Towards certification of software-intensive mixed-critical systems in automotive industry

Peter Reichenpfader, Mario Driussi, Florian Pölzlbauer

Towards certification of software-intensive mixed-critical systems in automotive industry

Peter Reichenpfader
Virtual Vehicle Research Center
Graz, Austria
peter.reichenpfader@v2c2.at

Mario Driussi
Virtual Vehicle Research Center
Graz, Austria
mario.driussi@v2c2.at

Florian Pölzlbauer
Virtual Vehicle Research Center
Graz, Austria
florian.poelzlbauer@v2c2.at

***Abstract*—Next-generation cars will provide a multitude of advanced driver assistance systems (ADAS) functionality up to *full autonomous driving* support [1]. These functions require high computational power and continuous software-maintenance (e.g. over-the-air update). Hence powerful general-purpose multi-core hardware and dynamic software-platforms/OS are being introduced to automotive systems. Furthermore these systems need to be certified according to safety standards. In this paper we highlight several challenges associated with the certification process. We thereby mainly focus on multi-core hardware architecture and RTOS aspects. One main question is whether Linux (with the PREEMPT_RT patch) can be utilized and certified accordingly.**

***Keywords*-certification; software-intensive; mixed-critical; automotive; Linux; PREEMPT_RT; real-time;**

I. Introduction

The EU has set the goal to halve the overall number of road deaths within the European Union by 2020 [2]. In order to achieve this goal, driver assistance systems (ADAS) become ever more important. This leads to a rising number of sensors, actuators and electronic components, whereupon a huge amount of data must be processed in real-time to respond to hazards in traffic. As a consequence these systems must operate as safely and reliably as reasonably possible. Driver assistance systems must not lead to uncontrollable conditions which can no longer be handled by the driver. Such a view on safety has consequences: Devices must be designed not only sufficient fail-safe, they also must return to a safe state when an incident occurs. Therefore, these factors have a strong influence on the entire development process of the system.

II. State of the Art & Research Question

In order to ensure an appropriately safe development of safety-critical products, the ISO 26262 standard [3] was introduced into the automotive industry. It is derived from the generic standard IEC 61508 [4]. Ultimately, the objective of both standards is to determine a system safety integrity level (ASIL or SIL accordingly), which must be taken into account in subsequent development steps for system-, hardware- and software-development. However, a closer look at future ADAS reveals a set of challenges:

- Hardware components, which are currently used in the automotive industry, meet the recommendations of the safety-standard. However, they have insufficient computing-power for real-time environment-detection, and they are still associated with high costs.
- Commonly used hardware platforms from the consumer sector (e.g. ARM, Intel , Nvidia, ...) would provide sufficient computing-power at significant low cost. However, these hardware-platforms are not approved for use in safety-critical systems, since they were designed for different development goals. They offer no safety measures, which in turn raises the question of reliability.
- The development-methods of the safety-standards require a static software system, where all the states of the software are already known at design-time. However, future applications of ADAS make extensive demands on the underlying software platform (e.g. RTOS) and thus introduce "dynamics":
  - Dynamic memory allocation
  - Software-update without garage visit
  - Software-security (e.g.: authentication, authorization, secure boot, ...)
- Many software-platforms which are used in automotive environments meet these requirements only insufficiently or not at all. This contrasts with software-platforms from the consumer sector (e.g. Linux) which already meet all these requirements, but were not designed for safety-critical applications.

All these challenges raise two research questions:

1) Can Linux be certified for safety critical applications?
2) If so, is Linux a cost-effective alternative, compared to adapting existing and certified software-platforms?

III. SIL2LinuxMP Project

The research project `SIL2LinuxMP` [5] has set itself the goal of developing a reference process that allows certification of software-intensive safety-critical systems. On the one hand, it involves the development process (which leads to the *evidence for certification*). On the other hand, the basic components (bootloader, root filesystem, kernel, c library bindings to access the Linux kernel) of Linux shall be certified for SIL 2. Figure 1 provides an overview.

VIRTUAL VEHICLE supports the project by providing an automotive use-case, which serves 3 purposes: (1) providing realistic application-software; (2) acting as source of safety-related technical requirements for RTOS and hardware; (3) evaluating effectiveness of reference development process;

IV. Use-Case: E-QUAD

Within the use-case, two software-applications will be implemented. The first application is a `steer by wire` appli-

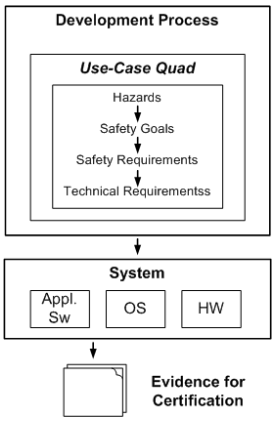

Fig. 1. Overview of the SIL2LinuxMP Project

cation, which needs to satisfy SIL 2. The second application is an `object detection` ADAS application, which can be developed according to QM[1].

A fully electric test vehicle (E-QUAD) will be retrofitted with the necessary sensors (e.g.: angle-sensor, camera, ...), actuators (i.e. steering motor), as well as a corresponding computing-hardware. Figure 2 shows the schematic structure of the use-case.

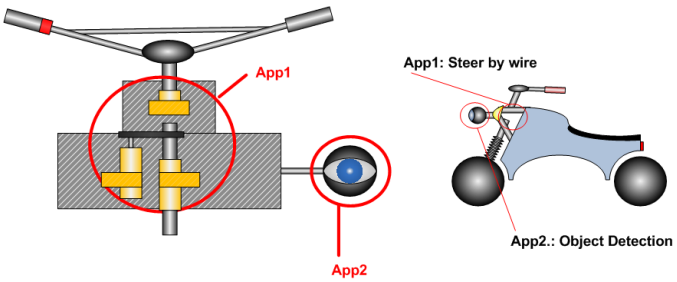

Fig. 2. Use Case E-QUAD

Application	Description
Steer by Wire (SIL 2)	Measuring of current steering-angle; Closed-loop control of steering via stepper-motor according to desired steering-angle; This function could later on be used for future applications (e.g. automatic parking, autonomous driving). The function will not require much computational-power.
Object Detection (QM)	Sampling of environment based on camera; Object-detection within camera-picture based upon OpenCV library [6]; In case that an object is detected, output collision-warning (e.g. light-bulb or acoustic); This application will be computational-intensive and requires dynamic resources. Hence it can provide two benefits: it is representative of future, computational-intensive applications, and it will stress the system in a realistic manner.

TABLE I
DESCRIPTION OF USE CASE APPLICATIONS

[1]The term QM (Quality Management) is used to state that a system or component does not have any safety-related requirements, and thus does not require to comply to a safety-standard.

The system integrates a safety critical application (steer by wire) and a non-safety-critical yet computationally intensive application (object detection) on a multi-core system. The object-detection will be camera-based, and will only be used to output a warning-signal. It will not control steering or breaking. Hence it can be developed according to QM. Figure 3 shows an overview of the applications and the underlying computing-platform.

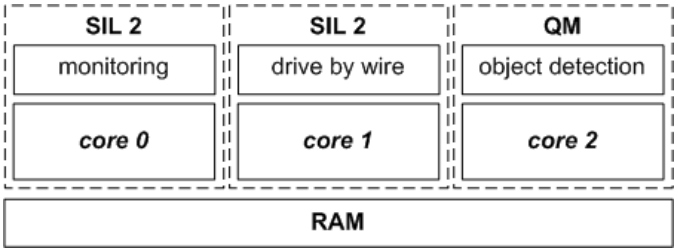

Fig. 3. Computing-Platform and Applications

V. TECHNICAL CHALLENGES

Current safety-standards (such as ISO 26262 or IEC 61508) require `freedom from interference` or `independence of execution` between software-components. This means that a non-critical software-component must not interfere onto a critical software-component (or at least the interference must be within acceptable bounds). One of our technical goals is to investigate if freedom from interference can be achieved for systems which are based upon Linux with the PREEMPT_RT patch which are executed on a general-purpose multi-core platform. If so, which configuration must be used? If not, which parts of Linux and/or the PREEMPT_RT patch would need to be modified?

The term interference relates to a scenario where the execution of a low-critical software has a negative impact onto a higher-critical software. This applies to normal operation, but especially to error cases. Interference can act upon two dimensions: time and space (i.e. memory)

In order to address the issue of interference, the system (SW-platform and/or OS) must provide effective protection-mechanisms at runtime. Here, the AUTOSAR standard [7] provides a set of timing-protection mechanisms (such as execution-time monitoring, inter-arrival rate monitoring, ...) as well as memory-protection mechanisms. Hence the questions arise: Does Linux provide similar protection mechanisms? If so, are they sufficient in order to satisfy the requirements of the safety-standards? If not, which aspects of the mechanisms have to be improved (or implemented)?

A. Timing-Protection Mechanism

One key timing-protection mechanism of AUTOSAR (version 4.x) is execution-time monitoring (ETM). If activated, it monitors the execution-time of a task. In case the task's execution-time exceeds a pre-defined value (i.e. the execution-time budget) the OS stops the task. This monitoring method is especially useful, since estimating the worst-case execution time (WCET) of a task is one of the hardest challenges in

computer science. This is especially true for modern (multi-core) computing-platforms. Hence execution-time monitoring provides a powerful tool for ensuring safe task execution.

Figure 4 shows a gantt-chart which demonstrates how execution-time monitoring can mitigate interference due to execution-time overruns. We see a system which contains 4 tasks (T1, T2, T3, T4). At the top we see an error case without execution-time monitoring: After 3ms, an error occurs inside task T3, which causes the task to execute for additional 2ms. During this time, T4 cannot execute due to pre-emption by T3. Once T3 finishes, T1 and T2 execute according to their priorities. Finally, T4 can execute. However, due to the prolonged pre-emption, T4 misses its deadline.

In the bottom view we see how the same error scenario performs, if T3 is subject to execution-time monitoring. Again, T3 is prolonged due to an error. However, after 4ms, its execution-time budget is consumed, and the OS stops T3. This leads to T4 being able to start executing. After a short pre-emption by T1 and T2, task T4 can resume its execution and finish before its deadline.

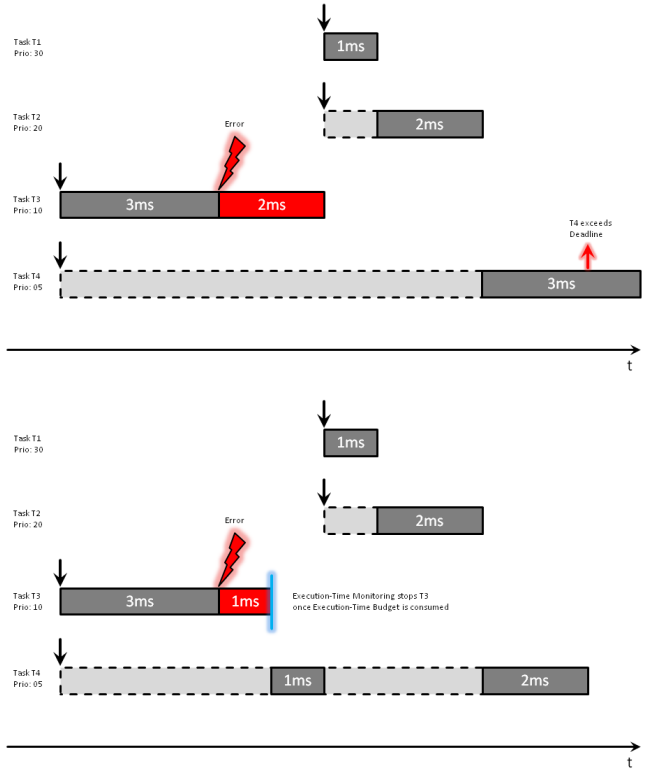

Fig. 4. Timing Protection: Execution-Time Monitoring of Tasks
Top: Error-case without Execution-Time Monitoring
Bottom: Error-case with Execution-Time Monitoring of T3

Once such timing-protection mechanisms are in place inside the OS, follow-up questions arise: How to configure the mechanisms in an appropriate manner? Which task needs to be protected? Which protection-mechanism needs to be used? What are appropriate budgets for certain tasks? etc. In order to answer these questions, at least 3 building-blocks are needed:

1) We need a model of the system, which captures the main timing-attributes of the system. However, the main challenge is not the model itself, but to retrieve the timing-values from the real system.
2) We need an analysis method, which allows us to analyse how the system will behave in certain cases. Hence the analysis-method needs to be aware of the protection-mechanisms.
3) We need to identify all relevant error-cases which need to be analysed, in order to feed them into the analysis-method.

VIRTUAL VEHICLE has implemented a timing-analysis method accordingly. It is based upon the response-time analysis (RTA) approach, but is extended such that it also considers timing-protection mechanisms. It can handle both *execution-time monitoring* as well as *inter-arrival rate monitoring*. Currently we are working on a systematic approach to identify error-cases, which shall be analysed by the timing-analysis method.

B. Memory-Allocation Mechanism

In the context of timing-analysis, memory access strategies become increasingly important. This is due to the completely different memory and hardware architectures and configurations of the systems (e.g. an automotive micro-controller with some KBs of RAM, 1 level cache and fixed memory allocation of data and code at design-time vs. general-purpose architectures with several GBs of RAM allocated on 2, 4, or more DIMMs with dynamic memory allocation at run-time).

Dynamic memory is a crucial shared resource, especially for multi-core systems. An inconvenient allocation could thwart the system-performance significantly. For example, mechanisms like the address space layout randomization (ASLR) or load balancing algorithm arrange the address space position of memory pages randomly. Hence, performance can significantly vary between *best case* and *worst case*.

For example, let's consider a system with 4 cores, shared cache, and shared RAM (see figure 5) in which tasks allocate memory at run-time. In the worst case scenario, every core Cx allocates memory pages Mx, on the same RAM DIMM. If C1, C2, C3, and C4 try to access their memory pages M1, M2, M3 and M4 at the same time t1, concurrent address- and data-bus access to the memory starts. If C1 wins the arbitration process C2, C3 and C4 have to wait accordingly. Hence, the time to access instructions and data in memory can become unpredictable.

Therefore, memory access time could be a bottleneck for safety-critical systems on general-purpose hardware. This raises other questions: Is it possible to configure a dynamic system in a way that the measured system behavior of a test run is meaningful? Do the test-runs output reproducible results? If not, is it possible to define a metric with barriers which are acceptable in a safety-critical context? Could we apply probabilistic timing-analysis methods to predict timing behaviour of a system?

Within the last years many features have been implemented into Linux to extend the memory management with respect to safety requirements and to provide support for process isolation. However, configuring these mechanisms (`Linux containers` [8], `cgroups` [9], `PALLOC` [10], or

MemGuard [11] to name just a few) in the right manner, in order to produce a traceable system behavior, is a challenging engineering-task. Linux containers for instance are used to isolate Linux processes into their own little subsystem. That means it is possible to run multiple isolated Linux systems (so called containers) on a single host. In contrast to a virtual machine which emulates the hardware to it's guest systems, Linux containers are lightweight and provide a virtual environment that has its own CPU, memory, block I/O, etc. space and a resource control mechanism. Namespaces and cgroups provide these features inside the Linux Kernel. Cgroups provide some other important features to limit, police and account the resource usage for a set of processes. With PALLOC we can allocate memory to specific DRAM banks, allowing us to improve the quality and performance in a flexible manner and to prevent mutual exclusion of memory access as described in the scenario above. Figure 5 shows an inconvenient multi-core memory allocation (on the right) and an optimized memory allocation with PALLOC (on the left).

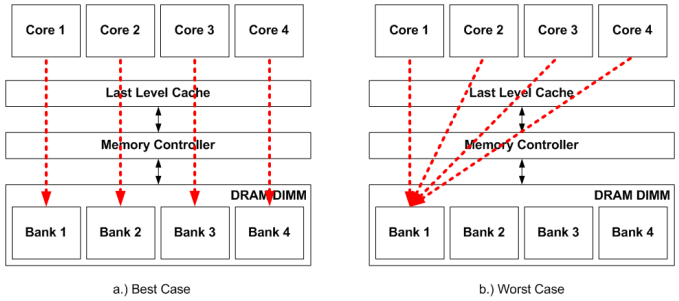

Fig. 5. Memory Protection and Performance
Left: Memory allocation with PALLOC
Right: Adverse memory allocation with loss performance

We are optimistic, that equipped with these features of the Linux kernel, which are provided by the open source community, it should be possible to configure a system based on general-purpose hardware to run safety-critical applications which can be qualified at least to SIL 2 / ASIL B with the reference safety process provided by the OSADL project SIL2LinuxMP.

VI. Outlook

We are still at an early stage of our research project. We have started developing the use-case (at item and hazard level), implemented a timing-analysis method for evaluating timing-protection mechanisms, and identified a set of technical challenges associated with timing-protection and memory-allocation.

In order to address these challenges and answer our research questions, our next steps will be as follows: First, we will implement both software-components (safety-critical steer by wire control and non-critical object-detection). The components will then be packed into real-time threads. Then, we will set up methods which allow us to profile the behaviour of these threads. Special focus will be set of timing (e.g. trigger-period, execution-time, etc). Based on this setup, we will then perform our tests:

First, we will profile the behaviour of the system for the case where only the safety-critical software-component is executed. This case will act as our reference-point. Second, we will profile the system-behaviour for the case where both software-components (safety-critical and non-critical) are executed on the same multi-core hardware-platform. Our hypothesis is that the non-critical (yet computational-intensive) software will stress the system, and thus cause some degree of interference. This is due to shared hardware-resources (such as RAM and L2-cache). The question is how large this interference is and whether it is within acceptable bounds. In case the interference exceeds acceptable bounds, further analysis will need to be performed, in order to identify which parts of Linux would need to be improved. Promising methods / technologies, which may help to reduce the degree of interference might be MemGuard [11], PALLOC [10], cgroups [9], Linux Container [8] and namespaces [12].

Acknowledgment

The authors would like to acknowledge the financial support of the "COMET K2 - Competence Centres for Excellent Technologies Programme" of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economy, Family and Youth (BMWFJ), the Austrian Research Promotion Agency (FFG), the Province of Styria and the Styrian Business Promotion Agency (SFG), the ARTEMIS Joint Undertaking project EMC^2 (grant agreement nb 621429), and the European Union's 7^{th} Framework Programme project eDAS (grant agreement no 608770).

Finally we would like to thank OSADL and the SIL2LinuxMP project team for their support.

References

[1] SAE International, "SAE J 3016: Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems," 2014. [Online]. Available: http://standards.sae.org/j3016_201401/
[2] European Commission, "Towards a European road safety area: policy orientations on road safety 2011-2020," 2010.
[3] International Organization for Standardization, "ISO 26262: Road vehicles – functional safety," 2011.
[4] International Electrotechnical Commission, "IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems," 2010.
[5] Open Source Automation Development Lab, "SIL2LinuxMP." [Online]. Available: https://www.osadl.org/SIL2LinuxMP.sil2-linux-project.0.html
[6] "Open Source Computer Vision (OpenCV)." [Online]. Available: http://opencv.org
[7] AUTOSAR Development Partnership, "AUTomotive Open System ARchitecture (AUTOSAR)." [Online]. Available: http://www.autosar.org
[8] "Linux Containers." [Online]. Available: https://wiki.archlinux.org/index.php/Linux_Containers
[9] "Cgroups." [Online]. Available: https://wiki.archlinux.org/index.php/Cgroups
[10] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 155–166.
[11] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.
[12] M. Kerrisk, "Namespaces in operation, part1: namespaces overview," 2013. [Online]. Available: https://lwn.net/Articles/531114/