtualization is based on the significantly lower overhead for identification of the cause of the exception and dispatching to the correct subroutine. In case of paravirtualization, only a register read-out is necessary in order to obtain the hypercall ID.

Table 3: Execution Time of Emulation Routine for *mtevpr*

Step of emulation routine	**Execution time in ns (processor cycles)** Full virtualization	Paravirtualization
Save registers	137 (41)	137 (41)
Analysis and dispatch	220 (66)	73 (22)
Emulate	20 (6)	20 (6)
Restore registers	127 (38)	117 (35)
Total	503 (151)	347 (104)

Hypercalls A guest OS can request hypervisor services via the paravirtualization interface. The hypercall *vm_yield*, which voluntarily releases the core, has an

execution time of 507 ns (152 processor cycles). By calling *sched_set_param*, the guest OS passes information to the hypervisor's scheduler. The execution time of this hypercall is 793 ns (238 processor cycles). The hypercall *create_comm_tunnel* requests the creation of a shared-memory tunnel for communication between itself and a second VM and is characterized by an execution time of 1027 ns (308 processor cycles). The hypercall *vm_yield* does not return to the VM and the execution time is measured until the start of the hypervisor's *schedule* routine. The other two hypercalls return to the VM and the execution time measurement is stopped when the calling VM resumes its execution.

4 Conclusion

Proteus is a hypervisor for embedded PowerPC multi-core platforms, which is able to host both paravirtualized and fully virtualized guest systems without hardware assistance for virtualization. It is a bare-metal hypervisor, characterized by a symmetric use of the processor cores. The synchronization mechanism for shared resource access does not rely on hardware support. This increases the execution time overhead, but extends the applicability of the hypervisor to shared-memory multiprocessor systems. The hypervisor ensures spatial and temporal separation among the guest systems.

Paravirtualization is due to efficiency advantages the prevailing virtualization approach in the embedded domain. Its applicability is however limited. For legal or technical reasons, not all operating systems can be ported to the paravirtualization interface. Proteus can host such operating systems nevertheless, based on full virtualization's execution of unmodified guests. Paravirtualized operating systems can use hypercalls to communicate with other guests, call I/O functionality, pass scheduling information to the hypervisor, or yield the CPU.

The evaluation highlighted the low memory requirement and the application-specific configurability. The memory footprint is 11 kilobytes for the base functionality and 15 kilobytes for a configuration with all functional features. The interrupt latencies and the execution times for synchronization primitives, hypercall handlers, emulation routines, and virtual machine context switch are all in the range of hundreds of processor cycles. The detailed WCET analysis of all routines make it possible to determine the WCET of a hosted application.

The Proteus Hypervisor is free software released under the GNU General Public License. The source code can be downloaded from https://orcos.cs.uni-paderborn.de/orcos/www .

References

1. Baldin, D., Kerstan, T.: Proteus, a Hybrid Virtualization Platform for Embedded Systems. In: Proc. of the International Embedded Systems Symposium (2009)
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proc. of the 19th ACM Symposium on Operating Systems Principles (2003)

3. Blanchard, H., Xenidis, J.: Xen on PowerPC (Jan 2006), http://www.xen.org/files/xs0106_xen_on_powerpc.pdf
4. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM 8(9) (1965)
5. Garside, R., Pighetti, J.: Integrating modular avionics: A new role emerges. In: IEEE A & E Systems Magazine (2009)
6. Ghaisas, S., Karmakar, G., Shenai, D., Tirodkar, S., Ramamritham, K.: SParK: Safety Partition Kernel for Integrated Real-Time Systems. In: Lecture Notes in Computer Science. vol. 6462, pp. 159–174 (2010)
7. Gu, Z., Zhao, Q.: A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization. Journal of Software Engineering and Applications 5(4), 277–290 (2012)
8. IBM: PowerPC 405 Processor Core. http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores (2005)
9. IBM: PowerPC ISA 2.06 Revision B. https://www.power.org/documentation/power-isa-version-2-06-revision-b/ (July 2010)
10. IBM Research: IBM PowerPC 4XX Instruction Set Simulator (ISS) (Oct 2012), https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_4XX_Instruction_Set_Simulator_(ISS)
11. Intel Corporation (White paper): Applying multi-core and virtualization to industrial and safety-related applications. http://download.intel.com/platforms/applied/indpc/321410.pdf (2009)
12. Kerstan, T., Baldin, D., Groesbrink, S.: Full virtualization of real-time systems by temporal partitioning. In: Proc. of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (2010)
13. King, S., Dunlap, G., Chen, P.: Operating System Support for Virtual Machines. In: Proc. of the USENIX Annual Technical Conference (2003)
14. Kumar, S., Raj, H., Schwan, K., Ganev, I.: Re-architecting VMMs for Multicore Systems: The Sidecore Approach. In: Proc. of the Workshop on Interaction between Operating Systems and Computer Architecture (2007)
15. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM 17, 453–455 (1974)
16. LeVasseur, J., Uhlig, V., Chapman, M., Chubb, P., Leslie, B., Heiser, G.: Pre-virtualization: Soft Layering for Virtual Machines. In: Proc. of the 13th Asia-Pacific Computer Systems Architecture Conference (2008)
17. Masmano, M., Ripoll, I., Crespo, A.: XtratuM: a Hypervisor for Safety Critical Embedded Systems. In: Proc. of the Eleventh Real-Time Linux Workshop (2009)
18. Popek, G.J., Goldberg, R.P.: Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM 17(7), 412–421 (1974)
19. Prisaznuk, P.: Integrated Modular Avionics. In: Proc. of the IEEE National Aerospace and Electronics Conference (1992)
20. Prisaznuk, P.: ARINC 653 Role in Integrated Modular Avionics (IMA). In: Proc. of the 27th IEEEE Digital Avionics Systems Conference (2008)
21. Smith, J.E., Nair, R.: The Architecture of Virtual Machines. IEEE Computer (2005)
22. Tavares, A., Carvalho, A., Rodrigues, P., Garcia, P., Gomes, T., Cabral, J., Cardoso, P., Montenegro, S., Ekpanyapong, M.: A Customizable and ARINC 653 Quasi-compliant Hypervisor. In: Proc. of the IEEE International Conference on Industrial Technology (2012)
23. Xi, S., Wilson, J., Lu, C., Gill, C.: RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In: Proc. of the International Conference on Embedded Software (2011)