A Novel Software Architecture for Mixed Criticality Systems

Ralf Ramsauer[1], Jan Kiszka[2], and Wolfgang Mauerer[1,2(✉)]

[1] Technical University of Applied Sciences Regensburg, Regensburg, Germany
ralf.ramsauer@oth-regensburg.de,
wolfgang.mauerer@othr.de
[2] Siemens AG, Corporate Research, Munich, Germany

Abstract. The advent of multi-core CPUs in nearly all embedded markets has prompted an architectural trend towards combining safety critical and uncritical software on single hardware units. We present a novel architecture for mixed criticality systems based on Linux that allows us to consolidate critical and uncritical parts onto a single hardware unit. CPU virtualisation extensions enable strict and static partitioning of hardware by direct assignment of resources, which allows us to boot additional operating systems or bare metal applications running aside Linux. The hypervisor *Jailhouse* is at the core of the architecture and ensures that the resulting domains may serve workloads of different criticality and can not interfere in an unintended way. This retains Linux's feature-richness in uncritical parts, while frugal safety and real-time critical applications execute in isolated domains. Architectural simplicity is a central aspect of our approach and a precondition for reliable implementability and successful certification. While standard virtualisation extensions provided by current hardware seem to suffice for a straight forward implementation of our approach, there are a number of further limitations that need to be worked around. This paper discusses the arising issues, and evaluates the suitability of our approach for real-world safety and real-time critical scenarios.

Keywords: Mixed criticality · Raltime · Virtualisation · Hypervisor · Linux

1 Introduction

Software for safety-critical systems requires strict certification, and uncritical parts must not interfere with critical ones. Reliability of the software is crucial while the amount of software, measured in Lines of Code (LoC) is a limiting factor for certification processes.

Obtaining a functional safety certification for a kernel like Linux that contains millions of lines of code is obviously a challenging enterprise OSADL Project: SIL2LinuxMP (OSADL 2014), yet product vendors do not want to miss the capabilities of Linux in mixed-criticality systems. We present a novel architectural approach

S. Keil et al. (Eds.): EADTC 2018/2019, LNEE 670, pp. 121–128, 2020.
https://doi.org/10.1007/978-3-030-48602-0_11

that satisfies both goals, safety for critical parts and feature-richness for uncritical parts: Jailhouse[1], a Linux-based partitioning hypervisor.

Jailhouse transforms symmetric multiprocessing (SMP) systems to asymmetric multiprocessing (AMP) systems by inserting virtual barriers to the system and I/O bus. From a hardware point of view, the system bus is still shared, while software is allowed to only access resources within its scope.

Jailhouse is enabled from a standard Linux running on bare-metal hardware (cf. Fig. 1). It takes control over all hardware resources described in a system configuration file, reassigns them back to Linux and lifts Linux in the state of a virtual machine (VM). The hypervisor core of Jailhouse acts as Virtual Machine Monitor (VMM). Jailhouse does not fit into the usual classification of hypervisors Formal requirements for virtualizable third generation architectures (Goldberg 1973), it can be seen as a mixture of Type-1 and Type-2 hypervisors: It is a bare-metal hypervisor that runs on raw hardware without an underlying system level, but requires Linux to initialise hardware before it takes global control over the whole system.

Unlike other real-time partitioning approaches like XtratuM Partitioned Embedded Architecture based on Hypervisor: The XtratuM approach (Crespo et al. 2010) or PikeOS Evolution of the PikeOS microkernel (Kaiser and Wagner 2007) that aim to manage hardware resources and hence forbid direct access, Jailhouse only supports direct hardware access. Instead of using complex and time-consuming (para-)virtualisation Xen and the Art of Virtualization schemes (Braham et al. 2003) for emulation of device drivers, Jailhouse uses virtualisation extensions only for isolation purposes and does neither provide a scheduler nor virtual CPUs. It is a signalbox for direct routing of hardware devices to isolated domains, called »cells«. Only resources that are essential for a hardware platform and that cannot be partitioned in hardware are virtualised.

For creating new isolated domains, Jailhouse removes hardware resources[2] from Linux (also called the root cell) and reassigns them to isolated domains, called non-root cells. Virtualisation extensions ARM Architecture Reference ManualSecure Virtual Machine Architecture Reference ManualIntel virtualization technology (ARM 2013; Uhlig et al. 2005; AMD 2005) guarantee strict isolation: any access violation, for instance prohibited access to certain memory areas, wake up (trap Formal requirements for virtualizable third generation architectures) the hypervisor (Popek and Goldberg 1974), which eventually stops execution. Certain instruction executed by the guest cause traps and must be handled by the hypervisor.

Since Jailhouse remaps and reassigns resources, the hypervisor will not get active after setting up and starting all cells under ideal conditions. The following circumstances require hypervisor intervention:

- Cell management (e.g., create, start, stop or destroy cells)
- Access violations (memory, I/O ports)
- Interception of non hardware virtualisable resources (e.g., parts of the ARM Generic Interrupt Controller)
- Trapping on certain CPU instructions (e.g., x86 *cpuid*)

[1] Available at https://github.com/siemens/jailhouseunderGPLv2.

[2] E.g. CPU(s), memory, (PCI) devices, …

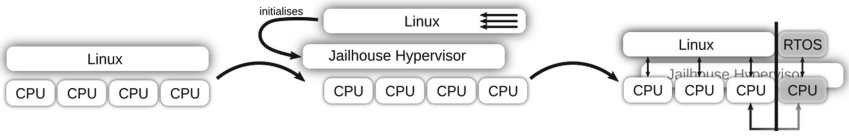

Fig. 1. Activation sequence of the Jailhouse hypervisor. After Linux has placed and started the hypervisor, an additional real-time operating system is started in an isolated critical domain.

On common bare-metal hypervisors, interrupts are dispatched by the hypervisor and reinjected into the guest. On Intel x86, we make use of Interrupt Remapping support and directly map hardware interrupts to cells without trapping the hypervisor: interrupts arrive directly in the assigned cell. This results in lower interrupt arrival times and interrupt latencies, which is beneficial for appliances with hard real-time requirements.

In this way, a safety-certified (minimalist) operating system or bare-metal application can run on a single multi-core system in parallel to Linux. The minimalist approach of Jailhouse results in only a few thousands lines of code for the core parts, which simplifies any certification process.

The rest of this paper is structured as follows: First, we present the hardware partitioning techniques of Jailhouse. We implement a multicopter demonstration platform and run critical parts software (the flight stack) in a jailhouse cell. We give architectural overview of our platform and brief introduction to our hardware setup. Afterwards, present obstacles that appeared during the implementation on real hardware, and present possible solutions.

2 Architecture

To activate the hypervisor (cf. Fig. 2), Linux must be booted with a predefined amount of reserved memory for the hypervisor and for additional non-root cells. After loading the hypervisor binary to its destination inside this memory area, the hypervisor startup code is entered by each CPU and the VMM is initialised.

After the hypervisor is initialised, non-root cells can be created. A non-root cell consists at least of one CPU and a certain amount of memory that can be preloaded by the root cell with a secondary operating system. Linux shuts down selected CPUs and calls the Hypervisor to create a new cell by providing a cell configuration. The VMM creates this new isolated domain by removing resources from the root cell and reassigning them to the newly created domain. Other resources like PCI devices, memory-mapped devices or I/O ports, can be exclusively assigned to a cell. After the cell has been started, it can reject any further tries on modifying its state. This ensures inadvertent modifications of critical domains.

Generally, Jailhouse allows guests to share physical pages with the root cell. Besides enabling inter cell communication, the mechanism also allows for sharing memory-mapped I/O pages, which, if desired, allows us to access hardware resources

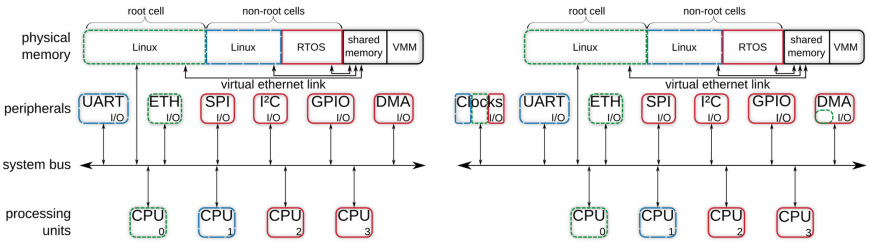

Fig. 2. Ideal vs. real hardware partitioning: Under ideal conditions (left), devices can exclusively be mapped to a cell. In reality (right), functionalities of some peripheral devices may be required in multiple domains or overlap. While the system bus is still shared, the Jailhouse hypervisor takes care that cells will only access resources within their scope. Safe communication between critical and uncritical domains is enabled by shared memory.

from within multiple domains. Such concurrent access is, however, not arbitrated by Jailhouse and needs to be addressed appropriately by the guests[3].

Figure 2 shows a possible partitioned system layout for three cells: the Linux root-cell, an additional Linux non-root cell and a bare-metal real-time operating system. As mentioned before, communication between cells is realised by memory regions that are shared between two cells, together with a signaling interface. This ensures a minimal code footprint. Jailhouse does not emulate any driver functionality, but device drivers may, for instance, use these means to establish a virtual high-performance ethernet connection between two cells. Depending on hardware support, signaling is implemented based on a virtual PCI device through Message-Signaled Interrupts (MSI-X) or legacy interrupts. On systems without PCI support, Jailhouse emulates a generic and simple PCI host controller.

Jailhouse currently supports 64-bit x86 (Intel and AMD), ARMv7, and ARMv8 architectures. Several operating systems were already successfully ported to run as Jailhouse guests. Linux can act as a Jailhouse guest on all supported architectures, the root file system is provided in memory as initial ramdisk. Let us remark that we successfully ported the RTEMS real-time operating system for the ARM architecture with limited efforts. Additionally, a port of FreeRTOS already exists[4].

3 Jailhouse Multicopter Platform

To prove the suitability of Jailhouse for industrial use cases, we implemented a fully functioning multicopter platform for demonstration purposes. We chose this platform as its requirements are similar to industrial appliances as they arise, for instance, in semiconductor manufacturing or when collaborative tasks between machines and humans need to be performed: The flight stack, a highly reliable and safety-critical part of the system, is responsible for balancing and navigating the aircraft. Sensor values

[3] This technique is mainly used for debugging purposes.

[4] https://github.com/siemens/freertos-cell.

must be sampled at high data rates, processed, and eventually be used to control rotors. The control loop is governed by different flight modes, such as a manual mode, stabilised mode or automatic modes like position hold. For a safe and reliable mission, the control loop must respond deterministically. System crashes may result in real crashes with severe consequences.

This obviously requires a real-time capable operating system. We ported the whole critical flight stack to a Jailhouse cell, while uncritical tasks still benefit from the Linux ecosystem and will not interfere with the flight stack in an unacceptable way. Remaining cells can serve any uncritical payload, such as communication with the ground station or camera tracking.

In the critical domain, a second tailored and minimalist Linux operating system with the PREEMPT_RT Internals of the RT Patch real-time kernel extension is executed. As flight stack, we chose the Ardupilot project. No modifications (besides board support and missing hardware drivers) are required. This underlines that existing applications can be deployed in a Jailhouse setup with little effort.

For controlling a multicopter platform, several sensors and actuators are connected to different inter-board buses and peripherals: gyroscopes, compasses, GPS, RC-control receiver and motor control form the controlling circuit. This requires access to SPI, I^2C, UART and GPIO hardware devices from the critical cell. A simplified architectural overview of the partitioned system is shown in Fig. 2.

As hardware platform, we chose an NVIDIA Jetson TK1[5] with a quad-core Cortex-A15 ARMv7 CPU with virtualisation extensions. The TK1 is connected to an Emlid Navio2[6] sensor shield. The system is divided into two parts: two cores are assigned to the uncritical part, the other two to the critical one.

We remove resources that are required for controlling the platform from the root cell and reassign them to the critical domain. The flight stack always controls the machine, even if uncritical cells misbehave. A crash in an uncritical cell does not cause a crash of the critical appliance. The functioning of this architecture is a solid testament to the suitability of Jailhouse for implementing real-time safety-critical systems that are based to a large extent on existing components.

4 Requirements on Partitioning Hardware

Despite the real-world practicability of our approach, we discovered limitations that are caused by hardware design. While every of these limitations can be worked around in software, the issues should be addressed by hardware manufacturers in future to provide optimal base components for mixed-criticality systems. Every workaround results in extra functionality in the hypervisor code, which contravenes the original goal of a most reduced minimal footprint, and also leads to slower response times. Such interception are, of course, contrary to the envisioned partitioning concept.

[5] http://elinux.org/Jetson_TK1.

[6] https://docs.emlid.com/navio2/.

4.1 Memory-Mapped I/O

Peripheral devices are usually accessed by reading from or writing to dedicated physical memory addresses. Those addresses are backed by the registers of the particular device. The typical page size of almost all modern architectures is 4 KiB or more, and represents the finest granularity of memory that can be assigned to a cell without the need for trapping and dispatching access.

While 32 or more bits for physical addresses provide enough space to place different devices on different pages, hardware manufacturers often place multiple devices on one single page, even different types of devices.

This is problematic for hardware partitioning, since only entire memory pages can be assigned to a cell without the need for trapping and dispatching on memory access. Jailhouse implements subpaging, a technique where the hypervisor allows for mapping memory areas to guests that are smaller than the page size. When subpaging is enabled for a certain memory area, Jailhouse will trap on any access to that page and either permit access or crash the cell because of access violations. This leads to noticeable and undesired slow-downs.

4.2 Indivisible Hardware Resources

Placing different devices on different physical memory pages is not always sufficient for hardware partitioning: functionality of a single device might be needed in two cells. Typical devices that are required in multiple cells are DMA controllers, system clock and reset controllers, or GPIO devices. Jailhouse supports sharing of physical memory pages, but it does neither moderate access nor understand the underlying hardware access protocol: Jailhouse will not ensure that parameters are not overwritten by other cells.

Most devices provide full functionality without DMA transfers. In real-time contexts, where I/O response time matters more than I/O throughput, DMA controllers should either be exclusively assigned to a single cell or should not be used if possible. Shared DMA access from different cells requires partitionable DMA controllers.

GPIO devices should exclusively be assigned to a single cell as well. As long as they are not partitionable, accesses have to be dispatched by the hypervisor.

Clock and Reset controllers allow for gating and ungating of device clocks, to select a particular clock source, and to select a prescaler for the clock. They also allow for setting and clearing reset lines of devices. Such clock and reset controllers are usually organised as a single hardware device that controls all available devices of a system. An uncritical cell that has access to the clock and reset controller can therefore deactivate or reset resources that are assigned to a critical cell, and influence the behaviour of the whole system.

One software based solution is to gate and initialise all devices, and then prohibit any further access to the clock and reset controller. While this solution would actually be straight forward, many existing drivers make the assumption that a clock and reset controller is always present and (de-)assert resets during runtime. Other device drivers, like SPI, UART or I^2C driver need to change their speed or baud rate during runtime, which requires them to access the clock and reset controller as well.

As long as clock and reset registers of all devices are bound to a single common clock and reset controller device, it is not possible to partition them without paravirtualisation or dispatching in the hypervisor. This solution is efficient for practical purposes, since clock and reset controllers are usually accessed very rarely compared to regular accesses of a device. The disadvantage is the variety of clock and reset controllers and their different protocols.

Even if shared access to clock and reset controllers is admissible, existing clock driver code is usually not prepared to run on partitioned hardware: available resources are often hard encoded in driver code, and clock drivers often reset or disable all existing system clocks during startup.

4.3 Erroneous Hardware Behaviour

Hardware misbehaves. During the implementation of our demonstration platform, we observed that accessing registers of devices with ungated clocks causes an immediate freeze of the whole system. This misbehaviour occurs in all Tegra-based platforms up to tegra186[7] and is caused by flaws in the hardware design. This problem can be fixed in software by trapping on affected memory areas when their clock gets ungated to guarantee the stability of the rest of the system.

5 Conclusion

Partitioning hypervisor techniques are promising and can be used in mixed-criticality scenarios. By using standard operating systems, we minimised the effort that is required for porting existing legacy payload applications. A minimalist hypervisor core simplifies certification efforts.

We successfully demonstrated the usability of hardware partitioning. However, hardware manufacturers need to change design aspects with respect to the demand that the hardware can be partitioned. Any software-based workarounds lead to more preventable hypervisor code and more hypervisor logic.

This demand requires software engineers and hardware manufacturers to strengthen their focus on Hardware-Software Co-design, in particular when it comes to building mixed-criticality systems that will gain increasing importance in many manufacturing domains.

References

OSADL: Open Source Automation Development Lab, OSADL Project: SIL2LinuxMP (2014)

Popek, G.J., Goldberg, R.P.: Formal requirements for virtualizable third generation architectures. Harvard University, Cambridge (1974)

Crespo, A., Ripoll, I., Masmano, M.: Partitioned embedded architecture based on hypervisor: the XtratuM approach (2010)

[7] http://www.mail-archive.com/jailhouse-dev@googlegroups.com/msg01522.html.

Kaiser, R., Wagner, S.: Evolution of the PikeOS microkernel (2007)
Barham, P., Dragovic, B., Fraser, K. et al.: Xen and the art of virtualization (2003)
ARM: ARM Architecture Reference Manual (2013)
AMD: Secure virtual machine architecture reference manual (2005)
Uhlig, R., Neiger, G., Rodgers, D., et al.: Intel virtualization technology (2005)
Rostedt, S., Hart, D.V.: Internals of the RT patch (2007)