Increasing the Reusability of Embedded Real-time Software by a Standardized Interface for Paravirtualization

Stefan Groesbrink · Heinz Nixdorf Institute, University of Paderborn · s.groesbrink@upb.de

Applying System Virtualization to Reuse Software. Hypervisor-based virtualization refers to the division of the resources of a computer system into multiple execution environments in order to share the hardware. Multiple existing software stacks of operating system and applications such as third party components, trusted legacy software, and newly developed application-specific software can be combined in isolated virtual machines to implement the required functionality as a system of systems. Virtualization is a promising software architecture to meet the high functional requirements of complex embedded and cyber-physical systems. The consolidation of software stacks leads in many cases to reduced bill of material costs, size, weight, and power consumption compared to multiple hardware units.

This work focuses on the increase of reusability of embedded real-time software by a standardized interface between hypervisor and operating system. Virtualization offers in this regard the following benefits:

- Migration to Multi-core: The integration of multiple single-core software stacks is a way to migrate to multi-core platforms. The required effort is significantly lower compared to a multi-core redesign or the parallelization of sequential programs.
- Operating System Heterogeneity: The granularity of virtualization facilitates to provide an adequate operating system for the subsystems' differing demands, for example a deterministic real-time operating system for safety-critical control tasks and a feature-rich general purpose operating system for the human-machine interface. This enables the integration of legacy software incl. the required operating system without having to port the application software.
- Cross-Platform Portability: By applying emulation techniques, virtualization enables the execution of software that was developed for a different hardware platform without a redesign. A hypervisor might emulate an I/O device, a memory module, or even the CPU (different instruction set architecture) transparent to the guest systems. This is valuable if hardware is no longer available or if legacy software and new software shall be combined on state-of-the-art hardware.

The increased software reusability reduces development time and costs, plus, extends the lifetime of software. It protects investments and makes a company less vulnerable to unforeseen technological change. This is in particular valuable for embedded real-time systems that require a time-consuming and cost-intensive certification of functional safety, such as transportation or medical systems.

Virtualization of Real-time Systems. The hypervisor-based integration of independently developed and validated real-time systems implies scheduling decisions on two levels (hierarchical scheduling). On the first level, if the number of virtual machines exceeds the number of cores, the hypervisor schedules the virtual machines. On the second level, the hosted guest operating systems schedule application tasks according to their specific local scheduling policies. An appropriate hierarchical scheduling provides temporal partitioning among guest systems, so that multiple independently developed systems can be integrated without violating their temporal properties.

Paravirtualization: Both a Blessing and a Curse. One distinguishes between two kinds of virtualization based on the virtualization awareness of the guest system. Full virtualization is transparent to the operating system. In case of paravirtualization, the operating system is aware of being virtualized and has to be ported to the hypervisor's paravirtualization application binary interface (ABI) [1]. It uses hypercalls to communicate directly with the hypervisor and access services provided by it, analogous to systems calls of an application to the operating system.

Paravirtualization is the prevailing approach in the embedded domain [2]. The need to modify the guest operating system is outweighed by the advantages in terms of easier I/O device sharing, efficiency (reduced overhead), and in terms of run-time flexibility of an explicit communication and the hereby facilitated cooperation of hypervisor and guest operating system. The major drawback is the need to port an operating system, which involves modifications of critical kernel parts. If legal or technical issues preclude this for an operating system, it is not possible to host it. For both kinds of virtualization, the applications executed by the operating system do not have to be modified.

It is an important observation that many hierarchical real-time scheduling approaches require paravirtualization, since an explicit communication of scheduling information is needed [3]. The operating system has to provide the hypervisor a certain level of insight in order to support its scheduling. The hypervisor might in turn inform the operating systems about its decisions. Moreover, an explicit communication between hypervisor and guest operating system is mandatory for the implementation of any dynamic or adaptive scheduling policy in order to inform the

hypervisor about dynamic parameters or adaptation triggering events such as task mode changes. Finally, instead of running the idle task, a paravirtualized operating system can yield to the hypervisor, which then can execute another virtual machine.

On a side note, hardware-assisted virtualization (e.g. Intel VT-x or AMD-V) introduced processor assists to trap and execute certain instructions on behalf of guest operating systems, increased the efficiency of x86 virtualization, and removed the need for paravirtualization on x86 platforms. However, it does not help to reconcile virtualization and real-time.

Proposal: A Standardized Paravirtualization Application Binary Interface. The drawbacks of paravirtualization are the limited applicability (some operating systems cannot be ported for technical or legal issues) and the porting effort. These issues are in fact the major obstacle regarding the goal to increase the reusability of software by virtualization. A standard for the interface between hypervisor and operating system could imply the following benefits:

- Paravirtualize Once: An operating system has to be paravirtualized specifically for a particular hypervisor. A standardized interface would make the porting independent from specific characteristics of a hypervisor.
- Let Paravirtualize: The existence of a standard would increase the pressure on operating system suppliers to provide an implementation of the interface with the result that the customer does not have to touch the operating system himself.

To the best of our knowledge, a standardized paravirtualization interface for real-time systems was not yet proposed. VMWare proposed an interface for non-real-time systems on the x86 architecture [4], but did not follow up due to the entering of hardware assistance. Our proposal includes hypercalls for hierarchical scheduling, communication between guest systems (if systems that have to communicate with each other are consolidated), and access to I/O devices:

sched_yield: notifies the hypervisor about idling (optional parameter denotes for how long, if known)

sched_pass_param: pass scheduling parameter, e.g. applied task scheduler, mode change, task deadlines

ipc_create_tunnel: creates a shared memory tunnel to another virtual machine

io_send_mac: sets MAC address of the I/O device

io_send_packets: signals hypervisor that ring contains packets to send

The virtualization of I/O devices assumes access via memory mapped I/O and packet-oriented communication. The hypervisor provides a ring buffer in which the guests place their packets. This proposal has to be extended for other kinds of I/O devices. The hypervisor might pass information to the operating system via shared memory communication. A memory region within the guest's memory space is dedicated to paravirtualization communication, accessible by the hypervisor but not by any other guest. A library provides read and write methods for the guest's access to the shared memory.

The major goal of this proposal is minimality, since a small interface reduces the effort to both port an operating system and add the interface to a hypervisor. In contrast to the server and desktop domain with its dominance of the x86 architecture, a specific challenge for the embedded domain is the support of different processor architectures (at least x86, ARM, PowerPC, and MIPS). The ABI has to include architecture-specific parts. Problematic for virtualization are instruction sets with instructions that are sensitive (i.e. attempt to change the processor's configuration or whose behavior depends on the configuration), but not privileged (i.e. do not trap if executed in user mode). Since only the hypervisor can be executed in supervisor mode, these instructions can affect the execution of the other guests and eliminate the isolation. To cope with this issue, our proposal demands a hypercall for each such instruction. Finally, the ABI defines for each processor architecture registers for the passing of the hypercall ID and hypercall parameters.

An implementation of the interface with our hypervisor Proteus and our real-time operating system Orcos on PowerPC 405 (@300 MHz) showed an additional memory footprint of 400 bytes for the hypervisor (to a total of 15 KB) and hypercall execution times between 0.5us and 1us. Hypercalls speed up the execution of privileged instructions in average by 39%, due to the significantly lower overhead for dispatching to the correct subroutine.

In future work, we plan to evaluate the proposal by analyzing the effort of porting both popular open source hypervisors (e.g. RT-Xen, Linux KVM) and real-time operating systems (e.g. FreeRTOS, uC/OS, Linux), as well as investigating whether popular hierarchical scheduling techniques can be realized with the offered hypercalls.

Literatur

[1] Paul Barham et al., *Xen and the Art of Virtualization.* In: Proc. Symposium on Operating Systems Principles, 2003.

[2] Z. Gu and Q. Zhao, *A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization.* In: Journal of Software Engineering and Applications, 4(5), pp. 277–290, 2012.

[3] Jan Kiszka, *Towards Linux as a Real-Time Hypervisor.* In: Proc. Real Time Linux Workshop, 2009.

[4] Z. Amsden et al., *VMI: An Interface for Paravirtualization* In: Proc. Ottawa Linux Symposium, 2006.