Running Linux and AUTOSAR side by side[*]

Tillmann Nett, Jörn Schneider
Trier University of Applied Sciences
Schneidershof
Trier, Germany
{T.Nett, J.Schneider}@Hochschule-Trier.de

ABSTRACT

Mixed criticality systems are systems, on which safety-critical and non safety-critical software must run simultaneously. For such a system it is necessary that all deadlines of safety critical jobs can be met, and no safety-critical function is impaired by any other function. Current approaches for designing such a system include virtualization, hardware partitioning or implementing all software as critical software. These approaches however introduce additional costs due to additional hardware, more complicated development techniques for non-critical software or loss of processing power due to the virtualization layer.

We demonstrate a novel method for implementing a system, that provides lean interfaces for real-time software and a full Unix interface for non real-time software. This system uses vertical partitioning to run two different operating systems on two cores of a single ARM multiprocessor.

1. INTRODUCTION

Real time operating systems such as AUTOSAR [1] allow provably fast responses to physical events. This is required for example in cars, where the speed of the wheels has to be continuously monitored to take action if the brakes lock. As a failure of such systems often can harm lives, such operating systems need to be simple and easy to verify for correct functional and temporal behaviour.

Interactive operating systems provide a good average case performance thus delivering fast responses to user request in most cases. However, they are not designed for predictability and the worst-case reaction times can be very high.

In a growing class of systems users interaction, e. g. through a GUI (Graphical User Interface), and guaranteed real-time behavior of safety relevant functions has to be available simultaneously. Such systems can be considered as a kind of mixed-criticality systems. This means that such a system must at the same time fulfill requirements for two types of systems, which cannot easily be provided by a single operating system.

A viable but impractical method for building such a hybrid system is to implement all parts to the same standards according to the highest level of criticality on top of a common real-time operating system. This means all interactive parts, including driver software and the complete user interface would have to be implemented as real-time tasks. This introduces additional costs during development and whenever parts of the system are changed.

In some cases it may also be possible to change an existing interactive operating system to provide real-time facilities. For Linux various patches add additional schedulers, synchronization primitives etc. [11]. However the changes to the Linux kernel are often quite large as for example in case of the PREEMPT_RT patch. Also it is often unclear whether these patches offer full hard real-time guarantees or only soft real-time guarantees. Furthermore all kernel code would have to satisfy the relevant safety standards such as [10] and be included in worst-case execution time analyses.

Another method to build such systems is to put different parts of the system on different hardware each with a single operating system, and then connecting them via a bus or network. This method however requires doubling the amount of hardware build into the automobile and introduces latency because of the communication via the network. Furthermore this method increases the power consumption of the system.

A third design method is to add a separate partitioning layer underneath the real-time and the interactive layer following the principle of [4]. This layer separates the real-time system from the interactive system by assigning individual slots in a major time cycle and separating memory as well as forcing all communication over a so-called system partition. However, the enforced indirection has several disadvantages as described in [12].

We present a novel method for designing a mixed criticality system, running two different operating systems (vertical partitioning) on two Cortex-A9 cores. A similar method has previously been used to run two differently configured Linux kernels on two separate cores of an x86 System [8]. Other vertical partitioning setups first start the Linux system and then set aside a separate core for real-time work using core isolation methods provided by Linux [5]. This however greatly increases the startup time of the real time portions as the complete Linux system has to be up and running before the real-time core can be isolated.

The implementation method described here was used for

[*]This work is partly funded by the German Federal Ministry of Economics and Technology under grant number 01ME12043 within the econnect Germany project.

In: Proc. of the 7th Junior Researcher Workshop on Real-Time Computing (JRWRTC). Sophia Antipolis, France, 2013.

implementing an interactive monitoring and control system in electric cars for a field trial on user acceptance of smart grid technology in the project econnect-Trier.

The rest of this paper is organized as follows: Section 2 shortly summarizes the requirements, which governed our design and implementation process. In section 3 we give a detailed overview of the implementation of our prototype, including startup of both operating systems, and the static resource sharing scheme. In section 4 we summarize our work and show how future work can extend our implementation.

2. REQUIREMENTS

The main driving force for this research were the conflicting requirements presented in Section 1. However at the same time other non-functional requirements also had to be taken into account. We will now present the detailed requirements and their justifications.

1. It shall be possible to derive and prove hard real-time bounds for tasks where needed.
2. It shall be possible to implement non real-time or soft real-time parts using operating system abstractions well known by desktop programmers.
3. It shall be easily possible to argue and verify that the real-time tasks are not impaired by the non real-time functions.
4. It shall be possible to add, change or remove non safety-related software without additional analysis or verification of the real-time software.
5. It shall be possible to derive a fixed deadline for the boot time, i.e. the real-time parts of the system must be responsive after a fixed time when the system is started.

Requirements 1, 2, 3, and 4 have been justified in Section 1.

Requirement 5 has been defined, as in cyber physical systems often rigorous constraints are imposed considering answer time of systems. For example safety critical systems in cars often have to answer to messages on the CAN-bus within 100ms after system startup. This often requires large optimizations of the boot code when implementing such devices [2]. Our method on the other hand will optimize the response latency after boot by first starting the real-time operating systems and then starting the non real-time operating system on a separate core.

3. SYSTEM CONFIGURATION

In a uniform memory architecture two or more separate processor cores operate on the same shared memory. All cores can access the same hardware and can receive the same interrupts. For the mixed-criticality system a OMAP4460 multiprocessor [9] was used, which provides two cortex-a9 cores [6], two cortex-m3 cores as well as other specialized cores. Only the two cortex-a9 cores were used and all other cores were deactivated, to keep the system uniform.

3.1 Startup Sequence

During startup of a OMAP4460 the Cortex-A9 *Core0* is initialized first by the ROM code. The ROM code fetches the boot code from non volatile storage and starts executing it. At this point, the system runs in single core mode and can perform initialization steps and load the operating system. At the same time the *Core1* is put into an idle state by the ROM code, from which it can be awakened by the operating system. Once the operating system is ready it can configure the start address of *Core1* and send a special signal which awakes the other core. At this point the system runs in dual core mode.

Hardware	Assigned to
Screen	Linux
Touch-pad	Linux
non-volatile memory (SD-Card)	Linux
UMTS Module	Linux
USB Subsystem	Linux
L3 OCM_RAM (SRAM)	AUTOSAR
GPS-Module	AUTOSAR
CAN-Module	AUTOSAR
SPI-Interface	AUTOSAR
UART-Interface	AUTOSAR
L3 and L4 Interconnects	AUTOSAR/Linux
Main Memory (DRAM)	AUTOSAR/Linux
Interrupt Distributor	AUTOSAR/Linux
Interrupt CPU Interfaces	One per Operating System
Hardware Spinlocks	AUTOSAR/Linux

Table 1: Hardware distribution among operating Systems

The mixed criticality system uses *Das U-Boot* [7] as a bootloader to load an AUTOSAR conforming operating system from non-volatile storage. This AUTOSAR image currently also includes a complete linux kernel statically linked in a separate section of the executable. The real-time system used is an open source version of the ArcticCore AUTOSAR implementation [3]. This Version was implemented for a single core machines, hence the second core is not started by default. We added additional startup code which configures and starts the other core. On the second core a short startup routine is used to load the Linux image contained in the AUTOSAR image and transfer control to the Linux image. All startup parameters needed by linux are provided by the core1 startup routine and are written to the correct locations in memory. These parameters also include the `maxcpus=1` option, which instructs the Linux kernel to run in single core mode.

Das u-boot currently uses different code, depending on the operating systems to be started. For Linux this code also deactivates all interrupts, flushes caches and turns of caches and the MMU. This code is not performed when starting AUTOSAR from a ELF-image, but is required to later start Linux from within AUTOSAR. To put the system into a state that allows Linux to be booted, a call to this code was manually added to the ELF-startup code of das u-boot. As ArcticCore does not use caches or the MMU, it was not impaired by these configuration changes. Interrupts are later initialized again by code which was added to ArcticCore in a way that interrupt lines could be statically assigned to one of the Operating systems.

3.2 Hardware Assignment

To simplify the design of the system and to avoid the need for resource management protocols, which may impair real-time functions, we decided to statically assign most hardware to one of the two operating system. However some Hardware was needed by both operating Systems. For this hardware is is necessary that initializations done by

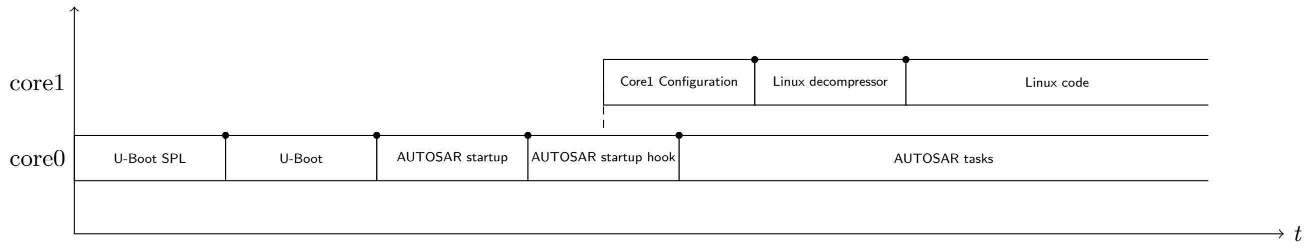

Figure 1: Startup sequence of the mixed system

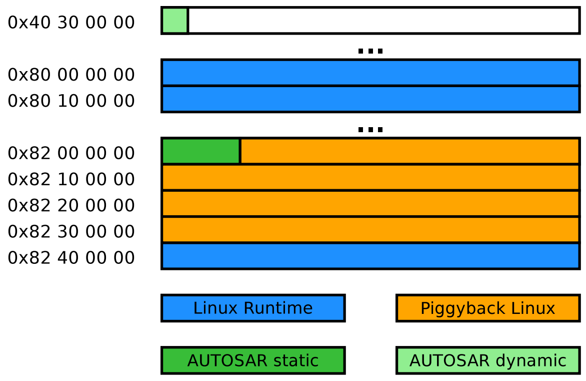

Figure 2: Memory configuration of the final mixed system

AUTOSAR are not be overwritten during booting of Linux. Hence for all shared hardware which is normally initialized by Linux, the initialization routines were deactivated and if necessary the initialization was performed during startup of AUTOSAR. One exception to this rule was the main memory. Main memory is initialized even prior to the start of the OS in das u-boot, hence this initialization could be kept in place. However to ensure that the address space reserved for AUTOSAR would not be used by Linux, we manually deactivated this address space in the Linux kernel using the `mem=` boot parameters. Like all other boot parameters, these were dynamically written by the *Core1* startup code within AUTOSAR, based on the actual address space used by AUTOSAR.

To ensure timing correctness for memory access the MMU was disabled on the core running AUTOSAR. For OMAP4460 SOCs disabling the MMU also disables all caches for memory used by that core. To provide the real time parts with a fast memory the L3 OCM_RAM (SRAM) was used for data parts of the system. The L3 and L4 Interconnects used for communication between subsystems as well as memory of the OMAP4460 was shared between subsystems. A closer analysis of the arbitration strategies for the Interconnects to ensure that all deadlines can be met is still required.

3.3 Interrupt Distribution

Both Cortex-A9 Cores in the OMAP4460 share a single generic interrupt controller (GIC) [1]. This GIC is responsible for global masking of specific interrupts and distributing interrupts to the cores. The GIC is divided into a global distributor and one CPU-interface per core. The distributor can distribute interrupts to one or more CPU-interfaces which then signal the interrupt to the CPU. Interrupts can then be acknowledged on the CPU-Interface. Interrupts can both be masked in the CPU-interface by setting a flag, as well as in the distributor by setting an empty target list.

Neither the application context of the described system, nor the chosen assignment of hardware resources require to signal interrupts to more than one core. At startup all interrupts are configured by AUTOSAR with an empty target list. The target is then set to the appropriate core when activating the interrupt in AUTOSAR or Linux. Reconfiguration of the target within the distributor requires setting a single bit in a memory mapped register. As other bits in the same register may be owned by another operating system, a lock had to be added around any configuration code. For this Lock one of the hardware spinlocks provided by the OMAP4460 SOC was used. Hardware spinlocks use memory mapped registers for operation, hence providing a fast way for synchronization between cores. This lock is only taken during short sections within the Linux kernel. Before requesting this lock we disable preemption in Linux. Because preemption is disabled, only a single kernel-thread can wait for the lock at any time. This means, the time that any AUTOSAR task must wait for the lock is bounded and a worst-case execution time analysis for AUTOSAR tasks remains possible, by taking this additional lock contention time into account. In case Linux is run on multiple cores however the lock could be requested by multiple kernel threads simultaneously. In this case it is possible for one of the threads to steal the lock from the waiting AUTOSAR task and the locking time becomes unbound. In this case a different synchronization primitive is necessary to ensure bounded execution times within AUTOSAR. For example a mixed-criticality lock [12] could be used. Within AUTOSAR all interrupts are configured during startup of the system, so that lock contention times only influence the startup and schedulability is not impacted by the lock.

3.4 Communication

To share data between AUTOSAR and Linux we added real-time communication facilities. For stream transmissions a non-blocking ring buffer implementation is used. Multiple ring buffers are provided for multiple data streams. Using these ring-buffers AUTOSAR tasks can send data packets to Linux, which are then received in a kernel thread. Multiple tasks sending data simultaneously through the same ring buffer must be synchronized with each other. For this the real-time resource sharing mechanisms within AUTOSAR are used.

For communication from Linux to AUTOSAR only the transmission of single word signals was needed. Hence these words were reserved within a shared memory space. These memory words are written in Linux using an atomic store operation and read within AUTOSAR using an atomic read operation.

All data structures needed for communication are set up by AUTOSAR within the address space used by AUTOSAR. The addresses of these data structures are then transmitted to Linux prior to startup using a boot parameter. These data structures are then read from a Linux kernel module which configures all kernel data structures and offers a device file for communication with user space. The provided communication facilities match the requirements imposed by the current project, i.e. single word transmission from Linux to AUTOSAR and stream transmission from AUTOSAR to Linux.

3.5 Modifications to the Linux Kernel

One of the goals of this setup was to keep the modification of the Linux source code to a minimum. Hence, additional functions were implemented in a kernel module, where possible. This module can be compiled into the kernel or loaded at runtime. Loading of the module has no impact on the timing behaviour of the real-time parts, because no shared resources are needed. Some additional modifications were still needed to allow a separation of the operating systems.

The Linaro Linux Kernel for an OMAP4460 SOC contains a complete hardware description of all submodules on the chip. This description is read during startup and all hardware modules are automatically initialized by the kernel. No dynamic configuration of the SOC hardware is performed. This initialization also includes all hardware modules which were shared or statically assigned to AUTOSAR. As this additional initialization would undo all previous initializations done by AUTOSAR, we had to reduce the hardware initialization. For this it was sufficient to remove any module used by AUTOSAR from the hardware description tables provided in the Linux source tree.

During it's initialization routine Linux on the OMAP4460 also configures the GIC and sets all interrupts to target the first core used by Linux. As the GIC is already set up by AUTOSAR this additional initialization code was removed. Instead the interrupt targets are set when the interrupt is activated within Linux. The code for setting the target upon activation was also added.

In the SRAM management code it was completely sufficent to set the SRAM size to zero. The current Linux kernel already includes code paths to skip all SRAM initializations in case there is no SRAM on the current system and to reject all allocations on the SRAM.

As these changes are quite small and only include parts of the kernel which change rarely, it is easily possible to re-apply the same patches to newer kernel versions, making updates of the Linux portions simple.

4. CONCLUSIONS AND FUTURE WORK

We were able to show that it is possible to use separate cores for different operating systems on multi-core processors. While previous work [8] only showed such a setup for x86 systems running two invocations of the same Linux system, we extended this work to the ARM-Platform where one of the operating systems was a real-time operating system. The real-time ability was not impaired by the simultaneously running non-real-time system.

Future work may include a better resource sharing among both systems. As it is rare for both systems to require the same hardware at the same time, it may be possible to provide a suitable protocol that enables the necessary resource-sharing between systems. However such a protocol would have to provide guaranteed latency for the real-time system, but not for Linux system, so an asymmetric protocol could be used. This could for example be implemented using Mixed Criticality Locks [12], which may be adapted for multicore architectures.

Also a detailed comparison and experimental evaluation of the different methods for implementing mixed criticality systems is still needed. At this point however such a comparison would be beyond the scope of this article.

References

[1] *ARM Generic Interrupt Controller - Architecture Specification.* ARM. 2011.

[2] Jan Altenberg. *Fastboot Technologie für Linux.* Tech. rep. Mülhofen, Germany: linutronix, 2010.

[3] *Arctic Core - the open source AUTOSAR embedded platform.* ARCCORE AB, 2011. URL: `http://www.arccore.com/products/arctic-core/`.

[4] *Avionics Application Software Standard Interface.* ARINC Report 653. Aeronautical Radio Inc., Jan. 1997.

[5] Michael Christofferson. *4 Ways to Improve Linux Performance for Multicore Devices.* IEEE Spectrum Online Tech Insider Webinar. 2013.

[6] *Cortex?-A9 MPCore - Technical Reference Manual.* ARM. 2009.

[7] Wolfgang Denk. *Das U-Boot.* 2012. URL: `http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=summary`.

[8] Adhiraj Joshi et al. "Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system". In: *Proc. of the Ottawa Linux Symposium.* Ottawa, Canada, 2010, pp. 101–107.

[9] *OMAP4460 Multimedia Device Silicon Revision 1.x. Technical Reference Manual.* Version F. Texas Instruments. 2011.

[10] *Road vehicles – Functional safety.* ISO/FDIS 26262. International Organization for Standardization, Dec. 2010.

[11] Steven Rostedt and Darren V Hart. "Internals of the RT Patch". In: *Proc. of the Ottawa Linux Symposium.* Vol. Two. Ottawa, Canada, 2007, pp. 161–171.

[12] Jörn Schneider. "Overcoming the Interoperability Barrier in Mixed-Criticality Systems". In: *19th ISPE International Conference on Concurrent Engineering - CE2012.* Trier, Germany, 2012.