CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Migrating a Single-core AUTOSAR Application to a Multi-core Platform: Challenges, Strategies and Recommendations

Master's thesis in Computer Science and Engineering

Simon Widlund
Anton Annenkov

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Migrating a Single-core AUTOSAR Application to a Multi-core Platform: Challenges, Strategies and Recommendations

Anton Annenkov
Simon Widlund

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Migrating a Single-core AUTOSAR Application to a Multi-core Platform: Challenges, Strategies and Recommendations
Anton Annenkov
Simon Widlund

Supervisor: Risat Pathan
Examiner: Jan Jonsson

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Migrating a Single-core AUTOSAR Application to a Multi-core Platform:
Challenges, Strategies and Recommendations Simon Widlund & Anton Annenkov
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As performance requirements on automotive Electronic Control Units (ECU) increase, multi-core hardware is becoming more common. Due to increasing complexity of automotive software, a group of vehicle manufacturers specified a standard called Automotive Open System Architecture (AUTOSAR). Since 2009, AUTOSAR supports multi-core platforms. The single- to multi-core transition does not come without challenges, however. An AUTOSAR software stack consists of Basic Software (BSW), which is essentially the OS and the drivers, and the Application Software (ASW), which is the automotive application functionality. Both should be parallelized in order to fully utilize multi-core hardware. The automotive industry is highly reliant on legacy software that is thoroughly proven and tested. This software eventually needs to be adapted and migrated to multi-core platforms to enable the vehicle manufacturers to add more functionality, e.g., active safety and autonomous driving, without increasing system complexity to unacceptable levels. This thesis describes the migration process, both in theory and practice. We have surveyed recent research as well as worked with migration of a real-world AUTOSAR-based system. We found that the AUTOSAR-compliant OS significantly deviates from the AUTOSAR standard, with respect to multi-core support. It was concluded that most current strategies for multi-core migration cannot be used without modifications, due to these deviations. Finally, we suggest an approach for multi-core migration, where the legacy program execution order is preserved, which should be evaluated in future work.

Keywords: AUTOSAR, automotive, real-time, multi-core, migration

Acknowledgements

We would like to thank our Chalmers supervisor, Risat Pathan, for his enthusiastic support and all his help. Thank you for asking the tough questions and pushing us to be better! We would also like to thank our examiner, Jan Jonsson, for providing valuable mid-thesis feedback. We also want to thank our Volvo supervisor, Ieroklis Symeonidis, for proposing the thesis in the first place, having already found a supervisor and examiner for us. Thank you for always caring about our progress, and helping us with anything we need. Furthermore, we want to thank some of the other engineers at Volvo, most notably Johan Wranker ("you're our only hope"), Gerhard Olsson, Karl Erlandsson and Kristian Larsson. All of you are incredible engineers, and we really appreciate taking so much time from your busy schedules to help us! We also want to thank the department manager, Anders Henriksson, for helping us with any problems related to the thesis work. Simon would like to thank his fiancée, Liisa, for her unwavering love and support. Anton would like to thank his S.O., Åsa, family and friends for their love and support.

Anton Annenkov & Simon Widlund
Gothenburg, May 2017

Contents

1

Introduction

In the automotive industry, Electronic Control Unit (ECU) means a generic embedded system used to control one or more subsystems in the vehicle. The performance requirements of these ECUs are constantly increasing, as manufacturers seek to increase the amount of software in vehicles [15]. Two types of functionality are under heavy development throughout the industry, namely infotainment and vehicle automation. Vehicle automation, including active safety, stands out by being inherently safety-critical, thus requiring high real-time performance [24].

As increasing ECU performance by increasing clock frequency is becoming obsolete, the industry has to transition from single-core to multi-core solutions. However, this presents new challenges since the automotive industry is reliant on legacy software, and is subject to strict safety requirements. Being able to migrate legacy software to multi-core platforms with minimal changes to the software is therefore an important step in this transition [24].

As the amount of software increases, the complexity of the software also increases, making it difficult for vehicle manufacturers to meet the demands of customers as well as adhere to new, stricter legal standards related to environment and safety. Consequently, leading vehicle manufacturers and suppliers decided to work together, creating the Automotive Open System Architecture (AUTOSAR) standard. AUTOSAR is a layered software architecture which on a high level consists of Basic Software (BSW) and Application Software (ASW). The BSW is equivalent to the operating system (OS) and the drivers on a modern PC or mobile device. The ASW is the specific functionality, equivalent to an application on a PC or mobile device.

To be able to fully utilize the potential of multi-core hardware, the software must be parallelized. Software parallelization is a very active research field relevant not just to the automotive industry, but to virtually every industry that wants to increase the amount of software in their products. The goal of migrating the application software to a multi-core system is to gain better performance, by distributing the workload such that it allows adding new functionality. This is related to Gustafson's law, which essentially states that multi-core platforms can deal with larger problem sizes [20]. However, there are several challenges with the migration process, since the ASW must execute correctly post-migration. Additionally, the single- to multi-core migration of the ASW requires adaption of the BSW in order for it not to become

a performance bottleneck [15].

Multi-core systems running AUTOSAR applications also present new challenges in task scheduling (i.e., scheduling tasks in such way that they meet their timing constraints), since traditional scheduling approaches for real-time systems cannot be applied directly for AUTOSAR applications [19]. Scheduling is important in order to meet timing constraints (i.e., strict deadlines) as well as distributing the application load evenly. A balanced processor load is needed since all cores in the system need to handle resource usage peaks due to event or error handling.

To the best of our knowledge, there exists little research in the area of AUTOSAR multi-core migration which consider all of the steps in the process of migrating the ASW and BSW. Macher et al. [24] have described some general strategies on a high level, but do not consider any specific multi-core solutions such as how ASW should be parallelized. While there are solutions on how the ASW and BSW should be parallelized, such as in [28] and [15], such solutions must be carefully evaluated on a real system. In general, few papers focus on multi-core migration of a real-world AUTOSAR software. In this thesis, we will perform a migration of a single-core AUTOSAR application to a multi-core platform. The platform, called Driver Assistance Control Unit (DACU), runs active safety application software and is provided by Volvo Group Trucks Technology. We will also present the challenges, strategies and our recommendations for the multi-core migration process.

1.1 Problem Definition

This thesis was conducted at Volvo Group Trucks Technology in Gothenburg. The hardware used is a dual-core ECU based on the Zynq 7000 processor made by Xilinx, further described in section 3.1. We will conduct our research using industry-standard software and tools, including an AUTOSAR software stack made by Vector, based on version 4.1 of the AUTOSAR architecture. This thesis addresses the following scientific problems:

- *Which parts of an industry-standard AUTOSAR implementation should, according to research, be considered/modified when migrating from a single-core platform to a multi-core platform?*
  As the amount of software in vehicles increases, being able to migrate an existing set of software to a multi-core platform is essential for the industry. We will survey recent research on AUTOSAR multi-core migration and present the theoretical migration strategies proposed by other researchers.

- *Is the theoretical framework for multi-core migration applicable to a real-life system?* We will perform multi-core migration and investigate if there are any deviations stopping us from performing the migration according to theoretical methods. Moreover, these deviations will be analyzed with respect to the theoretical framework.

2

Background

This chapter aims to give the reader a solid background for understanding the rest of the report. Relevant real-time system concepts are explained in section 2.1. Section 2.2 gives a short introduction to parallel software and a little bit about the hardware involved. AUTOSAR and its multi-core support is explained in section 2.3.

2.1 Real-time Systems Concepts

As mentioned in the introduction, a real-time system must meet strict timing constraints, i.e., deadlines, in order to provide a correct service [22]. A common definition for a real-time system is:

> A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated [32].

In a real-time system, *tasks* (i.e., pieces of software) can have *hard* or *soft* deadlines, where missing a hard deadline in e.g., an automotive system can result in physical damage or loss of life. Soft deadlines may be missed without causing a catastrophic failure, but rather result in reduced quality of the provided service.

Real-time systems are very common in the automotive industry, where control systems such as brake-by-wire systems consist of tasks that must meet hard deadlines. There are also numerous systems with soft deadlines, like infotainment and multimedia systems such as a GPS navigation service or a music player, where the service quality will be negatively affected when tasks miss their deadlines.

2.1.1 WCET Estimation

Determining the *Worst-Case Execution Time* (WCET) is essential in the area of real-time systems. Generally, most tests which are used in order to validate that the strict timing constraints are met use task parameters such as WCET for each

individual task. Therefore, in order to show that the strict timing constraints are met, the WCET of each task needs to be known. Generally, there are two ways to derive the WCET, which are explained below.

Assume that we want to find the WCET by simply letting the task run for a certain number of times, thus finding the *measured execution times* for the task. Thus, both the *minimal observed* execution time and *maximal observed* execution time can be found, as shown in figure 2.1. This is an example of a *measurement-based method* [36]. The measurement is done several times, after which a maximal observed execution time has been obtained. Furthermore, figure 2.1 shows two different curves. The white curve depicts the execution times which are observed through measurement. The same execution time may be observed several times, as shown in the y-axis of the figure as the distribution of times. The dark curve shows the *actual* execution time for a particular task. As shown in the figure, the actual WCET is higher than the maximal observed execution time.

A safety margin should be added in order to obtain a WCET that is pessimistic but tight. The issue with this methodology is that an over- or under-estimation of the safety margin is possible. Another issue is to find the input that yields the WCET for a task, which is practically infeasible for large systems. Despite these issues, this method still provides a general idea of the execution time for tasks [36].

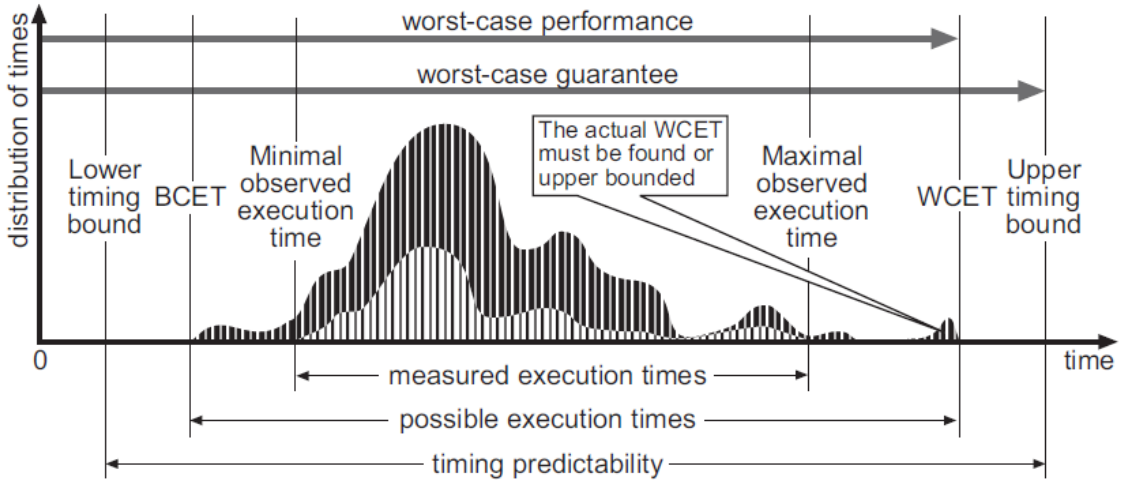

Figure 2.1: Timing analysis terms of a system. The white curve represents the executions times that are observed through measurement. The darker curve shows the *actual* execution times [36].

The second method to estimate the WCET is to perform an analysis of the application software, rather than executing the software. In such an analysis, an abstract model of the processor is combined with the software, and performs an analysis based on the software and the abstract processor model. This analysis provides a *lower and upper timing bound*, which should be close to the actual WCET. Such methods are referred to as *static methods* [36]. This means that, using a static method, the obtained upper timing bound is larger than the *actual* WCET. Thus,

the upper timing bound is pessimistic (since it is higher than the actual WCET), and should be as tight as possible to the actual WCET. Naturally, the hardware description in the abstract processor model must be highly detailed, including CPU pipeline and the whole memory hierarchy, in order to obtain realistic upper timing bounds for the execution times.

In general, both methods for performing a *timing analysis*, i.e., deriving estimates or bounds for execution times, can be compared with respect to two criteria: *safety* and *precision*. Safety means that the obtained estimate or bound should be pessimistic, i.e., higher than the actual WCET. Otherwise, the actual WCET might be missed. Precision considers the closeness to the exact value of the WCET. Generally, any static method will provide an upper timing bound which exceeds the actual WCET. Thus, static methods should provide a WCET which is close to the exact value of the WCET i.e., providing high precision [36].

2.1.2 Multi-core Task Allocation

For a multi-core processor, tasks can either be allowed to execute on one core exclusively (i.e., a group of tasks are *allocated* to a particular core), or on any core that is available at the moment. More specifically, two possibilities for task allocation exist: *partitioned scheduling* and *global scheduling*. Partitioned scheduling means that once a task has been allocated to a core, it cannot migrate to another core. This implies that partitioned scheduling is equivalent to multiple instances of single-core scheduling, since the scheduling of tasks is done core-by-core. This means that the scheduling frameworks that exists for single-core systems can be used in a multi-core system with partitioned task allocation.

The other approach is to use global scheduling, which means that tasks are allowed to fully migrate between cores. Instead of allocating tasks to a single core, a global ready queue is used where tasks that are ready to execute are stored. Then, the task scheduler can decide which core a task should be allocated to during run time. An obvious advantage of this approach, compared to partitioned scheduling, is that all cores can execute any task given that the core is available. For example, using partitioned scheduling, one of the tasks might be waiting to execute on one core, while the other core is available. In such cases, global scheduling is advantageous, since tasks can be executed on any core. The disadvantage, however, is that this approach makes it difficult to predict the task and resource interactions in the system [25]. AUTOSAR, for example, requires all tasks to be statically allocated to a particular core.

2.2 Parallel Software Execution

To understand the possibilities and limitations of parallel computing and multi-core hardware, one must understand Amdahl's law. Amdahl's law gives both a maximum speedup for a given number of processors or processor cores, and an upper bound on speedup for any number of cores.

Figure 2.2 shows some speedup plots in a case where 95 % of the application can be parallelized. Also included in the figure is a "mortar" curve, exemplifying what happens to the speedup when the cross-core communication overhead eats up the added performance of more cores.

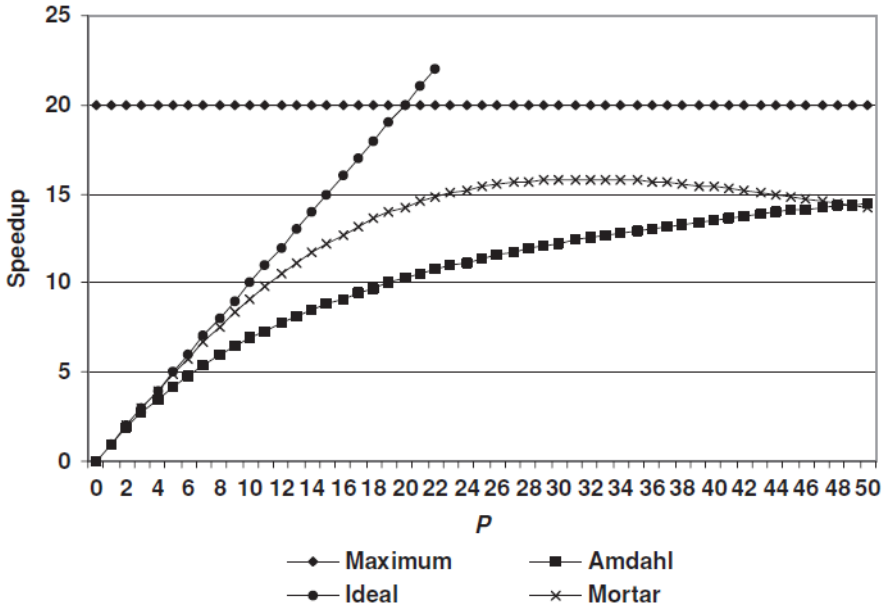

Figure 2.2: Plots of possible speedups with 95 % parallelization. P is the number of cores [17].

There is also an extension of Amdahl's law, called Gustafson's law. It was formulated by John Gustafson and published in 1988 [20]. Amdahl's law assumes a fixed problem size, while Gustafson instead considers an increasing problem size, with fixed execution time. This is highly relevant to the automotive industry, since that is what is currently happening throughout the industry.

2.2.1 Cache Coherence

Cache coherence is a requirement for correct execution on hardware architectures with any number of cache levels. It means that multiple copies of a shared data value must be consistent across all physical memories. Figure 2.3 describes the concept in general terms. In the figure, there is a shared memory resource that can be accessed

by two clients, each with its own private cache. This means that if one or both clients read some data from the shared memory and then update it, we could have three copies of the same piece of data, all with different values. Dubois et al. define cache coherency as follows:

> A cache system is coherent if and only if all processors, at any point in time, have a consistent view of the last globally written value to each location [17].

There are numerous mechanisms and protocols designed to maintain cache coherency. For the purpose of understanding this thesis, only basic knowledge about cache coherence is required.

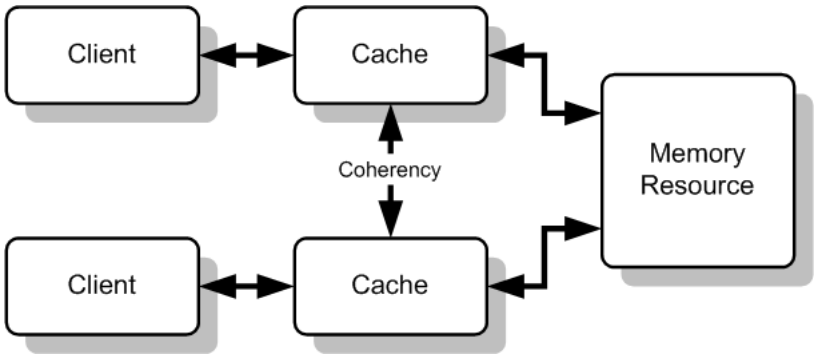

Figure 2.3: General overview of cache coherence.

2.3 AUTOSAR

AUTOSAR stands for Automotive Open System Architecture. The purpose of the initiative is officially described as follows:

> The AUTOSAR partnership is an alliance of OEM manufacturers and Tier 1 automotive suppliers working together to develop and establish a de facto open industry standard for automotive E/E[1] architecture which will serve as a basic infrastructure for the management of functions within both future applications and standard software modules [6].

AUTOSAR started as discussions between BMW, Bosch, Continental, Daimler-Chrysler and Volkswagen in August, 2002. By July 2003, they had set up a joint technical team and signed a formal partnership. Since then, many more automotive companies have joined, including Volvo. On the technical side, AUTOSAR has defined four technical goals: modularity, scalability, transferability and re-usability. The key to realizing these goals is *standardized interfaces* and using a layering approach similar to e.g., the TCP/IP stack, combined with well-defined interfaces between the layers.

[1] Electrical and Electronic, a term commonly used in the automotive industry.

In AUTOSAR, the application software is divided into different Software Components (SWCs), which in turn consist of *runnables*. A runnable can be described as the smallest piece of functionality i.e., an atomic piece of software. An SWC can be viewed as a piece of application software that is independent of the ECU that it is running on. Services needed by the SWCs are provided by the Run-Time Environment (RTE), which in turn uses the AUTOSAR OS and Basic Software (BSW) as seen in figure 2.4, which shows a high-level view of a single-core system running AUTOSAR. The BSW consists of services for the system, such as memory management and communication frameworks.

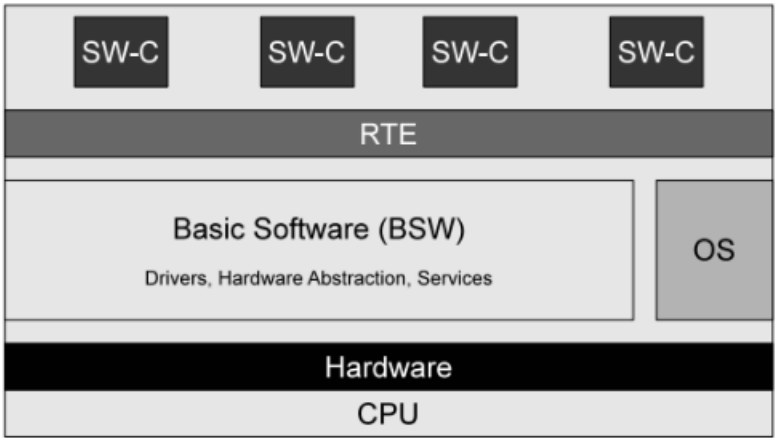

Figure 2.4: AUTOSAR system architecture for a single-core system [15].

The runnables are mapped to *tasks* in the *runnable-to-task mapping* of AUTOSAR. If the processor has more than one core, the tasks must be allocated to one of the cores in the *task allocation* process. AUTOSAR does not allow tasks to migrate from one core to another [12], due to the strict timing constraints in automotive software applications. Therefore, each task is *statically* allocated to a single core.

For AUTOSAR applications, both the runnable-to-task mapping and the task allocation must be considered in order to find an acceptable solution with regards to minimal communication overhead [24]. This overhead originates from inter-runnable communication, since some runnables need to interact with each other. For example, one runnable might read data from a sensor, process the data and send the result to another runnable. Thus, any AUTOSAR compliant system will include runnables that cooperate and communicate, in order to deliver its services. Reducing the inter-runnable communication time is desirable, since the system throughput can potentially be improved [18]. Also, certain constraints, such as precedence constraints, must be taken into consideration when allocating tasks, with the goal of minimizing communication overhead. The precedence constraints are necessary when one task is dependent on the output of another task. For instance, a particular task, A, depends on the value of a variable that is computed by another task, B. Therefore, B must execute before A, and there is a precedence constraint for these tasks. Preserving such precedence constraints when migrating legacy application software to a multi-core system is one of the main challenges with multi-core migration.

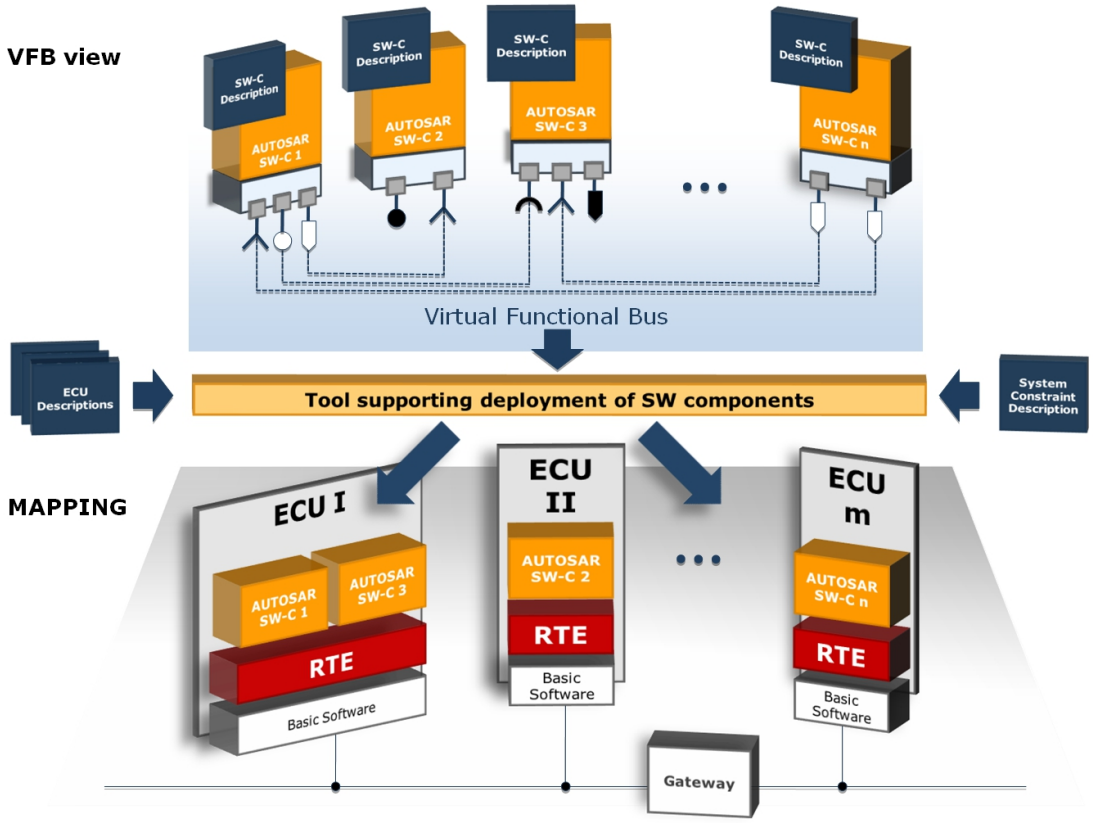

Figure 2.5: AUTOSAR overview [5].

2.3.1 Software Architecture

The AUTOSAR architecture enables software development before the vehicle hardware is specified. Figure 2.5 shows an overview of the AUTOSAR approach. At the top of the figure, the Software Components (SWCs) are shown. The SWCs are the logical components of the application software, containing small parts of the application functionality and logical connection interfaces to other SWCs [8].

Since an SWC consists of a part of the application software and the aforementioned communication interfaces, SWCs can be designed without knowledge about the underlying hardware. Therefore, it is possible to migrate SWCs to another system, due to the hardware independency. The SWCs can be further divided into runnables, as mentioned earlier. Each runnable has some sort of triggering condition, either periodically (i.e., the inter-arrival time is fixed) or driven by external events. Thus, the runnables have scheduling properties such as periods and deadlines which are used when scheduling the tasks, after the the mapping process.

The Virtual Functional Bus (VFB) is a set of abstract communication mechanisms which are independent of the hardware [9]. Therefore, in the VFB, the communication between the SWCs, BSW and OS are independent of the hardware, which makes it possible to integrate SWCs in early development. Integration, in this case,

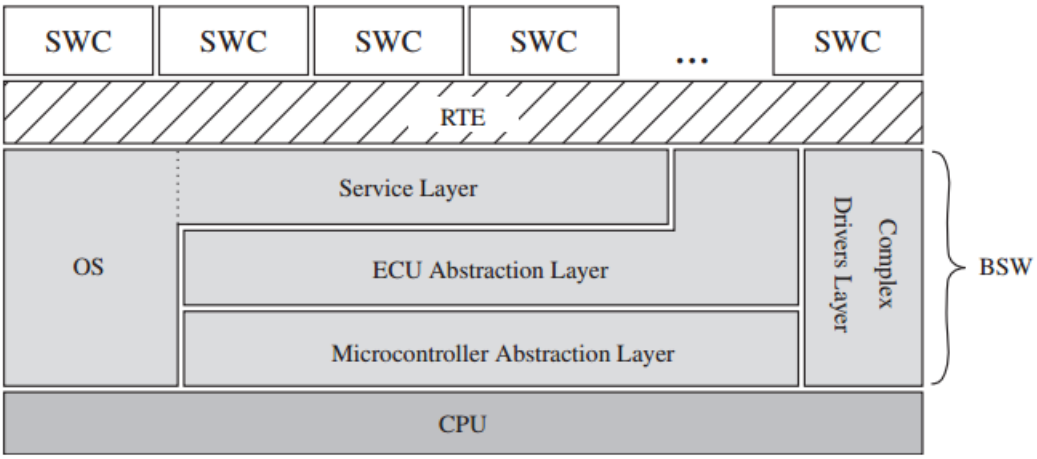

Figure 2.6: The AUTOSAR layered software architecture [14].

means mapping SWCs to physical ECUs and integrating them with the rest of the software stack and physical communication network, as shown in figure 2.5. This means that the abstract view of the communication mechanisms in the VFB is realized when mapping the SWCs to ECUs. The Runtime Environment (RTE) is an ECU-specific implementation of the VFB, which means that it provides communication services to the system. All ECUs in the system will have an ECU-specific implementation of the RTE, which is generated in the ECU configuration process, as seen in figure 2.5.

A slightly more detailed view of the BSW is presented in figure 2.6. The service layer consists of standardized services to the SWCs, RTE and other parts of the BSW such as diagnostic tools, memory management and communication frameworks. In general, the Service Layer is independent of the hardware. Next, the ECU Abstraction Layer provides interfaces to the lower layers in the architecture, and thus it provides an API as well as drivers needed in order to access the ECU hardware. The Complex Drivers Layer (CDL) contains drivers for devices that either have very tight timing constraints, or provides functionality that is not specified in AUTOSAR. Finally, the Microcontroller Abstraction Layer (MCAL) provides the drivers needed to control the peripherals of the microcontroller.

The AUTOSAR OS is considered part of the Service Layer and is an extension of the OSEK OS (ISO 17356-3), which has previously been used extensively in the automotive industry. Both operating systems conform to real-time constraints from the application software, such as hard task deadlines, and therefore the AUTOSAR OS is considered to be a real-time operating system (RTOS). AUTOSAR OS provides priority-based scheduling for *basic* and *extended* tasks. Basic tasks have three different states: suspended, ready and running. Extended tasks include an additional waiting state. On a single-core processor, only one task at a time can be in the running state, where its instructions are being executed. The ready state means that a task fulfills all the requirements of a transition to the running state, but is waiting for the scheduler. Suspended means that a task is passive, but can be

re-activated. The waiting state means that a task is waiting for a pre-defined event before it can continue execution [27]. Moreover, the AUTOSAR standard uses the concept *OS-Applications* to describe a set of e.g., tasks, alarms and interrupt service routines (ISRs) which form a functional unit [7].

2.3.2 Multi-core in AUTOSAR

Multi-core support was first introduced in version 4.0 of the AUTOSAR standard, released in 2009 [30]. However, the OS can still only execute a single thread at a time, which means that the OS has to be replicated on each core. In this version, additional mechanisms were added in order to support multi-core systems. One such mechanism is the Inter OS-Application Communicator (IOC), which makes it possible for OS-applications to communicate between cores. This means that the OS can be used in parallel for each core, enabling true parallel task execution [12]. As mentioned in section 2.1.2, AUTOSAR only allows static task allocation, meaning that tasks are not allowed to migrate between cores.

2.3.3 Multi-core BSW

In AUTOSAR version 4.0, multi-core support was implemented in a very conservative way, allowing the BSW to run only on one dedicated core as seen in figure 2.7. The reason for this approach is that minimal effort is required when migrating single-core software to the multi-core platform, since the SWCs do not need to be modified to support parallel execution. Thus, one of the cores will be the *BSW core*. The other *non-BSW cores* need to access the BSW via cross-core communication through the RTE. The problem with this approach is that it generates massive amounts of inter-core communication in the RTE. Since many BSW calls in AUTOSAR are synchronous, the non-BSW core(s) on a multi-core processor will have to block execution until it receives a reply [15]. Böhm et al. conducted a performance evaluation of this approach, and concluded that it performs worse than the default single-core configuration because of the aforementioned cross-core communication [15].

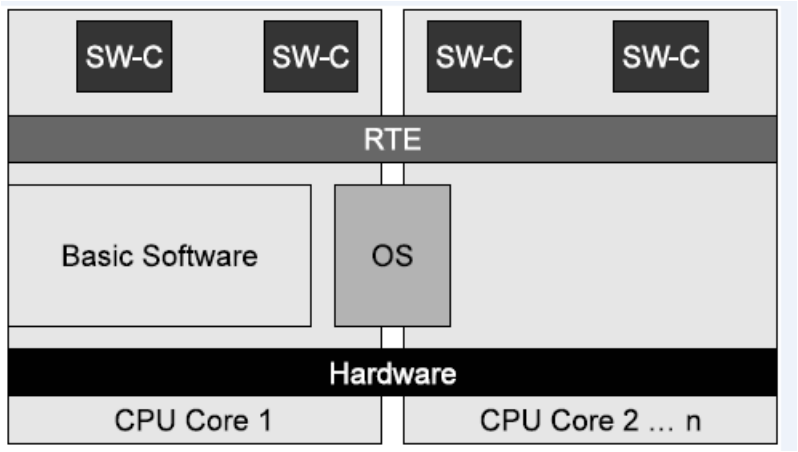

Figure 2.7: AUTOSAR multi-core architecture [15].

In AUTOSAR 4.1.1, released in 2013, the multi-core support was extended, allowing assignment of BSW modules to partitions spread out over multiple processor cores. BSW modules can also be split into *functional clusters*, gathering a number of functionally coherent BSW modules, meaning that all modules related to some function are gathered in a cluster. The clustering is not yet standardized, but a suggestion/example mentioned in AUTOSAR documentation is having separate clusters for communication, memory, I/O and watchdog [11].

3

Method

This chapter describes the hardware, software, tools and methods we used during the migration process. Section 3.1 describes the ECU hardware used during the migration. Next, section 3.2 contains an overview of the software suite and tools. Finally, section 3.3 describes our method of debugging various errors in the software.

3.1 ECU Hardware

The ECU hardware is based on a Xilinx Zynq 7000. It combines two identical ARM Cortex-A9 cores and a Xilinx FPGA on a single chip. Xilinx has logically divided the ECU into a *Processing System* (PS) part and a *Programmable Logic* (PL) part. The PL is the FPGA and its peripherals, while the PS is everything else in the ECU, including the ARM cores [37]. We will not, however, further explain the PL, since it is not relevant for this project.

Figure 3.1 shows an overview of the Zynq 7000, divided into functional blocks. The PS contains an *Application Processor Unit* (APU) which, apart from the ARM cores, notably includes the shared L2 cache and the on-chip memory (OCM). The OCM includes 256 KB of RAM and 128 KB of ROM used for boot code. The other major blocks comprising the PS are memory interfaces, I/O peripherals and interconnects.

Each ARM core has two 32 KB private L1 caches for instructions and data separately. A *Snoop Control Unit* (SCU) monitors the caches and maintains coherency between the private L1 caches. There is also a 512 KB shared L2 cache for both instructions and data. The L2 cache controller allows locking cache content, meaning disabling cache replacement for a cache line, potentially increasing hit rate for critical data. The Cortex-A9 implements the ARM v7-A Instruction Set Architecture (ISA).

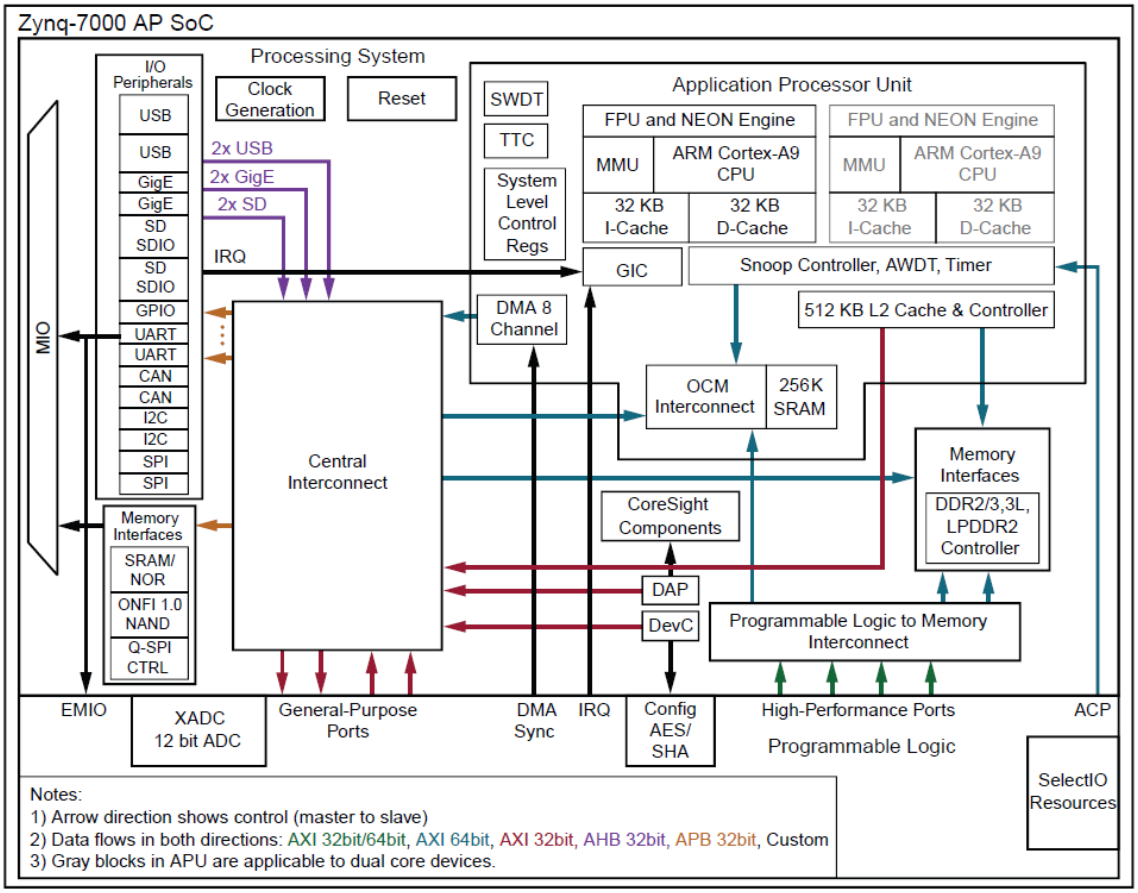

Figure 3.1: Xilinx Zynq 7000 block diagram [37].

3.2 Configuration and Development

We used an AUTOSAR software suite developed by Vector, combined with Volvo's in-house build environment based on Cygwin [16] and *make* [26]. The Vector software tools used were DaVinci Configurator Pro [33] and CANoe [35]. They are parts of a complete toolchain for AUTOSAR, including all steps of the software development process, from system design to verification. DaVinci Configurator is used for SW integration onto the ECU, which includes configuration of BSW and RTE. DaVinci Configurator takes a configuration as input, and outputs source code. It is essentially a GUI used to read and write the configuration.

In figure 3.2, we have illustrated a simplified version of the process of building the software package that is loaded onto the ECU. The process is much bigger than what is included in this figure, but what is shown are the parts that we have been in contact with during our work. We never modified or examined the *Communication and vehicle signals databases*, but it is included in the figure for the sake of completeness.

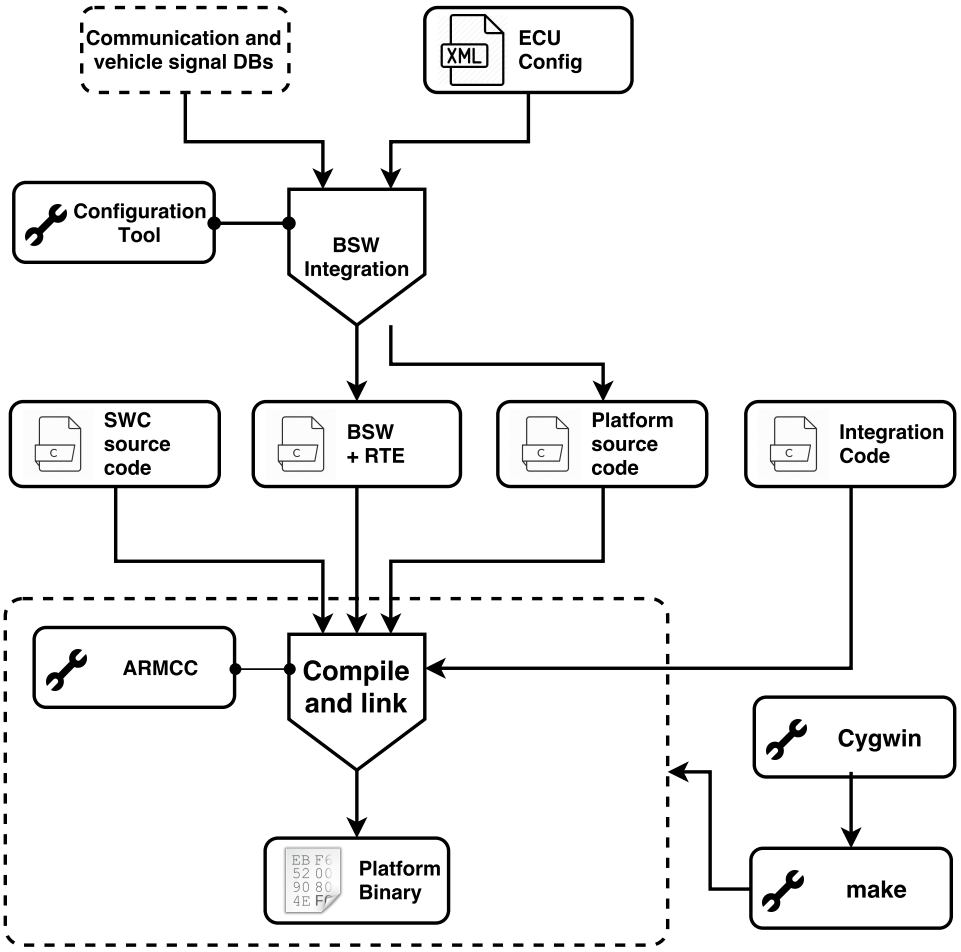

Figure 3.2: DACU software build process.

3.3 Debugging

To debug the software running on the ECU, we used hardware and software from Lauterbach. Specifically the Trace32 PowerDebug 2 module. With the software that comes with the Trace32 debugger, we could debug most features of the APU, including physical CPU registers, cache and main memory. It was also possible to set breakpoints and step through individual assembly instructions in the compiled code. We made extensive use of instruction-stepping, (conditional) breakpoints and memory dumps to find the root causes of the crashes and errors we encountered.

Some errors and crashes were intermittent, so finding the specific part in the code where the error occurred would not always help us fix the problem. This is where our colleagues at Volvo helped us. There were a few experienced software engineers at Volvo who were able to help us debug some of the most elusive problems. In other cases, finding the instruction sequence or an erroneous value in memory was just the start of finding the cause of the problem. In many cases, the problem was on a much higher level, requiring further debugging and discussions with colleagues.

After finding the cause of a problem, fixing it was not always as easy as editing some lines of code. Because of the complex software architecture, there were usually different approaches to solve a software issue. Since a large portion of the BSW source code is generated based on a configuration, we had to choose between trying to fix the configuration parameter(s) causing the problem, edit the generated code, disable the feature or function causing the problem, or in some cases even use the debugger to modify memory during live software execution.

The first choice was always to fix the configuration, since any changes in the generated code could be overwritten. This happens when the code is generated after changing the configuration. In some cases, where we could not find a way to fix the configuration, we had to resort to either script-based or manual patching of the generated source code. Disabling the feature where the error occurs was in most cases a natural step in the debugging process.

4

Related Work

This chapter contains our analysis of the theory related to multi-core migration, providing a more detailed view of each step in the migration process. Proposed solutions for each of the problems will be briefly discussed and will serve as a theoretical framework for the discussion (chapter 6), where we compare theory with practice.

Figure 4.1 shows the general steps which should be made in the multi-core migration process. After choosing suitable hardware (HW), the BSW/OS should be selected. Section 4.1 contains the general strategies which should be used when selecting a multi-core AUTOSAR OS. Furthermore, in the multi-core migration process, one multi-core BSW solution must be chosen, as shown in figure 4.1 as the *BSW parallelization* step. These solutions are described and analyzed in this section. The next step in the process is to parallelize the application software. Both section 4.2 and 4.3 describe the *ASW parallelization* step. Section 4.2 describes the problems with parallelizing automotive application software in general, and recent suggestions made by researchers. Parallelizing the ASW, as shown in the figure, can be made using one of the different solutions that exist. These solutions specifically consider the runnable-to-task mapping and/or task allocation step in the migration process. Section 4.3 describes the problems and solutions related to the runnable-to-task mapping and task allocation process in the multi-core migration.

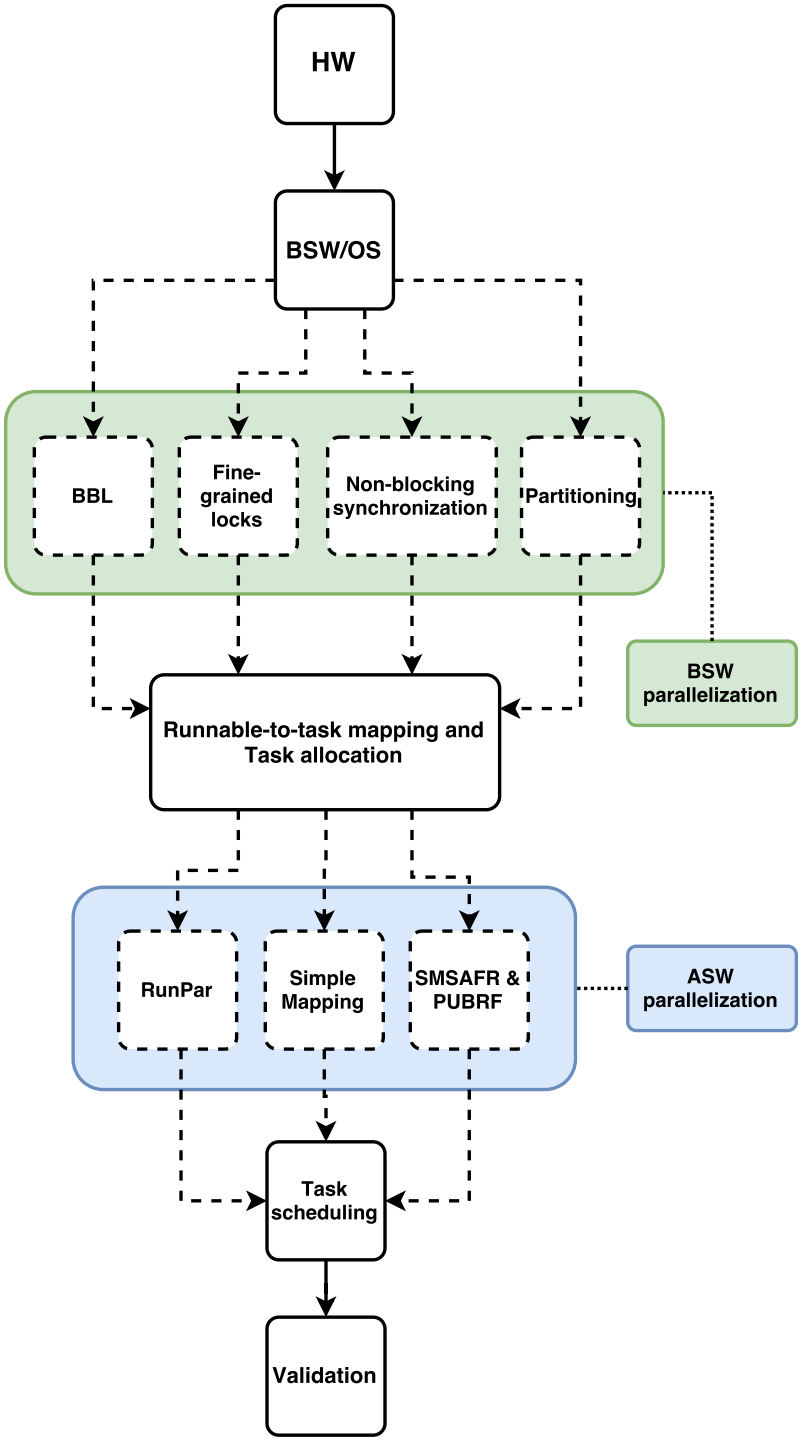

Figure 4.1: Software platform migration flow.

4.1 Multi-core OS/BSW

Automotive software typically consists of many BSW modules and hundreds of thousands of lines of application software code, which should not be modified to any greater extent when migrating to a multi-core system. Both the BSW modules

and the application software are typically very complex and have strict real-time constraints. Additionally, new functionality for e.g., fault tolerance or security is sometimes continuously introduced in the system. Preserving the real-time properties of legacy software, while adding new functionality, is one of the main challenges. Ultimately, the goal is to migrate legacy software while still being able to meet the strict timing constraints [24].

The transition from single-core to multi-core systems in the automotive industry has resulted in several challenges regarding the migration of real-time software. Armengaud et al. state that

> ...the parallel execution, resulting resources and timing conflicts require a paradigm change for the embedded software. Consequently, efficient migration of legacy software on multi-core platform, while guaranteeing at least the same level of integrity and performance as for single cores, is challenging [4].

When choosing a multi-core OS and BSW, Macher et al. [24] propose that both the hardware and software should be considered. First, a CPU and an OS must be chosen with the application software in mind. CPU selection should be done with criteria such as support for debugging tools, in addition to potential benefits of the particular CPU (e.g., performance, lockstepping and HW safety features). The authors state that a profiling of the application should be done in conjunction with selecting the CPU. Some CPUs provide mechanisms that shift the computation of low-level software to hardware. As a consequence, more computation resources are available to the application software. Moreover, the choice of OS should be done based on the chosen CPU. For example, the BSW modules should provide access to hardware mechanisms provided by the CPU. In general, it is important to evaluate the different OSs, listing their key features and analyzing the required integration of the application software for each OS. In general, a multi-core AUTOSAR OS must guarantee mutual exclusion for critical sections as well as avoiding deadlocks [24].

As previously described in section 2.3.3, the AUTOSAR multi-core BSW solution where one of the cores is assigned as a BSW core will result in extensive cross-core communication [15]. This communication originates from whenever the BSW core needs to handle a service request from a non-BSW core. This interrupt will cause the BSW core to activate a task which handles the request, thus fulfilling the cross-core operation. As shown by Böhm et al. [15], the BSW core approach causes high cross-core communication due to BSW access and overhead for cross-core task activation.

Thus, it is not sufficient to only distribute the SWCs in the multi-core system and expect the system to perform better than single-core. The BSW also has to be considered, due to cross-core communication. This means that even though a runnable-to-task mapping and task allocation has been found for a multi-core system, in such way that all deadlines are met in theory, the default BSW solution could cause deadlines to be missed. As shown in figure 4.1, the phase called *BSW*

parallelization must be considered when migrating to a multi-core system. In the following subsection, we will describe one of the multi-core BSW solutions proposed by other researchers. There are other promising multi-core BSW solutions which are explained in Appendix A.1. Moreover, Appendix A.1 describes why these solutions were not considered in this thesis.

4.1.1 Big BSW Lock

The proposed solution on how to integrate the BSW on multi-core platforms by Böhm et al. [15] is called Big BSW Lock (BBL). It is inspired by an old Linux kernel synchronization technique called Big Kernel Lock. The idea is to duplicate the entire BSW and allow it to execute on any core, but only on one core at a time, using mutual exclusion techniques. BBL was implemented with the spinlock API introduced in AUTOSAR 4.0. This is done automatically, by modifying the RTE generator such that spinlocks automatically are added around BSW calls. ISRs are adapted manually i.e., without any modifications of the generator, since spinlocks only are needed for ISRs which access shared data.

Böhm et al. found that BBL is promising, since it shows a slight performance improvement of 3.3 % when only running the BSW, and higher performance gains when having a higher workload[1]. Their performance metric is *activation rate*, which is defined as how many times per second a task can be triggered when the system is under full load. Using BBL, the BSW can be located on both cores without violating fact that only one core at a time can access the BSW. Thus, BBL is promising since it performs better than the BSW core approach, and requires little effort to integrate [15]. The latter is particularly interesting, since the cost and effort required to integrate a multi-core BSW solution should be considered [24] [15].

BBL has some disadvantages, as pointed out by the authors. Using spinlocks to ensure exclusive access to the BSW can cause high latencies on other cores that require the BSW, meaning that they will wait until the lock has been released. This makes the post-migration validation more difficult, since such non-deterministic latencies makes it difficult to perform a timing analysis. Böhm et al. also point out that their workload was not realistic, since they did not have access to real-world automotive applications.

4.2 ASW Parallelization

After choosing a multi-core BSW solution, the application software should be parallelized. Macher et al. [24] describe the general strategy for migrating legacy software

[1] In the authors' setup, the BSW receives simulated CAN-messages which contains certain values. That value is used in each task as input for a busy loop. The busy loop is thus used to simulate a workload.

to a multi-core system. As a first step, the legacy application software should be analyzed and profiled by e.g., performing a WCET analysis/estimation or analyzing the total workload of the software. Next, the system should be decomposed into its smallest possible independent parts. This is an important step in order to extract the most parallelism possible. For example, if two runnables, A and B, have a precedence constraint such that A must finish its execution before B, it is not possible to let both runnables execute in parallel on different cores. This is a time-consuming job, due to the large number of tasks (which in turn consist of many runnables) in an automotive system.

In general, completely independent tasks are uncommon, meaning that only parts of the application software may run in parallel. Thus, the level of parallelism will differ depending on the nature of the application. Task synchronization and identification of shared memory is also important, as this must be considered in the migration process. Shared memory must be kept consistent for each core, meaning that each core must have the same view of the shared memory regions. Finally, the mapping of tasks to each core should ideally be made with the goal of minimizing inter-core communication. This means that tasks which have intense inter-task communication should be statically allocated to the same core [31] [24].

4.2.1 Recommendations on ASW Parallelization

As mentioned by Macher et al. [24], there are some challenges with migrating legacy software to a multi-core system. Some of these challenges are: identifying shared memory regions, preserving the task/runnable execution order according to precedence constraints and mapping tasks to cores such that inter-core communication is minimized.

Schneider et al. [31] describe one of the main challenges with migration to a multi-core system, which mainly consists of handling mutual exclusion constraints. For example, in legacy software, critical sections can be handled by enabling and disabling interrupts. While this prevents race conditions between tasks and/or ISRs, which access the same shared resource, it will possibly generate unnecessary mutual exclusion constraints when analyzing the source code[2]. The authors propose that a program analysis, which generates a memory access pattern, should be performed on the system. This way, by analyzing the memory locations that might be accessed for each critical section, a set of accessed memory locations is generated for each critical section. If two critical sections have totally different sets of accessed memory, it is safe to state that there is no mutual exclusion constraint on these critical sections [31].

As mentioned in section 2.1.2, AUTOSAR does not allow tasks to migrate between cores. This is disadvantageous whenever dynamic load balancing is needed. In general, a balanced processor load is needed since all cores in the system need to

[2] An example of this is described in Appendix C.1

handle resource usage peaks [24]. For example, if the system consists of a dual-core processor, it is important to distribute the workload evenly on each core. If a particular piece of the application software exceeds its expected execution time, there is still some margin left for the software to finish its execution time without missing any deadlines. Additionally, this margin can also make it possible to add more SWCs for e.g., fault tolerance or security functions.

Also, if there are data dependencies between communicating runnables, they cannot execute in parallel, thus having a negative impact on the level of parallelism of the software application [21]. Additionally, given that the desired level of parallelism requires a new runnable-to-task mapping, it is important to validate both the functional correctness and timing behavior of the application.

4.3 Runnable-to-task Mapping and Task Allocation

As mentioned in section 2.3.1, each SWC consists of small pieces of functionality called runnables. The SWCs are used when constructing a logical view of the system, abstracting the exact details of e.g., communication. Moreover, each runnable needs to be mapped to a task. This must be done with several requirements in mind, in order to avoid inefficient mappings. Consider the case where one runnable sends data to multiple runnables. If each runnable is mapped to a different task, task switching occurs for each receiving runnable. Such overhead can be avoided by mapping those runnables to the same task [23]. Extensive amounts of cross-core communication is not desirable, but cannot be completely removed if the application software is parallelized [19] or if the system uses the default AUTOSAR multi-core BSW i.e., consists of a BSW core and application core(s) [15].

There are some solutions on how to parallelize the application software, as shown in figure 4.1. Faragardi et al. [18] propose a solution where both the runnable-to-task mapping and task allocation are considered when minimizing the communication cost. They argue that this is needed, since a set of ill-formed tasks (i.e., runnables have been mapped to tasks without considering communication cost) might result in bad system performance despite having an optimal[3] task allocation and vice versa [18]. Three different solutions[4] are analyzed, and their respective benefits and disadvantages are furthermore discussed. Each of the solutions focus on minimizing the communication cost in the system. One of the solutions, called PUBRF, provides both reduced communication cost and CPU utilization [18] and seems promising. However, due to the fact that these solutions allow different tasks to run in parallel, this requires extensive post-migration analysis, in order to detect if any e.g., precedence constraints have been violated, which is a time-consuming task [28] [24]. Another solution, proposed by Panić et al. [28], instead focuses on parallelizing indi-

[3]With respect to minimal communication cost.

[4]Explained in Appendix D

vidual tasks while preserving the precedence constraints. This solution is explained below.

4.3.1 RunPar

Previous research suggests that dependent runnables should be mapped to the same task, with the goal of minimizing inter-task communication. This is argued by Panić et al. [28] to be a disadvantageous strategy for two reasons. First, if the legacy application software consists of highly connected runnables (i.e., high amount of inter-runnable communications exist in the system), then most runnables will be mapped to the same task. This task is then executed serially on one core. Thus, for applications where runnables are inter-connected with each other at a high degree, this approach will affect the level of parallelism negatively. Second, allowing different tasks to execute in parallel might cause a violation of the e.g., mutual exclusion constraints that exist in the system. Such mutual exclusion constraints are hidden in the legacy application software, since tasks execute sequentially on a single-core platform.

Panić et al. propose a solution called RunPar, suitable for migrating legacy software applications, where runnables from *the same task* are allowed to run in parallel. This is done to preserve runnable dependencies (e.g., runnable A needs to finish its execution before runnable B is allowed to start its execution). Their approach can be summarized with the following example[5]: Consider an AUTOSAR application, running on a single-core system. The application consists of three SWCs as shown in figure 4.2. The arrows depict the dependencies between the runnables e.g., runnable r_1 *must* finish its execution before r_2 and r_3 can start executing due to data dependencies. The runnables are mapped to tasks based on their periods, thus the tasks have periods which correspond to their runnables.

[5]This is a simplified version based on the example provided by Panić et al. [28].

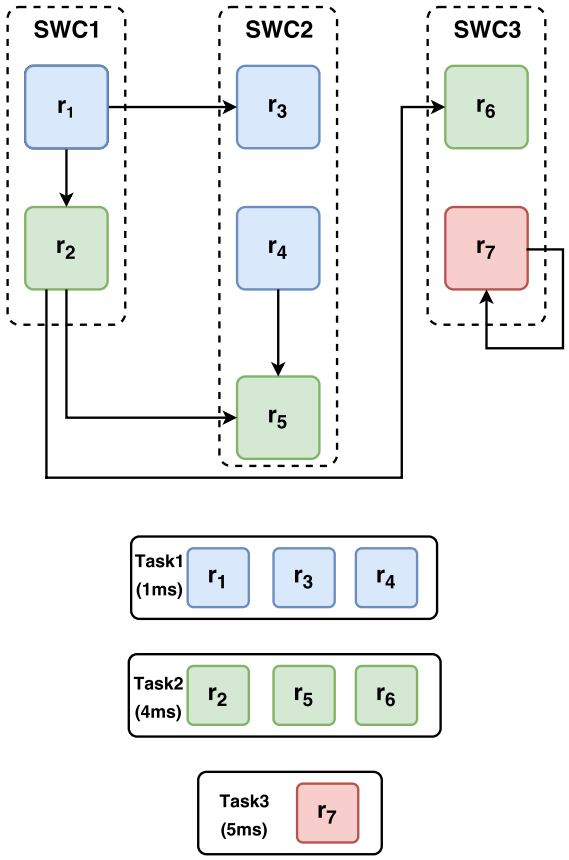

Figure 4.2: Dependency flowchart and runnable-to-task mapping of the RunPar example. Based on the example from Panić et al. [28].

Using the RunPar algorithm on this application generates the task allocation showed in figure 4.3. Each runnable is considered to be the unit of scheduling and the execution of runnables is governed by a scheduling table. This means that runnables are scheduled according to the entries in the schedule table, which contains a starting point of execution, the order in which runnables are executed and on which core they execute. This approach is different than what is suggested by AUTOSAR, where tasks are the unit of scheduling and not runnables [28]. Thus, RunPar does not only perform the task allocation process, but also assumes that runnables are the unit of scheduling in the system

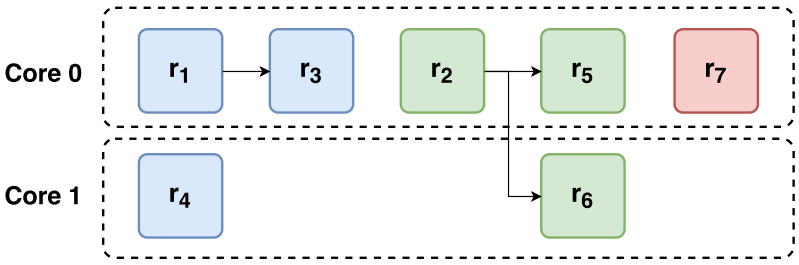

Figure 4.3: A valid allocation of the runnables in the example.

As shown in the figure, only runnables from the same task are allowed to execute in parallel. Moreover, by viewing the run-after dependencies shown in figure 4.2, the dependencies are still respected using RunPar. This means that the WCET of each task could be reduced, but not the WCET of the runnables. Panić et al. evaluated the WCET speed-up of tasks and concluded that a WCET reduction of approximately 26% could be obtained for a dual-core ECU [28].

5

Migration in Practice

This chapter describes our experiences, in approximately chronological order, when migrating the software to make use of the second core of the DACU. We aim to describe the process without too many engineering details, while still being informative.

5.1 Running the OS on the Second Core

The starting point was a software stack based on Vector's implementation of AUTOSAR 4.1[1], coupled with Volvo's integration code and SWCs. At this point, the OS and application software were executing correctly on the first core (*core 0*), and the second core (*core 1*) was powered on, but in a sleeping state. The CPU load on core 0 was approximately 20%. The integration team at Volvo had not yet attempted to start up the OS and run application software on the second core. Thus, we were the first to try code execution on the second core.

Volvo's supplier of the AUTOSAR stack had provided an AUTOSAR-compliant OS with multi-core support. We quickly realized, however, that the multi-core support was not complete. Parts of the source code had comments explicitly stating that running the OS on the second core gave "unstable" results. The supplier had left some code which was relevant for starting the second core. We used some of this code to try to start the OS on the second core, but ended up spending significant amounts of time debugging numerous problems related to the boot-up process of the OS. Most of these problems were related to the memory protection, caching, OS boot synchronization and mutually exclusive shared memory access. For readers interested in the technical details, we have described some of the problems and their solutions in Appendix B.1.

We encountered two types of problems during this process: Firstly, situations where the error-handling of the OS caught the error, and secondly, crashes which caused a disconnection between the debugger and the ECU. The first type was easier to debug, since we could use the *Embedded Trace Buffer* (ETB) of the ECU to trace

[1] MICROSAR 4 R16

the execution leading to the error. The trace functionality is further explained in section 5.2.2.

The second type of crashes, those causing a loss of connection with the hardware, were debugged by using a strategy based on isolating which part of the code that was running when the crash occurred. This was done by finding a starting point where we knew that the program was executing correctly and setting a breakpoint there. We then went from there, either stepping through the program manually until we found the location of the crash, or arbitrarily setting another breakpoint in a "later" part of the code. If the crash occurred before this arbitrary breakpoint, we moved the breakpoint closer to the starting point until we got close enough to step manually.

The main workaround we did to get the OS working, as a first step, was to disable both cache levels of the CPU. However, disabling caching would ultimately result in an unacceptable reduction in performance. At this point, we discovered that we had severe cache coherence[2] issues between the cores, which was why it helped to disable caching altogether. After finding out about the cache coherence problem, we found a way to move the affected variables to an uncached region of the memory. This solved the problem, and allowed us to enable the caches again.

We considered this phase of the migration to be finished when we had an idle loop task executing on both cores. Configuration-wise, we made sure the configuration of the second core was as simple as possible, meaning that only the SchM (Schedule Manager) and an idle task were allocated to the second core. Using this minimal configuration enabled us to focus on getting the OS to work before we even tried to migrate any BSW modules or application software to the second core. After booting the OS on both cores, the next step was to verify that the software was running normally on the first core. Even though each core runs its own instance of the OS, there are numerous dependencies and shared hardware resources between the cores. This means that until we could successfully boot the OS on the second core, there was no way of knowing the state of the application software execution on the first core. Once the OS was running on the second core, some of the workarounds we implemented to get the OS to run on the second core ended up breaking various things in the application execution, further explained in Appendix B.1.

5.2 WCET Estimation

After having the OS running on both cores, and the application software was supposedly running as before on the first core, we decided it was time to find a way to measure execution times, preferably on runnable-level. As mentioned in section 2.1.1, determining the WCET can be done by either measurement or analysis. There are several tools for estimating the WCET analytically, such as aiT [1], RapiTime

[2]Explained in section 2.2.1.

[29] and Avelabs Artia [13]. Such tools must, however, support the target processor and compiler. Currently, none of those tools support the ARM Cortex-A9. An analytical approach for determining the WCET would be the ideal solution, as previously mentioned in section 2.1.1. There are additional tools for determining the WCET, but in general such tools do not support the ARM Cortex-A9 nor the compiler, or the tool requires adaption to the CPU target (i.e., does not work off-the-shelf). This adaption means creating a very detailed model of the system, including the CPU core with its pipeline and memory hierarchy. To the best of our knowledge, no analytical approach of determining the WCET is feasible for the current platform given the time frame of this project.

After ruling out an analytical approach, we decided to evaluate some measurement methods. The measurement would then be repeated a number of times. Once enough data was collected, the worst of these runtimes would be used as a basis for the WCET of the runnable. Finally, a margin would be added to the worst case, providing a reasonable WCET estimation. It is not only important how we measure execution time, but also *what* is being measured. Therefore, we wanted a realistic workload for benchmarking and execution time measurements, since having access to real-world automotive software gives more value to our research. When just running the Volvo software on the DACU in the debugger, we have a basic workload consisting of all tasks running idle, and also sending and receiving some data over CAN. One of the testers at Volvo helped us to set up a test scenario executing in real-time. This scenario involves the adaptive cruise control and emergency braking features of the active safety platform. Unfortunately, the software did not execute the test cases as expected after the modifications, which we made to get the second core running, so we decided to proceed without a synthetic workload. In the following subsections, we will explain which measurement method we decided to proceed with, and why.

5.2.1 RTE Hooks

First, we considered adding run-time environment (RTE) hook functions before and after calling the runnables in the software. Hook functions are user-implemented functions invoked by the OS, and usually used for diagnostics and runtime monitoring [10]. This technique was used in an older project at Volvo, where they wrote hooks that used timestamps to measure the worst-case runtime. We went as far as to try porting these hooks to our platform, but we quickly realized that they did not use any standardized software or OS APIs, which made it very challenging to port. We decided to spend our time on more sophisticated measurement methods instead.

5.2.2 Tracing or Sampling

After discarding the RTE hook method, we found another option for estimating the WCET by measurement using the Trace32 debugger to measure execution times. There are many different methods to measure performance and execution times in Trace32. These can be divided into three categories - on-chip tracing, off-chip tracing, and sampling. The tracing methods provide exact execution times down to the resolution of the hardware timer, but require either on-chip memory or off-chip memory combined with a high-speed physical debugging interface. The trace procedure saves the entire execution sequence with timestamps for every state change, making offline analysis of any part of the captured execution possible. The DACU has a small on-chip trace memory that can be used with Trace32. However, this memory is far too small to capture data for a long enough time for us to get viable execution times. The size of the trace memory is crucial, since the maximum length of the trace is proportional to the available memory. As for the off-chip trace, we eventually found out that the DACU does not have a fast enough debug port to be able to use Lauterbach's PowerTrace 2, which is an extension to the PowerDebug debugger.

After ruling out the tracing method, we found out that it was possible to measure runtimes with Trace32 through sampling the Program Counter (PC). The trace module would then use the debugging interface to sample the PC, looking at which address it is pointing to, matching it to the compiled code to find out which function is currently executing. This method does not depend on separate trace memory, the data is instead streamed to the debugging host and stored there. There are two problems with this method - first of which is the fact that since it is sample-based, there is a hard limit on the accuracy of the measurement. Secondly, and most importantly, it does not record the execution path. This means that if runnables, which are defined as functions, call other functions, the runtime of the inner function will not be added to the runtime of the runnable function. In other words, this method only works for measuring the innermost function of any piece of software. When we realized this, we dismissed the sampling method altogether.

5.2.3 Vector RTM

After realizing that there was no way we could use Trace32 for runtime measurement, we considered Vector's primary tool for runtime measurement, which is an extension of CANoe. This tool includes a Run-Time Measurement (RTM) option, which allows for measurement via XCP messages through a CAN bus. XCP is a network protocol used for connecting measurement and calibration equipment to an ECU. XCP is transport layer-independent, and works over CAN as well as LIN, FlexRay, Ethernet and USB [34].

By using the RTM measurement points, a specified start and stop time can be measured for a particular time window, such as runnable runtime. These measurement

points can either be inserted manually, or configured to be inserted automatically by the code generator. To be able to use the RTM, we had to add and configure two BSW modules, XCP and RTM. To help us with the configuration, we used a reference software project from Volvo, which had the XCP and RTM modules set up. In order to use RTM on the DACU, we also had to find a way to insert measurement points in the code. It is possible to manually insert measurement points on runnable-level in DaVinci Configurator. However, this would have to be done for every runnable, so we decided to write a script that automatically patched the generated code to create and place the measurement points.

In parallel with working on RTM, we were trying to migrate the ASW to the second core. Unfortunately, we never got to a point where we could test the RTM integration, due to the CAN communication not working properly. In theory, the RTM should have been working at this point, since we configured and integrated the required BSW modules according to Vector's instructions, and inserted measurement points according to the RTM API. RTM was the last runtime measurement method we tried, and since it failed, we never managed to perform any measurements. We will discuss the implications of this in the discussion chapter.

5.3 Configuration of BSW and Application Software

The configuration of the BSW core approach proved to be more challenging than anticipated. The main reason was that the OS was found to have unexpected limitations. In theory, configuration of a BSW core should be easy, since we just have to leave all BSW modules on one core, move the application to the other core, and enable IOC. To move the application SWCs to the second core, we chose an approach where we copied the *OS Application* on core 0, assigned it to core 1, and then removed the duplicate elements on both cores respectively. The reason we chose this approach was that it is not enough to just move the runnables, which can be done with drag-and-drop in the configuration tool, but the runnables are dependent on *alarms* and *events* as well. In the AUTOSAR OS, events are used to trigger a list of actions, and alarms are simply events that are triggered periodically.

When changing the configuration, the tool makes a preliminary consistency check in real-time, meaning that the user will receive some error messages and warnings without needing to start the code generation process. However, the real-time check only catches some errors. This migration turned out to be a long and error-prone process. Through trial and error, we eventually realized that the BSW had unexpected limitations, of which the most important ones are listed in table 5.1. The *AUTOSAR* column in this table denotes whether we were able to find this restriction in the AUTOSAR documentation. The *BSW Docs* column denotes whether we were able to find it in the documentation for the BSW. The *Severity* column indicates to what degree the restriction/deviation affected our migration.

Some of these limitations proved quite challenging to work around, most notably mode-switching. In this context, *mode* means a user-defined mode used to control the state of ECU software execution, e.g., *shutdown*, *sleeping*, *wakeup*, *running*. Using the mode functionality is not a requirement, but Volvo's application is using several modes. Since the BSW does not support cross-core mode switching, we had two choices - either try removing the mode dependency altogether, or "faking" the mode switching on the non-BSW core. After consulting some senior engineers at Volvo, we chose to create a dummy mode switching component that eventually switches to the *running* mode after start-up. This is not the most sophisticated nor sustainable solution to the problem, but for the sake of this thesis, we figured it would be good enough.

Restriction	**AUTOSAR**	**Vector Docs**	**Severity**
Entire SWC must be on same core	No	Yes	High
Mode-triggered runnables to same task	No	No	Low
No cross-core mode switching	No	Yes	High
No cross-core unmapped runnable calls	No	Yes	Medium
Limited number of events per task	No	Yes	Low

Table 5.1: Discovered limitations in the BSW.

Another critical component needed to run multi-core AUTOSAR is the IOC[3]. We tried to find information and recommendations on how to configure the IOC, but found only basic documentation from Vector. We came up with a basic configuration where we simply guessed some of the values. At that point, we had no way to test whether the IOC was actually working, since the system was not yet running. When we had an error-free configuration, we discovered that the size of the generated code had doubled, forcing us to move the OS code to a different region of the OCM for it to fit. After achieving a valid configuration and moving the OS code, we could build the entire software without errors.

[3]Previously described in section 2.3.2.

5.4 Running the ASW on the Second Core

At this point, the OS was booting fine. However, the performance monitoring in Trace32 showed that the CPU was idle over 99 % of the time, meaning that nothing meaningful was executing. In AUTOSAR terms, no tasks were running, which we confirmed using a Trace32 extension made by Vector which can monitor the status of the tasks.

Using a series of workarounds and fixes which are explained in Appendix B.2 enabled us to put the ECU in a state where the steady-state (post OS boot) CPU load, defined as the inverse of the CPU idle time measured by Trace32, was around 15 % on each core. By using breakpoints, we could confirm that critical runnables related to CAN communication and the periodic active safety application runnables were running as expected. However, when we tried to use the Test Application to connect to the ECU, to have a closer look at whether the application was working, we found that the CAN communication was not working properly. This was the final attempt we did in this project. In short, we got to a point where it is very likely that the ASW was executing correctly. Since the CPU load on core 0, the BSW core, was around 15 % and the CAN communication stack was working, as well as the IOC, we concluded that the critical BSW modules were also functional.

6

Discussion

In this chapter, we will discuss our experiences and findings, with the primary focus on the differences between theory and practice in the area of migrating automotive real-time systems to multi-core hardware. Chapter 4 serves as a basis for discussion when comparing theory with practice. Section 6.1 describes the various implementation challenges we encountered and the lessons learned. Section 6.2 contains discussion on the attempts we made to estimate the WCET of the tasks and runnables in the system. Section 6.3 contains discussions on differences between theory and practice. Some general suggestions on how application software should be migrated are made here. Section 6.4 describes some of our observations on practices within the industry, and how these can be improved. Finally, section 6.5 contains our general suggestions for future work within the area of multi-core migration.

6.1 Implementation Challenges

Originally, this thesis focused on runnable-to-task mapping, task allocation, task scheduling and communication cost in multi-core AUTOSAR. When we started the thesis, it was assumed that the multi-core OS was in a working state, meaning that it would require little effort to run the OS on the second core as well. This, however, was not the case. Looking back, most of our time was spent working on making the OS run, without severe performance penalties, on both cores of the DACU. It is worth mentioning that the issues with the multi-core OS from Vector have been largely unknown to Volvo, due to the fact that multi-core AUTOSAR is not yet used in production. It is also possible that the problems we encountered have been resolved in more recent releases. After months of work, upon realizing that we did not have enough time left to finalize the migration, we found an enlightening quote by Schneider et al.

> Manually migrating existing legacy software to multi-core platforms in this industrial setting is practically infeasible, because it would be highly error-prone, even if a host of skilled programmers were available [31].

This is, naturally, the opinion of a few researchers doing research on automated migration methods. We do not agree with the assessment that it is practically

infeasible to perform manual migration. However, we do agree that it is highly error-prone and requires significant resources. Most previous research on multi-core AUTOSAR has been conducted using custom ASW written by the research team, as opposed to real-world automotive software. We have only found one research report where migration of AUTOSAR-like legacy software has been performed successfully, which is the work by Armengaud et al. [4]. Their results are similar to ours, except that they were able to verify correct post-migration operation of the software. They did not explore timing behavior, runnable-to-task mapping nor task allocation. It is of course possible that there have been successful migrations in the industry, that are not publicly available due to competitive reasons.

During the process of attempting to run the OS on the second core, we encountered several issues related to shared memory. As pointed out by other researchers, dealing with shared memory is one of the main challenges with multi-core migration of legacy software [24]. We later realized that some of these issues could have been prevented if we would have read this research in the beginning of the project. Nevertheless, handling shared resources and solving those issues were an important learning outcome.

During our work, we realized that migration of single-core AUTOSAR applications to a multi-core system will be very important to the industry in the near future, because of the rapidly increasing amount of complex software and increased demands on safety and security. Starting out, we did not have enough knowledge about the automotive industry to understand the challenges when migrating legacy software to a multi-core environment. After reading about AUTOSAR very early in the project, we assumed that AUTOSAR conformity meant that migration would be relatively simple, and that extracting parallelism and minimizing cross-core communication cost would be the major challenges, which was indicated by previous research in the field. In retrospect however, we understand why most of that research was conducted using application software designed by the researchers, as opposed to real-world automotive software designed to run on single-core ECUs.

6.2 WCET Estimation and Performance Profiling

Performance profiling and timing analysis of the system were difficult to perform due to lack of support for our platform. We were initially planning to make a thorough theoretical analysis of the real-time properties of the software. Determining the WCET of the runnables should ideally be done analytically, as proposed by Wilhelm et al. [36]. Additionally, using an off-the-shelf tool appears to be preferred in research to determine the WCET of runnables or tasks [28] [24]. However, off-the-shelf tool support for analytical WCET estimation on our target hardware proved to be non-existent as far as we could find, as described in section 5.2.

Thus, we evaluated several measurement-based methods which, at the very least, could give us a general idea of the system performance. The RTM measurement

method was always our first choice, since it officially supports our platform, and that it has support from Vector. We later found out that it is possible to insert measurement points manually, which was why we developed the script for inserting measurement points. The script makes it easy to add new functionality to the ASW and re-evaluate the system performance. The fact that RTM only has support for CPU load and task-level runtime measurement out of the box is disadvantageous, because in many cases, task-level is too coarse to be able to evaluate the real-time performance of the system.

It is regrettable that we did not manage to estimate the WCET on any level. However, since we never had a fully working system running on two cores, any measurements made would have been pure speculation as to the performance of the system. We considered integrating the RTM functionality on a stable branch of the software, running on one core. However, we prioritized trying to make the system run on two cores over measuring the single-core software.

6.3 Differences Between Theory and Practice

There are some noteworthy differences between the theoretical migration process and doing it on a real system. In this section, we will discuss the different constraints on the DACU in relation to the migration strategies presented in chapter 4.

6.3.1 Multi-core BSW

Given the different multi-core BSW solutions presented in section 4.1, BBL appears to be promising in the sense that most of the integration work is done by the RTE generator (i.e., not much manual adaptions are required). The fact that each core has its own copy of the BSW allows for flexible runnable-to-task mapping, since each core has access to the entire BSW. Thus, the runnable-to-task mapping and task allocation will not depend on which BSW modules that exist on a particular core. Moreover, the other methods will either affect the runnable-to-task mapping and task allocation, or will be rather time-consuming to implement. Although being promising, BBL still needs to be evaluated on real-world automotive applications. Additionally, BBL should be evaluated when using different runnable-to-task mapping strategies and task allocation algorithms, in order to fully investigate its potential.

6.3.2 Runnable-to-task Mapping & Task Allocation

The observed limitations in the OS meant that we could not freely assign runnables to tasks, which meant that it would not have been possible to evaluate the algorithms

and strategies presented in chapter 4.3, even if we had more time. However, with the observed restrictions in mind, we were able to analyze some of the algorithms and strategies for runnable-to-task mapping and task allocation, in order to find which strategies that might be suitable for our system in future work.

The RunPar algorithm seems to be promising when migrating legacy software to a multi-core system, due to the fact that the algorithm preserves the precedence constraints and efficiently parallelizes the software. However, this algorithm cannot be used without modifications in Volvo's system, due to e.g., the restriction that all runnables from the same SWC must be allocated to the same core. With this restriction in mind, the example in section 4.3.1 would not be a viable allocation, since runnables from the same SWC are allocated on different cores. Moreover, RunPar is designed for applications which have non-preemptable tasks. This is not the case for Volvo's application software, where some tasks are preemptable.

However, the idea of using runnables as the unit of scheduling, while trying to parallelize each task with dependencies in mind, and including the aforementioned restriction could still be a viable solution. The only problem is when a task consists of runnables, of which at least two belong to the same SWC. Then, the runnables which belong to the same SWC must be allocated to the same core. This limits the level of parallelism if the number of runnables belonging to the same SWC are relatively large. Thus, given the aforementioned restriction of our system, the RunPar algorithm should be extended and further evaluated. It would also be relevant to evaluate the cross-core communication cost of using RunPar, as minimizing communication cost is one of the main goals when parallelizing application software [18] [24].

As described in section 4.3, there are some strategies and algorithms for minimizing the communication cost. While these solutions preserve inter-runnable dependency constraints, it is important to note that tasks may run in parallel using these solutions. As mentioned by Panić et al. [28] as well as Macher et al. [24], this requires a re-validation of the application. While this can be quite time-consuming, it is still worth to evaluate the possibility of using e.g., a modified version of PUBRF. The constraints described in section 5.3 should be taken into account when doing this evaluation.

In summary, while minimizing the communication cost still is a relevant goal when migrating ASW to a multi-core system, the various constraints in the current system must be taken into account. An application profiling should be done, with the goal of highlighting e.g., handling of mutual exclusion and precedence constraints. Moreover, as suggested by recent research, runnable-to-task mapping and task allocation should in general be considered simultaneously and not separately. This is a reasonable approach and appears to be more common within research in AUTOSAR. Moreover, as shown in previous research, there are at least four goals when considering multi-core migration in runnable-to-task mapping and task allocation, namely: reducing the communication cost, preserving legacy mutual exclusion constraints, timing constraints and precedence constraints [18] [28] [31]. The aforementioned

solutions for runnable-to-task mapping and task allocations should be further evaluated with these goals in mind.

6.4 Industry Practices

During this thesis, we have identified some industry practices which ultimately can result in problems. One of these is the usage of functional tests as the only means to verify that the system is working as intended. Testing is of course extremely important, especially for safety-critical systems, but combining it with performance profiling and timing analysis would enable vehicle manufacturers to be more proactive in solving problems related to performance [24]. Additionally, there should not be a tendency within the industry to keep adding functionality to an ECU until the tests will not pass and then start looking for performance optimization.

Furthermore, we observed the tendency to prioritize time to market over cost reduction when choosing ECU hardware. Even though we did not focus on the financial aspects in this thesis, it is worth mentioning that the Zynq 7000 offers much higher performance than needed. An interesting example of this is that, in the process of analyzing the start-up code of the OS, we found that a well-known hardware-level optimization called *branch prediction* was not enabled for the first core. Even though this was unrelated to executing code on the second core, it was found that enabling this feature significantly increased the performance of the application when running normally on the first core. This is, essentially, ”free“ performance which has no negative impact on reliability, safety nor security, unless there is a bug in the hardware.

6.5 Future Work

As mentioned previously, the different solutions for multi-core BSW, runnable-to-task mapping and task allocation should ideally be evaluated on a real-life system. It is worth noting that the limitations discovered during the configuration of the OS are specific to the particular vendor. These limitations may or may not apply for other AUTOSAR OSs.

It remains as a future work to further evaluate the different multi-core BSW approaches that are suggested. BBL is one of the promising solutions, since it is relatively simple to implement and shows good performance gain when having a higher workload on the system. If goals such as time to market are included as well, we strongly believe that this solution should be further evaluated on Volvo's system.

Moreover, the RunPar algorithm presented by Panić et al. [28] should be further extended to include preemptable tasks and the OS constraints mentioned in section

5.3. This extended version of RunPar could then be implemented and used in Volvo's system, as a first step of parallelizing software. This way, the risks of multi-core migration of legacy software[1] can be mitigated by using the conservative approach of running runnables from the same tasks in parallel (while still respecting the dependency constraints). The combination of BBL and RunPar could be a viable combination and should definitely be evaluated further in the future. At the very least, we consider this to be a conservative approach which could be used as a baseline when considering more sophisticated solutions e.g., PUBRF, for runnable-to-task mapping and task allocation where the goal is to minimize the communication cost.

[1] E.g., violating precedence constraints, mutual exclusion constraints

7

Conclusion

In the beginning of this thesis work, we set out to explore different ways to use multi-core hardware to increase the performance of an automotive real-time system based on AUTOSAR. In the end, the focus changed to the process of migrating the single-core legacy software to a multi-core platform. In the initial literature study, we focused on methods, processes and algorithms that would enable us to make the system perform better with two cores instead of one. There were, however, limitations in the OS which made it difficult to evaluate these solutions on the system. Thus, the focus of this thesis was changed to investigate the multi-core migration process and how deviations from the AUTOSAR standard affect the theoretical solutions.

In this thesis, we attempted a legacy software migration on a dual-core automotive ECU. We tried to implement the most basic approach to multi-core AUTOSAR, called *BSW core*, meaning that we use one core for BSW and the other for ASW. During this process, it was found that the multi-core support in the OS was not fully implemented, which significantly slowed down our progress. After fixing the OS, we managed to migrate the ASW to the second core, but in the end we could only verify that both the BSW and ASW were running, and that the inter-core communication was working. We were not able to verify correct operation of the active safety application originally running on the ECU.

From previous research, we identified the need to consider both the runnable-to-task mapping and task allocation. The proposed solutions for these steps in the multi-core migration process aim to minimize the communication cost and/or parallelize the application software such that mutual exclusion constraints are still preserved. With this thesis, our main contribution is showing the importance of having an AUTOSAR software stack that fully complies with the AUTOSAR standard. This is important since many algorithms and solutions for e.g., runnable-to-task mapping assume that the software stack fully complies with the AUTOSAR standard. We have shown that deviations from the standard negatively affects the possibility of using such solutions. Much of the previous research on multi-core AUTOSAR assumes that the BSW, including the OS, is fully compliant, which turns out not to be the case in the industry. We have also shown the importance of, and challenges related to, OS and application profiling. In our case, neither had been done beforehand, which hindered the progress of our work. We made significant efforts to analyze the real-

time properties of the system, but tool support proved to be weak or non-existent.

Due to time limitations and observed deviations in the OS, we could not implement and evaluate the proposed algorithms for runnable-to-task mapping and task allocation. With these deviations in mind, it was concluded that a conservative approach should be used as a first step. We propose that the BBL approach by Böhm et al. [15] should be used as a multi-core BSW solution in the future. Furthermore, the idea of letting independent runnables from the same task run in parallel, used in RunPar by Panić et al. [28], should be combined with BBL. This way, the amount of effort and risks with multi-core migration will be relatively low, since the legacy program flow is preserved. At the very least, this approach could serve as a baseline when evaluating more sophisticated methods which e.g., aim to lower the communication cost.

The transition from single- to multi-core ECUs in the automotive industry is inevitable. As road vehicles are becoming increasingly automated, and more than just a means of transportation, the amount of software will keep increasing. Multi-core computing is one of the primary ways to reduce cost and add more functionality to the system, which is why the industry needs to embrace it.

Bibliography

[1] AbsInt. ait wcet analyzer. https://www.absint.com/ait/. [Online; accessed 20-March-2017].

[2] ARM. Arm synchronization primitives. http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf. [Online; accessed 11-April-2017].

[3] ARM. In what situations might i need to insert memory barrier instructions? http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka14041.html. [Online; accessed 20-march-2017].

[4] Armengaud, E., Mustedanagic, I., Dohr, M., Kurtulus, C., Novaro, M., Gollrad, C., and Macher, G. Migration of automotive powertrain control strategies to multi-core computing platforms–lessons learnt on smart bms. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)* (2016).

[5] AUTOSAR. Autosar overview. http://www.autosar.org/fileadmin/images/media_pictures/AUTOSAR_Basic_Approach.jpg. [Online; accessed 24-April-2017].

[6] AUTOSAR. Background. http://www.autosar.org/about/basics/background/. [Online; accessed 2-february-2017].

[7] AUTOSAR. Glossary. https://www.autosar.org/fileadmin/files/standards/classic/4-0/main/auxiliary/AUTOSAR_TR_Glossary.pdf. [Online; accessed 15-May-2017].

[8] AUTOSAR. Technical overview: Autosar. https://www.autosar.org/about/technical-overview/. [Online; accessed 9-May-2017].

[9] AUTOSAR. Virtual function bus: Concept. https://www.autosar.org/about/technical-overview/virtual-functional-bus/concept/. [Online; accessed 9-May-2017].

[10] AUTOSAR. Specification of rte. http://www.autosar.org/fileadmin/files/standards/classic/4-0/software-architecture/rte/standard/

AUTOSAR_SWS_RTE.pdf, 2011. [Online; accessed 30-January-2017].

[11] AUTOSAR. Autosar – guide to multi-core systems. http://www.autosar.org/fileadmin/files/standards/classic/4-1/software-architecture/general/auxiliary/AUTOSAR_EXP_MultiCoreGuide.pdf, 2013. [Online; accessed 2-February-2017].

[12] AUTOSAR. Specification of autosar os. https://www.autosar.org/fileadmin/files/standards/classic/4-1/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf, 2014. [Online; accessed 10-April-2017].

[13] Avelabs. Artia. http://www.avelabs.com/products/artia.php. [Online; accessed 20-March-2017].

[14] Becker, M., Dasari, D., Nélis, V., Behnam, M., Pinho, L. M., and Nolte, T. Investigation on autosar-compliant solutions for many-core architectures. In *Digital System Design (DSD), 2015 Euromicro Conference on* (2015), IEEE, pp. 95–103.

[15] Böhm, N., Lohmann, D., Schröder-Preikschat, W., and Erlangen-Nuremberg, F. A comparison of pragmatic multi-core adaptations of the autosar system. In *7th annual workshop on Operating Systems Platforms for Embedded Real-Time applications* (2011), pp. 16–22.

[16] Cygwin. Cygwin homepage. https://www.cygwin.com. [Online; accessed 2-February-2017].

[17] Dubois, M., Annavaram, M., and Stenström, P. *Parallel computer organization and design*. Cambridge University Press, 2012.

[18] Faragardi, H. R., Lisper, B., Sandström, K., and Nolte, T. A communication-aware solution framework for mapping autosar runnables on multi-core systems. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE* (2014), IEEE, pp. 1–9.

[19] Faragardi, H. R., Lisper, B., Sandström, K., and Nolte, T. An efficient scheduling of autosar runnables to minimize communication cost in multi-core systems. In *Telecommunications (IST), 2014 7th International Symposium on* (2014), IEEE, pp. 41–48.

[20] Gustafson, J. L. Reevaluating amdahl's law. *Communications of the ACM 31*, 5 (1988), 532–533.

[21] Kehr, S., Panić, M., Quiñones, E., Böddeker, B., Sandoval, J. B., Abella, J., Cazorla, F. J., and Schäfer, G. Supertask: Maximizing runnable-level parallelism in autosar applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016* (2016), IEEE, pp. 25–

30.

[22] Krishna, C. M. *Real-Time Systems*. Wiley Online Library, 1999.

[23] Long, R., Li, H., Peng, W., Zhang, Y., and Zhao, M. An approach to optimize intra-ecu communication based on mapping of autosar runnable entities. In *Embedded Software and Systems, 2009. ICESS'09. International Conference on* (2009), IEEE, pp. 138–143.

[24] Macher, G., Höller, A., Armengaud, E., and Kreiner, C. Automotive embedded software: Migration challenges to multi-core computing platforms. In *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on* (2015), IEEE, pp. 1386–1393.

[25] Monot, A., Navet, N., Bavoux, B., and Simonot-Lion, F. Multicore scheduling in automotive ecus. In *Embedded Real Time Software and Systems-ERTSS 2010* (2010). [Online; accessed 10-April-2017].

[26] Morse, D., Frysinger, M., McGrath, R., and Smith, P. make man page. `http://man7.org/linux/man-pages/man1/make.1.html`. [Online; accessed 2-February-2017].

[27] OSEK. Osek specification. `http://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf`, 2005. [Online; accessed 10-April-2017].

[28] Panić, M., Kehr, S., Quiñones, E., Boddecker, B., Abella, J., and Cazorla, F. J. Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis* (2014), ACM, p. 29.

[29] Rapita Systems. Rapitime. `https://www.rapitasystems.com/products/rapitime`. [Online; accessed 20-March-2017].

[30] Schmerler, S. Autosar – shaping the future of a global standard. `http://www.autosar.org/fileadmin/files/papers/AUTOSAR-BB-Spezial-2012.pdf`, 2012. [Online; accessed 2-February-2017].

[31] Schneider, J., Bohn, M., and Rössger, R. Migration of automotive real-time software to multicore systems: First steps towards an automated solution. In *22nd EUROMICRO Conference on Real-Time Systems* (2010).

[32] Stankovic, J. A. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer 21*, 10 (1988), 10–19.

[33] Vector Informatik GmbH. Configuring autosar basic software with davinci configurator pro. `https://vector.com/vi_davinci_configurator_pro_en.html`. [Online; accessed 5-April-2017].

[34] Vector Informatik GmbH. Measurement and calibration protocol xcp. https://vector.com/portal/medien/solutions_for/xcp/Vector_XCP_Basics_EN.pdf. [Online; accessed 11-April-2017].

[35] Vector Informatik GmbH. Vector canoe product page. https://vector.com/vi_canoe_en.html. [Online; accessed 10-April-2017].

[36] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS) 7*, 3 (2008), 36.

[37] Xilinx. Zynq-7000 all programmable soc technical reference manual. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 2016. [Online; accessed 2-February-2017].

A

Appendix 1

A.1 Fine-grained Locks, Non-blocking Synchronization & Partitioning

Böhm et al. [15] mention three more methods to enable multi-core support in AUTOSAR. The first, called fine-grained locks, is a further improvement on BBL. Instead of locking the whole BSW, locks are added module-by-module. Thus, whenever a core accesses e.g., the BSW module responsible for CAN communication, the other cores may still access the other BSW modules. This needs to be done module-by-module of the BSW, and would be a very time-consuming task, thus the implementation cost would be significant.

The next method is called non-blocking synchronization, which consists of algorithms which synchronize shared data without blocking tasks on the other core [15]. Non-blocking synchronization such as wait-free algorithms, would make it easier to analyze the real-time properties of the system [15]. However, implementing these algorithms can be complicated, and they are dependent on atomic instructions, e.g., Compare-And-Swap (CAS). This makes a non-blocking synchronization approach less portable between different hardware platforms. For example, the ARMv7-A ISA does not support CAS, but instead has the Load-Exclusive and Store-Exclusive instructions, supposedly providing the same functionality but with lower interrupt latency [2]. Implementing non-blocking synchronization would require extensive modifications to the entire set of BSW.

Partitioning means running completely independent AUTOSAR systems on each core, and then partitioning all shared hardware resources, making the systems independent. The benefits of this approach is that no multi-core adaption is required and that real-time analysis can be performed as if the system was single-core. The obvious disadvantage of this approach is the overhead introduced by running the entire AUTOSAR stack on every core. In theory, this solution is not scalable at all, since the shared resource partitions will be smaller the more cores we have, but the resource overhead will be constant.

B

Appendix 2

B.1 Fixing the Multi-core OS - Engineering Details

Software-wise, our starting point was a software stack based on Vector's implementation of AUTOSAR 4.1, coupled with Volvo's integration code and SWCs. At this point, everything was executing correctly on the first core (*core 0*), and the second core (*core 1*) was in what we refer to as Wait For Event (WFE) mode. In this state, the core is waiting for a special event to "wake it up". It is essentially sleeping, not running any AUTOSAR/Volvo code. After getting the hardware and debugger up and running, we could start looking at how to execute code on the second core.

To get the second core to run AUTOSAR, we had to set up the stack pointer for the second core manually, and using the API calls to start Vector's AUTOSAR OS on the second core. Next, an idle task was added and configured to run on core 1. During OS boot, we had numerous *bus errors*. A bus error means that the hardware has entered a state where the debugger can not communicate with the CPU, which makes it challenging to debug. We traced the problem to an instruction performing a write operation on a hardware configuration register, enabling the MMU (Memory Management Unit). When further debugging this issue, we found a workaround that meant not initializing the core 1 MMU at all. This workaround enabled us to run the idle task on core 1, with core 0 also executing normally. The problem with this workaround was that disabling the MMU meant that both the instruction- and data-caches were disabled, which would have resulted in terrible performance if executing real software.

Disabling the caches was not an acceptable trade-off, so we continued the debugging, finally finding the root cause. When writing to the aforementioned configuration register, the CPU was in *user mode*, a mode where some hardware configuration registers are read-only. After some research, digging into the ARM Cortex-A9 reference manuals, we found out how to switch the CPU to *system mode*, to be able to correctly start the MMU.

Next, data and instruction synchronization barriers were used after initializing the

MMU in order to avoid a bus error. The data synchronization barrier ensures that any memory accesses before the barrier are completed before the barrier is passed. On the other hand, the instruction synchronization barrier is needed to make sure that the changes that are made before the barrier are visible to the consecutive instructions after the barrier. After including these barriers, no bus errors were generated and the MMU was enabled for core 1.

Next, we tried enabling all caches. This caused some new problems, most notably that Vector's mutex lock implementation, referred to as *MiniLock*, stopped working. Mutex locks are needed to synchronize access to shared memory, protecting the system from race conditions and deadlocks. We eventually found that cache coherence is not enabled for the whole memory on the Cortex-A9. Instead, the memory region where the lock variable resides needs to be marked as shareable in the MMU. This means that the memory is shared between the cores, thus there is a need for cache coherence. There was already a shared memory region set up by Volvo, so we modified the memory map so that the lock variable would be placed in that region, which fixed the problem. We also found a potential issue with the implementation of the MiniLock. It was essentially implemented as a semaphore, but did not comply with ARM's recommendations concerning usage of data synchronization barriers in the acquire and release functions [3]. We fixed this, which had the unexpected side effect that the compiler decided to make an "optimization", removing one of the function calls to release the lock. We therefore had to disable compiler optimization.

The MiniLock is used during OS boot, to grant mutual exclusion to the *osStartBarrier*, which is a barrier-style synchronization between the two cores. There are three barriers, where one core needs to wait for the other to catch up before resuming the boot process. After fixing the caches, we ran into an intermittent problem where these barriers did not work properly. Until this point, most of the problems we had were completely deterministic and quite easy to reproduce. This problem, however, could be circumvented by using the debugger to manually step through each instruction in the barriers. At first, that made no sense to us, but after some help from a senior software integrator at Volvo, we found that the problem was caused by the two cores sharing the same call stack. Because we had copied some hardware initialization code from the first core, the stack pointer was initialized to the same address on the second core. This caused a race condition on the stack, resulting in very unpredictable behavior.

After fixing the stack, the OS could boot on both cores. The second core was configured with only an idle loop, and was running flawlessly. The first core, which has Volvo's AUTOSAR tasks mapped to it, crashed on a cryptographic seed generation, because of a memory alignment problem. After some debugging, we decided to work around the problem by disabling the seed generation. For the purpose of this thesis, it does not matter if the code is safe, from a computer security perspective.

B.2 Running the ASW on the Second Core - Engineering Details

Once again, the problem was cache coherence, this time disrupting the initial activation of the tasks. At this point, we realized that moving critical parts of the OS to uncached memory was not a good solution, since we would have to keep moving variables, each one requiring extensive debugging. Instead, we found a way to modify the memory region configuration, enabling cache coherence in the L2 cache for the regions where all OS variables reside. Unfortunately, this had to be done in generated code, since Vector did not include the option of changing it in the configuration tool. In theory, it is possible to enable cache coherence in the L1 caches, which would yield better performance, but despite our best efforts we could not get it to work.

After enabling cache coherence, the tasks were activated as expected, but they were still not being scheduled to run periodically. We eventually found that this was due to the mode switching not working properly, which was no surprise considering the dummy mode switching component that we implemented ourselves was a quick fix. This was a typical example of a situation described in section 3.3, where finding a specific erroneous value does help in working around the issue, but finding the root cause was much harder. We did find the cause of the tasks not being scheduled, and by editing the memory manually in the debugger, we managed to get the tasks running.

However, this was not the only workaround required to get the ASW running. After enabling cache coherence, we had been trying to use Vector CANoe to connect to the ECU to be able to monitor the state of the ASW execution and try to get RTM working. Without CANoe, the only ways to confirm that the ASW was running was the CPU idle time as measured by Trace32 and breakpoints in critical runnables. While debugging the mode switching, we found two additional workarounds required to get the ECU into a state where the ASW is running. The first one was related to power cycling the ECU (i.e., turning it off and on again). We found that having a short (approx. less than 3 seconds) power cycle time put the ECU in a state where no tasks were being scheduled. After finding the workaround, we did not investigate this further, and our colleagues could not explain this behavior. The second workaround involved sending CAN signals through CANoe to get the ECU in the working state. We never found out what caused this behavior.

C

Appendix 3

C.1 Example of Mutual Exclusion Constraints

The following example, based on the example provided by Schneider et al. [31], illustrates the problem of using interrupt disabling as a means for achieving mutual exclusion. In a single-core system, there are two tasks (A and B) and one Interrupt Service Routine (ISR 1) as shown in figure C.1.

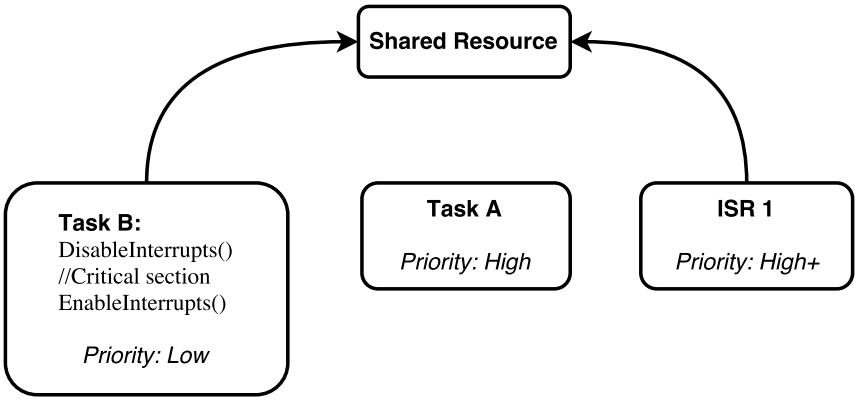

Figure C.1: An example of mutual exclusion constraints.

A has higher priority than B, while ISR 1 has higher priority (marked as High+ in the figure) than the tasks. Task B has a critical section, shown in the source code as a call to disable and enable interrupts around the critical section, to protect a resource that is shared with ISR 1. This will prevent any race conditions between task B and ISR 1. However, by simply analyzing the source code for Task B, the developers' intentions for using interrupt disabling cannot be determined (unless stated in the code). This means that, by merely analyzing the source code, we cannot exclude the *possibility* that there is a mutual exclusion relation between task A and B. This is due to the fact that disabling interrupts before Task B's critical section will not allow task A to run concurrently. Without knowing the developers' intentions, one must *assume* that there is mutual exclusion between A and B [31]. This problem can be resolved by using the memory access analysis explained in subsection 4.2.

D

Appendix 4

D.1 Simple Mapping

The first solution proposed by Faragardi et al. [18], called *Simple mapping*, performs the mapping and task allocation separately. The approach of considering these two steps separately is commonly used in previous literature [18]. The runnable-to-task mapping is performed by letting each inter-runnable transaction represent one task [18]. This means that if runnable A and B communicates by sending and receiving data, then the transaction itself is considered to be a task. Moreover, the authors state that the deadline for each of the constructed tasks corresponds to the deadline of the respective transaction. After constructing the task set, each task is statically allocated to one of the cores in the system. This is done by using an algorithm called *Systematic Memory Based Simulated Annealing* (SMSA). This algorithm can be used to find a task allocation such that all deadlines of the runnables are met. After allocating the tasks to the cores, the tasks are merged in a *refinement function* (REF). This is done by merging tasks located on the same core, if they have the *same* period and communicate with each other. This is done since each inter-task communication requires a context switch, which further adds to the communication cost. By merging the tasks which communicate with each other, the inter-runnable communication is done within the task context [18].

D.2 SMSAFR and PUBRF

The next solution, called Systematic Memory Based Simulated Annealing with Feedback Refinement (SMSAFR), is similar to the Simple mapping solution. However, the refinement is instead performed frequently within the task allocation step. This means that the task allocation step is performed with respect to the refined task set provided by REF. In other words, tasks are allocated to cores in such way that the merging performed by REF is considered beforehand. This way, any potential solution for which the communication cost is significantly reduced by REF can be considered.

The third solution, called PUBRF, is a further extension of SMSAFR. Instead of using REF as a way of merging tasks, they use a refinement function called Utilization-Based Refinement (UBR). This function merges tasks which communicate with each other, similarly to REF, but also merges tasks with *different* periods if they have intense communication between them. Thus, PUBRF merges task on the same core, given that the total CPU utilization can be reduced by this merge. PUBRF is also more effective in terms of execution time required to find a mapping and task allocation which minimize communication cost. Faragardi et al. [18] evaluated these three solutions, with communication time as one of the performance metrics. PUBRF outperformed both Simple Mapping and SMSAFR, providing the least communication time cost as well as providing better CPU utilization reduction. The authors also state that PUBRF should be scalable, although this was not proved in their paper.