Requirements on Next-Generation Operating Systems for Automotive Systems

Arne Hamann, Dirk Ziegenbein
Robert Bosch GmbH
{arne.hamann,dirk.ziegenbein}@de.bosch.com

Selma Saidi
Hamburg University of Technology
selma.saidi@tuhh.de

ABSTRACT

Automotive E/E architectures are currently undergoing a radical shift in the way they are designed, implemented and deployed. Especially, the computational power and communication bandwidth required for new functionalities, such as automated driving or connected vehicle functions, exceed the capabilities of current compute nodes (mainly micro-controller SoCs) and is leading to a reorganization of automotive systems following the paradigm of so-called *centralized E/E architectures* that are based on a new class of computing nodes featuring more powerful micro-processors and accelerators such as GPUs. This paper discusses the impact of that trend on needed execution management mechanisms and motivates requirements for future operating systems in the automotive domain.

1 INTRODUCTION

Traditionally, automotive Electric/Electronic (E/E) architectures consist of around 100 mainly mono-functional electronic control units (ECUs), which are ordered into domains like chassis, powertrain, comfort and infotainment and connected via a mix of in-vehicle networks featuring CAN, LIN, Flexray and Ethernet. For over two decades a move from mainly mono-functional ECUs to domain- or vehicle-centralized architectures has been advocated but did not materialize beyond occasional small-scale integrations, such as a combined engine and transmission control ECU.

However, computational power and communication bandwidth required for new functionalities, such as automated driving or connected vehicle functions, exceed the capabilities of current compute nodes (mainly micro-controller SoCs) and is leading to a reorganization of automotive system following the paradigm of so-called *centralized E/E architectures* (Fig. 1) that are based on a new class of computing nodes featuring more powerful micro-processors and accelerators such as GPUs.

This trend towards centralized E/E architectures directly leads to novel requirements and challenges for operating systems used in the automotive domain that will be briefly discussed in the subsequent sections.

2 COMPOSABILITY OF HETEROGENEOUS SW APPLICATIONS

With the new organization of automotive systems using centralized E/E architectures, heterogeneous applications will be co-existing on the same HW platform, heterogeneous not only in their model of computation (ranging from classical periodic control over event-based planning to stream-based perception applications) but also in their criticality in terms of real-time and safety requirements. As a result, the burden of integration is shifted from the network to the ECU level and in this regard typically from the vehicle manufacturer to the supplier of the control unit. Catering for the various requirements of these applications causes new integration challenges, in contrast to the often stated argument that a centralized architecture reduces the complexity of integration.

To cope with these challenges, there is a need for advanced and expressive *Execution Management Mechanisms* that enable shared resources (especially the processing units but also the memory bandwidth) to be provisioned in a "correct-by-construction" manner. Constructive approaches are especially important given the emerging distributed development style, where one integrator is in charge of integrating software functionalities from many different vendors onto a single platform.

Based on this discussion, we identified key requirements imposed on future automotive operating systems:

(1) *Support for heterogeneous applications:* Future execution mechanisms must be capable of catering for the real-time requirements of heterogeneous applications with different *Models of Computations.* This is in terms of varying workloads (massive streaming data, short scalar data), activations (time triggered, event triggered), data flow semantics (synchronous, actor based, register semantics, independent activations, task chains), real-time characteristics (soft, hard, firm, weakly-hard), and fixed and dynamic workloads. Application heterogeneity also encompasses integrating tasks with different criticalities and safety requirements on the same platform leading to the need for *temporal isolation.*
(2) *Efficient temporal isolation:* Another central requirement is temporal isolation ensuring that the execution behavior and temporal properties of a given application is independent of other co-existing applications. With this, applications can be verified in isolation and integrated on the target platform in a "correct-by-correction" manner, thereby reducing qualification efforts. However, existing operating system and hypervisor mechanisms ensure temporal isolation by "fixed allocations" and by dimensioning systems according to their worst-case execution requirements. Obviously, this comes at the cost of wasted resources which is prohibitive in the cost-driven automotive domain. Hence, the need for efficient isolation mechanisms that utilize the system resources meaningfully.
(3) *Precise QoS control:* Mechanisms ensuring temporal isolation allow for composability of applications which is key for efficient development. These mechanisms shall additionally allow to comprehensively control the QoS for each application, in particular, to ensure progress of "best-effort" parts. Consequently, future operating systems shall allow to configure and enforce minimum guarantees on the resources that each application receives.

NGOSCPS 2019, April 15, 2019, Montreal, Canada
.

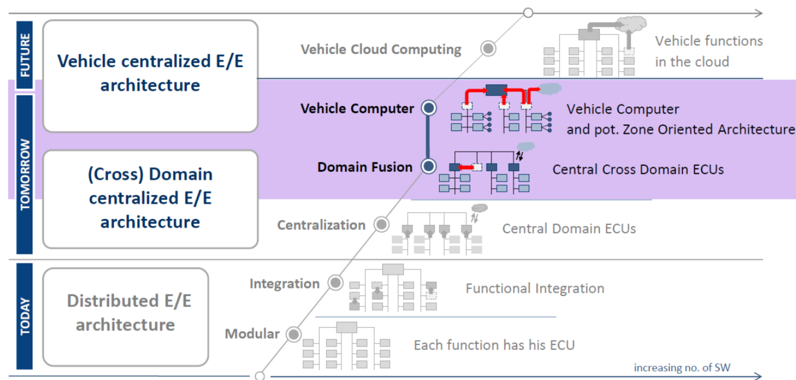

Figure 1: Roadmap showing the evolution from mono-functional ECUs to centralized E/E architectures

System designers have traditionally employed scheduling mechanisms like Fixed Priority Preemptive scheduling (FPP) or Time Division Multiple Access (TDMA) to integrate multiple applications onto a common HW platform. For instance, existing virtualization techniques based on hypervisors (e.g. PikeOS, LynxSecure) wrap safety-critical functions in separate OS components which are given exclusive access to hardware resources, thereby guaranteeing temporal isolation based on TDMA.

FPP and TDMA were adequate in dealing with traditional automotive applications that were simpler as they had well known activation patterns (usually periodic) and execution requirements (worst-case execution times) that were easy to obtain for the used simpler micro-controllers. Hence it was easy to reasonably dimension a system for these static and periodically occurring workloads. However, emerging applications have more complex activation and data flow semantics and it is difficult to precisely determine their execution requirements on heterogeneous HW platforms.

The main disadvantages of FPP is that it is not composable and does not provide temporal isolation. For instance, a newly added functionality affects the temporal behavior of all lower priority functionalities. In other words, temporal guarantees must be reassessed, and usually a system reconfiguration is necessary. Due to its static nature, TDMA also suffers from non-composability. It is built for ensuring temporal isolation but at the cost of heavy inefficiency due to its non work-conserving nature where unused time slices cannot be utilized opportunistically by other applications. This makes TDMA in particular unsuitable for heterogeneous applications with varying execution requirements.

There are promising and well known approaches especially in the domain of *server-based scheduling* that were proposed by the real-time research community in the last decades. Examples are Polling Periodic Servers [3], Deferrable Servers [3], Sporadic Servers [3, 5] as well as Constant Bandwidth Servers [2] in conjunction with mechanisms such as CASH [4]. However, only very few of these approaches have found their way into mainline commercial RTOSes and hypervisor solutions.

The task of the next years is to devise new abstractions for the application layer as well as execution management mechanisms based on those approaches. The building blocks are available but an integration into a comprehensive execution management framework addressing all of the above mentioned requirements is still missing.

3 PROGRAMMABILITY OF HETEROGENEOUS HW PLATFORMS

One important topic when it comes to heterogeneous HW platforms is programmability, i.e. a common framework and programming models that enable software engineers to seamlessly exploit the parallel capabilities of SoCs. Established (cross-platform) abstraction layers such as CUDA, OpenMP, OpenCL, and SyCL are crucial to enable portability and increase development efficiency. The problem, however, is that different HW platforms from different vendors support a different subset of those frameworks. For instance, the GPUs in the Nvidia Jetson series are programmed with CUDA, and OpenCL is not officially supported. The Xilinx Zynq Ultra-Scale+, on the other hand, is programmable using OpenCL, and neither OpenMP nor CUDA are supported. The lack of a commonly supported programming model challenges the efficient use of heterogeneous platforms from an engineering point-of-view.

Another important issue is the lack of clear semantics from a performance point-of-view. Event though code written, for instance, in OpenCL might be "code-portable", it might not exhibit the same execution semantics on a different platform, or, in other words, it is not "performance-portable". Recent studies reveal performance pitfalls when using CUDA for programming real-time applications on NVIDIA GPUs [7]. That work showed by means of micro-benchmarks that the documented (or intended) behavior of CUDA program code often diverges from the actual behavior on the platform. One prominent reason for that are blocking effects due to implicit synchronization that are not transparent to the programmer. Moreover, depending on the implementation, the behavior of the same piece of code may change between different HW platform generations.

In addition to programmability and execution semantics, standard implementations of parallel programming models (e.g. CUDA and OpenCL) challenge software qualification by employing features restricted by certification standards, like the use of pointers, dynamic memory allocation and explicit data management between address spaces. These features jeopardize time predictability and can also be very error prone thereby increasing the maintenance and validation effort. One possible approach as proposed by [6] relies on restricting some features of standard parallel programming languages non compliant with safety standard guidelines in order to later ease the certification process.

It is evident, there is a need for a formally defined programming model for heterogeneous HW platforms that encompasses real-time predictability to solve the issues concerning the programmability of high performance platforms for real-time embedded systems. However, in the end such a programming model must be strongly rooted in (standardized) operating system mechanisms. Which mechanisms are needed and how they can be standardized is still an open question.

4 RECONCILE PREDICTABILITY AND PERFORMANCE

The centralization and integration trends in the automotive industry are driving the adoption of common embedded consumer devices as new building blocks for centralized E/E architectures. The utilized platforms include heterogeneous Multiprocessor Systems-on-Chip (MPSoCs) integrating general-purpose ARM-based microprocessors and accelerators such as GPUs, previously restricted to the high performance computing domain.

Compared to micro-controllers, general-purpose MPSoCs are highly parallel and feature an increased I/O interaction and a more complex memory system composed of multiple levels of on-chip shared SRAMs memories (caches or scratchpads) and off-chip DRAMs, in addition to dedicated DMA engines providing hardware support for moving data between memory locations. However, with the increased complexity in the memory system also appears an increased correlation between the execution of a program and the access to the data it manipulates. This data is transparently available to the program through virtual address space, however, it is in practice stored in different remote memory locations with different access latencies and accompanied with complex mechanisms for ensuring data coherency and consistency.

As a consequence, the timing effects of memory contentions is by orders of magnitude more severe in heterogeneous MPSoCs with off-chip DRAMs compared to micro-controllers. This is further worsened in the presence of data-intensive applications which are becoming an important class of automotive systems to realize advanced assisted or automated driving functions. Therefore, bounding the timing effect of memory contentions in architectures with a multi-level shared memory hierarchy and complex mechanisms for transferring data becomes extremely challenging.

To address this problem future real-time operating systems and hypervisor solutions must adopt constructive approaches controlling and limiting contention effects on the memory hierarchy to ensure timing predictability. One guiding example in this direction from the past years is the work on *MemGuard* [8] where the operating system monitors and enforces each core's memory bandwidth usage to bound cross-core interference and, thus, improve predictability.

5 POSSIBILITY FOR RESOURCE INSPECTION ON APPLICATION LEVEL

Anytime applications [1] gain importance in embedded real-time and cyber-physical systems, especially when dealing with open contexts, intelligent decision making and advanced control approaches such as model-predictive control (MPC). The basic property of an anytime application is that it allows a trade-off between the quality of the application results (i.e. QoS - Quality of Service) and its execution time.

While being in use for general purpose computing for many years, the use of anytime algorithms in embedded real-time systems requires to reconcile the execution time flexibility with hard functional and quality of service guarantees (e.g. hard real-time requirements). This is of particular importance for upcoming integration platforms in the automotive domain (centralized E/E architectures) accommodating automated driving-related applications.

There are two different solution approaches available today:

(1) The trade-off between QoS and execution time is performed offline at system design time (e.g. the number of prediction iterations of an MPC algorithm is fixed) and the execution time budget is assigned according to the worst case execution time of the chosen trade-off. By this, the anytime application is in fact reduced to a fixed-time application.
(2) The anytime application performs the trade-off online during runtime based on some system-specific information (i.e. specific and fixed combination of competing applications and underlying processing platform).

Obviously, both approaches are not satisfactory for efficiently using anytime applications in embedded real-time systems. The first approach leads to over-provisioning of resources, and thus non-optimal usage of computing resources and higher product cost. The second approach results in high engineering efforts for porting applications between product generations and variants.

One possible solution to enable the efficient usage of anytime algorithms in embedded real-time systems is for the operating system to provide a resource introspection interface to the application layer allowing to reason about available computational resources on user-level.

Such an interface would provide the possibility to implement applications with dynamic adaptation of the QoS based on the current runtime situation and deployment context. This capability is not available in any of the established (real-time) operating systems in the market. The current question at hand is how such an interface should look like and how it can be integrated with existing real-time techniques into next-generation operating systems.

REFERENCES

[1] Shlomo Zilberstein, last access January 2019. *Using Anytime Algorithms in Intelligent Systems.* Available at http://rbr.cs.umass.edu/shlomo/papers/Zaimag96.pdf.
[2] Luca Abeni, Giuseppe Lipari, and Juri Lelli. 2015. Constant Bandwidth Server Revisited. *SIGBED Rev.* 11, 4 (Jan. 2015), 19–24. https://doi.org/10.1145/2724942.2724945
[3] Giorgio C. Buttazzo. 2004. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series).* Springer-Verlag TELOS, Santa Clara, CA, USA.
[4] M. Caccamo, G. Buttazzo, and Lui Sha. 2000. Capacity sharing for overrun control. In *Proceedings 21st IEEE Real-Time Systems Symposium.* 295–304. https://doi.org/10.1109/REAL.2000.896018
[5] Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. Aperiodic task scheduling for Hard-Real-Time systems. *Real-Time Systems* 1, 1 (01 Jun 1989), 27–60. https://doi.org/10.1007/BF02341920
[6] M. M. Trompouki and L. Kosmidis. 2018. Brook auto: high-level certification-friendly programming for GPU-powered automotive systems. In *Proceedings 55th Annual Design Automation Conference (DAC 2018).*
[7] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith. 2018. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *Proceedings 30th Euromicro Conference on Real-Time Systems (ECRTS 2018).*
[8] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium.* IEEE Computer Society, 55–64.