1 | Introduction

> "Software gets slower faster
> than hardware gets faster."
>
> Niklaus Wirth, 1995

The performance requirements for an automotive embedded system have increased steadily in recent years and this also raised the number of *electronic control units (ECUs)* per car. Today, they contain a complex in-vehicle network of 70 or more ECUs [Für10], which realize thousands of control functions altogether. Each of these ECUs consumes electrical energy and this leads to a higher overall energy consumption of the car. Nevertheless, the demand for more functionality continues and this means more control functions must be integrated. This requires more computational power in the form of ECUs and it increases the overall electrical power consumption of the car. As a result, minimizing the resource consumption per car and per ECU is an important design objectives for automotive embedded system design [Mös10].

On the contrary, numerous control functions in a car must guarantee upper bounds on the response time to realize the desired functionality. This marks an essential difference between general purpose computing and automotive control software. For this reason, minimizing the latency of critical sensor/actuator paths is a typical optimization objective for automotive embedded systems [Nat+07]. An example for a critical sensor/actuator latency is the maximum time between pushing the gas pedal and the point in time when the driver recognizes the acceleration. The more control functions the *engine management system (EMS)* ECU has to compute (active cylinder management, exhaust gas recirculation, pre- and post-injection, etc.) the longer is the response time.

Increasing the processor's clock rate in the ECU was a first countermeasure to satisfy the increasing performance requirement. However, this solution leads to a higher electrical power consumption and heat dissipation, whereas the additional gained compute power is low. Moreover, this technique faces physical limits with the result that it is impractical to further increase the clock rate. The automotive industry is therefore searching for alternative hardware platforms that satisfy the performance and efficiency requirements.

Fortunately, embedded multicore processors, like the Infineon AURIX [Inf] or the Freescale Qorivva [Fre15], have become widely available for the automotive industry. These processors were designed for safety critical applications in the first place, but they also provide a surplus of computational power, in comparison to a single-core processor. This makes it possible to efficiently exploit *thread-level parallelism* of automotive control software.

Consequently, the ECU can either execute a more complex algorithm or execute the same application on multiple cores with a reduced clock rate, such that deadlines are still kept, to save electrical energy. Thus, multicore ECUs are seen as the hardware platform for current and future automotive control software in cars.

However, introducing multicore ECUs is a great challenge [MB09] for the automotive industry. Until now, software is developed and optimized for the execution on a single-core ECU. This software is well tested and much effort was spent on the optimization and maintenance. As a result, most existing (legacy) control software is supposed to be re-used, because the code is known to be reliable and the development process can be shortened. Consequently, the migration of automotive legacy software to multicore ECUs must be supported. This is a fundamental step for the adaptation of multicore ECUs in the automotive domain and for taking full advantage of these platforms. Consequently, this Ph.D. thesis focuses on the migration of automotive legacy control software to multicore ECUs.

Automotive control software is developed according to the *AUTomotive Open System ARchitecture (AUTOSAR)* standard [AUT14a]. The standard established an industry-wide understanding, a uniform development methodology, and a uniform terminology for automotive control software. A hierarchical *software-component (SW-C)* model describes the application according to the concept of a *virtual function bus (VFB)*. In this concept, *runnables*, i.e. elementary code pieces; a schedulable job, within SW-Cs implement the functional behaviour of the component. Each SW-C realizes a part of the overall control and runnables frequently communicate with each other. Executing the runnables in the correct order and frequency realizes the control function.

However, multicore ECUs have significantly different properties than general purpose multicore processors and automotive control software strongly differs from inherently parallel *high performance computing (HPC)* programs. This makes the parallelization a non-trivial task that demands for a specific approach, as the next section describes in more detail.

1.1 Problem Statement and Approach

The central requirements for embedded systems design are *predictability* and *robustness* [Hen08]. The consequences for automotive software parallelization are twofold:

a) *Predictability* — The execution order of runnables must be deterministic to form a predictable data-flow, i.e. an order in which runnables process data, from sensors to actuators.

b) *Robustness* — An upper bound on the sensor/actuator latency must be guaranteed.

The cores of a multicore processor perform calculations independent of each other, but they share resources such as the bus, memory, or other peripherals. That means the parallel execution on multiple cores requires a coordination of accesses to these shared resources to avoid unforeseen computational interleaving. This is necessary to avoid *data races*, i.e. concurrent access to the same shared memory location from at least two cores. Otherwise, inconsistent data might be the result or the data-flow between a sensor and an actuator might break. Another important factor is the duration of this computation, which should be as small as possible in automotive control software. Consequently, *the challenge in this Ph.D. thesis* is to schedule tasks to cores in a way that the data-flow between sensor and actuator produces a valid result with a low latency.

To achieve this, constraints must specify the correct functional behaviour to create the same sensor/actuator data-flow like in the former sequential execution. Unfortunately, the information about the legacy application is limited. However, this original application's configuration for the single-core ECU describes a correct functioning system and it can be used to derive parallelization constraints. Hence, the parallelization approach in this thesis relies on the original application's configuration for the single-core ECU. This approach leads to further challenges that are described in the following.

Automotive control software typically contains a high number of data dependencies. Many runnables frequently exchange data with each other. Every runnable is a consumer and a producer of interim results in a data-flow from a sensor to an actuator. This results in a dense *task dependence graph (TDG)*. Sensor data flows into this TDG in different runnables, traverses it on multiple paths, and the results leave it in different runnables. Thereby, paths overlay each other and a clear distinction between relevant and less relevant data-flows is hard.

Runnables consume and produce data with a fixed period and they have to be executed in the correct order to realize a data-flow from the sensor(s) to the actuator(s). This results in a large amount of *precedence constraints* that must be respected during parallelization and this frequently forces serialization of producer and consumer runnable. This imposes the question:

- ◇ *How can parallel execution of communicating runnables be enabled, but sensor/actuator data-flows be guaranteed with a worst-case latency?*

Furthermore, the AUTOSAR standard itself imposes limitations on the parallelization. A runnable is seen as smallest schedulable entity that requires an allocation to a core and a mapping to a task. The runnable is allocated indirectly by assigning the SW-C, containing the runnable, to a core. Hence, if two independent runnables are mapped to the same core, no parallel execution is possible. A parallelization approach beyond the state of the art potentially requires extensions of the existing standard. The challenge here is to minimize these changes and remain compliant to the standard as far as possible.

◇ *Which extensions for the AUTOSAR standard are required to guarantee efficient and deterministic parallel execution on a multicore ECU?*

This Ph.D. thesis attempts to answer these research questions. The contributions are described in the following section.

1.2 Contributions of this Thesis

This thesis was developed in the context of the research projects parMERASA [par11; Ung+16] and EMC^2 [EMC14]. The research activities in the projects were conducted in close collaboration with project partners and thus results have been published jointly. During the parMERASA project, a processor architecture was developed to address the challenge of overestimations in the *worst-case execution time (WCET)* by pessimism. The design of the processor architecture follows the demands of automotive software. The WCET estimations are thus less pessimistic than usual. Nevertheless, the approach in this thesis is applicable to any automotive control software.

Initially, a set of functional and non-functional objectives for automotive software parallelization has been defined based on the challenges described previously. A qualitative comparison and discussion of existing approaches has been conducted. As a result, the following main deficiencies have been identified in state-of-the-art approaches:

▷ Reconfiguring approaches define an efficient new application configuration for which a re-validation of the functional correctness is required. Moreover, their applicability is limited to systems with specific scheduling policies. Contrarily, preserving approaches maintain the same data-flow and fulfil most of the requirements, but they are not compatible with AUTOSAR.

▷ The AUTOSAR standard does not provide a method for predictable interprocessor communication between parallel executed tasks. The proposed interprocessor communication mechanisms either access shared memory locations in an unpredictable order or data dependencies force frequent serialization of the task scheduling. Strong *deterministic multithreading (DMT)* and *time-triggered architectures (TTAs)* ensure determinism. However, the former one misses a link between the internal artificial clock and real time. The latter one has strict time budgets that cannot be exceeded or the data-flow can break.

▷ A relaxation of inter-task data dependencies for improving parallel performance by reading less up-to-date input data is possible. However, state-of-the-art approaches do not define rules for transforming inter-task communication in such a manner that the sensor/actuator data-flow is predictable and reproducible. Moreover, the end-to-end latency constraints are maintained.

These deficiencies motivate for a parallelization approach that achieves determinism and robustness by considering the original application's configuration. Therefore, precedence constraints are derived between runnables of the same release period, using the original control flow, and between tasks with different release periods, using the original task priority. Latency constraints are derived, for relevant end-to-end chains, by taking precedence constraints and the original task period into account. These constraints build the foundation for further parallelization steps. The following sections describe the contributions of this Ph.D. over the state of the art.

Runnable-level Parallelization—The static partitioned scheduler RunPar [Pan+14] is proposed for the separate parallelization of tasks, which schedules runnables and not tasks. RunPar is a result of collaborative work in the parMERASA project. The author of this thesis mainly contributed in formulating the problem, analysing the case study, and conducting experiments. RunPar uses a bin-packing heuristic and a priority rule to assign the longest chain of dependent runnables first. The heuristic guarantees predictability, robustness, and data consistency of the parallel program by construction. The runtime overhead is low and a re-validation is unnecessary. The implementation can be done efficiently and only minimal modifications at operating system level are required. A complex diesel EMS is used to extensively evaluate RunPar. The results show that RunPar efficiently reduces a task's WCET on two cores, but the improvement on higher core counts is minor. The reason for this is that RunPar executes tasks in sequential order and this limits the achievable parallelism.

Consequently, this thesis proposes a new AUTOSAR structure named *Supertask*, which further exploits runnable-level parallelism of AUTOSAR tasks and still maintains the original data-flow of the application. Runnables from (originally) consecutive scheduled tasks are grouped into one Supertask, which then becomes a unique scheduling entity with a period equal to the least common multiple of tasks composing it. Runnables of the Supertask are then scheduled with RunPar, whereas inter-task data dependencies are respected. This allows for maintaining the original data-flow and increasing the parallelism in the legacy application. The scalability of Supertasks is evaluated and compared against separate parallelization with RunPar. The results report a significant improvement over RunPar, but more than two cores cannot be used efficiently with runnable-level parallelization.

These results also show that the original target of the parallelization is achieved only in parts. Consequently, distributing complete tasks is investigated as an alternative strategy.

Task-level Parallelization—The high number of precedence constraints between tasks causes frequent serialization, when RunPar is used and prevents an efficient usage of computational resources. The target of task-level parallelization is a relaxation of constraints that were respected by RunPar, so that communicating tasks can execute in a parallel way and still communicate in a predictable manner.

Therefore, this thesis proposes the novel communication mechanism *timed implicit communication (TIC)* [Keh+15], which overcomes the shortcomings of AUTOSAR implicit communication (an unpredictable data-flow). TIC [Keh+15] allows dependent tasks to execute in a parallel way, while maintaining the application's data-flow independent of the task schedule. Therefore, the communication between producer and consumer is decoupled by shifting the reception of data by one producer period (and bound to the task period). The producer task stores data in a buffer and attaches a publication timestamp, which is the end of the current producer period. Afterwards, the consumer task reads from the previous producer instance (compared to the single-core ECU execution) by selecting a value with the appropriate timestamp from the buffer. Thereby, the functional behaviour is independent from the point in time at which a task instance is scheduled within its period.

The communication between tasks is transformed in a predictable manner, which guarantees an identical data-flow for all target platforms. Thus, predictability and reproducibility are guaranteed. The approach is compliant to AUTOSAR and is implemented at AUTOSAR *run-time environment (RTE)* level and does not require modification of source code. The runnable-to-task mapping remains unchanged, which guarantees a correct data-flow within a task. The evaluation of TIC showed a speed-up of a 2.7 times faster execution on four cores and a 4.5 times faster execution on eight cores execution, when the utilization is at its maximum. However, the end-to-end latency is increased due to delayed transmission. Despite the benefits of TIC, it requires careful choice in applying this mechanism.

For this reason and for a better classification, runnable-level parallelization (with Supertasks and separate parallelization with RunPar) is compared to task-level parallelization (with TIC). The results show that Supertasks provide a higher speed-up under low processor utilization and TIC provides better performance under high processor utilization. Interestingly, these observations suggest the use of Supertasks and TIC as complementary strategies for increasing the overall system performance. Therefore, a method for deriving a hyperperiod schedule must be aware of the end-to-end latency and it must apply TIC and RunPar selectively. The coordination of RunPar and TIC is thus investigated.

Coordination of Runnable- and Task-level Parallelization—Introducing TIC allows parallel execution, but it also introduces an additional delay between a sensor and an actuator. Combining runnable- and task-level parallelization can compensate the negative impact on the latency and exploit the performance of the multicore ECU at the same time.

Coordinating runnable- and task-level parallelization means to optimize contradictory targets. However, each approach has individual advantages that can be used to compensate the shortcomings in the other. To achieve this, an evolutionary algorithm for solving the *resource-constrained project scheduling problem* is adapted to generate a set

of possible hyperperiod schedules. The *parallel schedule quality* (*PSQ*) is established as a metric for quantifying the quality of a schedule. This allows for selecting the schedule with the highest overall benefit from parallelization and finding a satisfactory solution in reasonable time.

The computed schedule is predictable, robust, efficient, cost-effective, and can be implemented in AUTOSAR straightforward. The *first-in-last-out (FILO)* latency is identical to the reference platform and reducing the processor's clock rate to a minimum utilizes idle intervals. All task periods are scaled with the same value for this.

RunPar and TIC significantly reduce the clock rate, whereas TIC achieves a much lower value (58% less than RunPar). Thus, solely using RunPar provides an overall better performance, if less than half of the inter-task communication uses TIC. The trend reverses when more inter-task communication is replaced and thus the best solution is found in this region. However, combining runnable- and task-level parallelism outperforms the individual approaches and provides the best overall performance.

Proof of Concept—The proposed approach is applied to a real diesel EMS to investigate the performance and efficiency with a real application as an example. In the first instance the mechanisms are applied and evaluated in simulation studies. Promising techniques are selected for implementation on an Infineon AURIX platform. The studies on the real platform are conducted to analyse the performance of simulative studies and deployment on a real platform. Finally, a migration process for an industrial environment is derived from these experiences. One section describes a step by step process. This eases the migration of legacy software to a multicore ECU.

1.3 Thesis Outline

Chapter 2 provides the background to this thesis and introduces fundamental terms. Automotive control software is explained and a short overview about the AUTOSAR software architecture standard is given. Fundamental work steps of software parallelization are explained. The processor architecture, which is used for the evaluation of the approach, is explained as well as its simulation environment. An overview about timing analysis techniques for the estimation of WCETs is given.

Chapter 3 begins with a definition of functional and non-functional objectives, taking the properties of legacy automotive embedded control software and the AUTOSAR methodology into account. Afterwards, state-of-the-art approaches for each parallelization step are explained and discussed in the context of the previously defined objectives. This specifically includes methods for automotive or embedded systems and more general methods. A discussion in the context of the objective is conducted. The chapter concludes with a summary of deficiencies.

Chapter 4 introduces the partitioned scheduling approach RunPar for runnable-level parallelization and introduces the optimization with Supertasks. The implementation in AUTOSAR and the handling of interrupts are described. The approaches are evaluated and compared against each other in an evaluations section.

Chapter 5 introduces a mechanism for predictable task-level communication called TIC that allows for parallel execution of tasks. Moreover, the migration with TIC and the implementation in AUTOSAR are described. The evaluation of the approach investigates the performance, buffer, and latency overhead. The findings motivate the combination of TIC with RunPar.

Consequently, chapter 6 describes the combination of the approaches from chapters 4 and 5 in a coordinated manner. Rules for the classification of communication are defined and a self-adaptive evolutionary algorithm, for combining the approaches, is described. The evaluation first compares the individual approaches against each other and investigates the performance of the combined approach afterwards.

The chapters 4 to 6 contain a qualitative analysis of each approach. A list of research questions is derived to motivate for experimental investigations. The results are discussed in the context of these questions and main findings from experiments are summarized in each chapter separately.

Finally, chapter 7 summarizes the main findings of the three main chapters 4 to 6 in this thesis and draws conclusions on the results. Moreover, ideas for improvements and future work are given.

Appendix A describes the applicability of the approach in an industrial mass production environment. Moreover, the chapter lists the tools required to implement the parallelization approach and it describes the migration process step by step to ease reproducing. Finally, the applicability of the approach is shown with an implementation on the Infineon AURIX processor.

2 | Background

> "There is nothing more
> practical than a good theory"
>
> Kurt Lewin, 1952

This chapter introduces the fundamentals for this thesis. Section 2.1 explains automotive control software basically. Domain specific wording, which is used throughout the following chapters, is introduced. Section 2.2 gives an overview about software parallelization. This includes fundamentals about subtask decomposition and dependency analysis. Section 2.3 describes the extraction of relevant parallelization constraints from automotive control software and it introduces necessary notations. Section 2.4 describes the considered processor architecture and employed tools. The last section in this chapter provides a summary of the main points.

2.1 Automotive Control Software

Automotive software is described according to the *AUTomotive Open System ARchitecture (AUTOSAR)* standard [AUT14a]. The consortium behind the standard is a worldwide development partnership of car manufacturers, suppliers, and companies from the electronics, semiconductor, and software industry. AUTOSAR standardises the software architecture, services, and a methodology comprising configuration and description methods. The software architecture covers a superset of properties from existing software and common development practices. AUTOSAR shaped a uniform understanding of control software across the whole automotive industry and harmonises the development process. The terminology is used throughout this thesis.

Automotive control software is integrated in an embedded real-time system; the *electronic control unit (ECU)*. In the scope of this thesis, this device executes one application with real-time properties. The model of automotive control software is derived from a classical task model presented in the fundamental work of Liu and Layland [LL73] as well as the formulation of Forget and Pagetti et al. [For+10; Pag+11].

Definition 2.1 (Real-time application): *A real-time application consists of a set of n tasks:*

$$\mathcal{A} = \{\tau_i \mid 1 \leq i \leq n, i \in \mathbb{N}\}. \tag{2.1}$$

The real-time attributes $(\pi_i, T_i, C_i, O_i, D_i)$ *characterise each task* $\tau_i \in \mathcal{A}$.

- π_i *is the priority.*
- τ_i *is instantiated periodically with period* T_i.
- τ_i^p *denotes the p-th iteration of task* τ_i.
- C_i *is the worst-case execution time (WCET) of* τ_i *expressed in time units.*
- O_i *is the release time of the first instance of* τ_i*, i.e. the offset with respect to the start time of the system.*
- $D_i \leq T_i$ *is the relative deadline of* τ_i*; they are implicitly defined, i.e.* $D_i = T_i$.
- *The release time of* τ_i^p *is* $o_i^p = O_i + pT_i$.
- *The absolute deadline of* τ_i^p *is* $d_i^p = o_i^p + D_i$.

Automotive control applications are real-time applications in the sense of definition 2.1. Later in this chapter, in section 2.3.2.2, the formulation is extended to describe and derive latency constraints. Chapters 4 to 6 consider different aspects of automotive control applications and adapt this definition for the specific needs.

Automotive control software has strict real-time constraints and the order in which output is produced matters. This mainly distinguishes the parallelization of such software from others, for example from the area of high performance computing. For this reason, precedence constraints are extracted to define a partial order for the execution of runnables, and latency constraints are extracted to specify the acceptable response time of the application.

2.1.1 AUTomotive Open System ARchitecture (AUTOSAR)

The software architecture of automotive control software is divided in three parts as shown in figure 2.1. A hierarchical component-based model describes the control application on the top layer. In this model, elementary code pieces (*runnables*) within the components implement the functional behaviour. Each component realizes a subtask of the overall control and communicates frequently with other components.

The structure of such an application is a block diagram, as they are characteristic for control engineering and similar to the representation in a model-based development environment like MATLAB. Executing the runnables of the blocks in the appropriate order guarantees the data-flow through the blocks. Therefore, runnables with the same

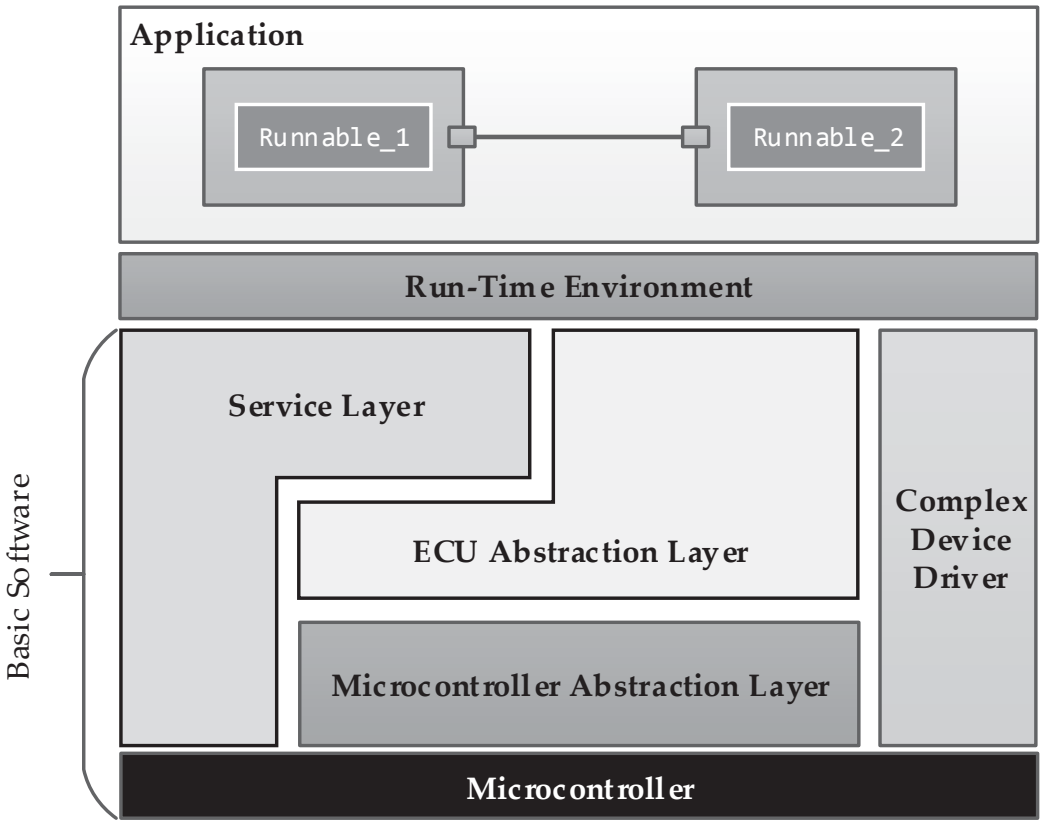

Figure 2.1: The AUTORSAR Software Architecture.

release time (periodic or sporadic) are grouped into AUTOSAR task structures and scheduled by the AUTOSAR OS, which is part of the *basic software (BSW)*. In contrast to the application layer, the BSW has a layered architecture and many hardware-specific components. Thus, the *run-time environment (RTE)* serves as separation between them. This makes the control application independent from the hardware architecture below.

The *AUTOSAR operating system (AR-OS)* [AUT14a] supports *periodic* and *sporadic* task execution. Deadlines are defined implicitly, which means a task instance must finish its execution before the release of the next task instance. Typically, a fixed priority-preemptive scheduling with *rate monotonic (RM)* priority assignment [LSD89] is used.

2.1.1.1 Virtual Function Bus

The *virtual function bus (VFB)* is a model for describing an AUTOSAR control application. A particular characteristic is the independence from the ECU's hardware architecture. The model has a component-based structure, describes the functional separation, and the communication between components. This allows for easier re-use of components.

The structural element is the *software-component (SW-C)*, which can be nested. No limitation about the number hierarchy levels is made. Within a SW-C runnables implement the functional behaviour. Within the component, they can exclusively exchange data through *inter-runnable-variable (IRV)*, which cannot be accessed from runnables in another SW-C. This concept is similar to the encapsulation of private data in a class instance of an object oriented programming language.

Interaction across component boundaries is possible through *ports*. Within the SW-C a runnable is associated to the port whose data are either read or written. Outside of a SW-C ports are associated with each other with an *assembly-connector*. One port can have multiple connections with other ports and multiple runnables can read from or write to the same port. That means the port serves as an interface and a runnable does not have a direct connection to another runnable, although they communicate with each other practically.

A *port-interface* further specifies a port. Multiple kinds of port-interfaces can be specified, because this is the only point for inter-component interaction. Most relevant for this thesis are sender-receiver communication, for transmitting a datum, and client-server communication, for invoking operations.

The developer can define the communication via IRVs, within a SW-C, or through a sender-receiver port, between SW-C, either as *explicit* or as *implicit*. By default, communication is explicit, i.e. a precedence constraint is imposed from producer runnable to consumer runnable, defining a strict order of execution. The consumer reads the most recent value of the producer. Implicit communication means that a datum is distributed to all consumer runnables after the producer runnable has finished its execution. On the consumer side, the datum is buffered and calculations are performed on a copy meanwhile. As a result, concurrent execution of runnables is possible, because the datum is buffered, but delivered with a delay. This is a form of *asynchronous communication*. Storing data in a queue is another possibility, but their use is rare and is therefore not considered in this thesis.

In client-server communication, the provider of the operation is denoted as *server* that offers an operation to one or more *clients*. That means the client-runnable triggers the execution of a server-runnable through a well-defined interface. The server-runnable typically maintains an internal state and parallel execution is therefore not possible.

Apart from those communication mechanisms, four other kinds of ports can be specified:

1) a *parameter* interface provides a constant value or calibration data,
2) a *non-volatile data* interface provides access to non-volatile memory,
3) a *trigger* interface immediately starts the execution of a runnable,
4) and a *mode-switch* interface notifies a SW-C about the state from the mode manager to adjust the component's behaviour.

2.1.1.2 Run-Time Environment

The AUTOSAR RTE serves as a separation between the layered architecture design of the BSW and the component-based design of the application. It makes the application independent from a specific ECU and is the realization of the port-interfaces in the VFB model of an application. This includes the application's access to the BSW modules, the connections to the AUTOSAR OS, and communication services. A SW-C can only access the BSW modules on the same ECU, but it can communicate with SW-Cs on another ECU through the RTE. The RTE therefore hides the concrete implementation of the communication behind a standardised *application programming interface (API)*.

The RTE guarantees data consistency for IRVs and sender-receiver communication. Additionally, it is possible to use *per-instance memory (PIM)* for direct memory accesses, but the RTE does not guarantee consistency in this case and thus PIM is not recommended and consequently not considered in this thesis.

The RTE is in addition responsible for scheduling SW-Cs from the application and from the BSW. Therefore, runnables with the same release period are typically grouped into the same task. Although, this is not stipulated by the standard, but it represents a reasonable and logic combination. The task bodies and schedule tables are automatically generated from the RTE configuration. The OS starts the tasks and the runnables within the task are executed one after another. Guards may check if all runnables must be executed. The RTE and the BSW scheduler are generated for each ECU to ensure efficiency of the implementation.

2.1.1.3 Basic Software

The *microcontroller abstraction* makes the layers above independent from the underlying processor architecture. This layer has direct access to the microcontroller hardware, contains the hardware-specific drivers, and provides a standardized set of interface functions to the ECU *abstraction* layer above. ECU abstraction provides interfaces to the hardware drivers of the microcontroller abstraction, contains external drivers, and it provides an API for accessing peripherals and devices regardless of their physical location. The communication service for example is part of this layer. A *complex device driver* allows direct access from the RTE to the microcontroller hardware. This interface is used to integrate proprietary functions, implement time-critical or complex sensor/actuator operations, or encapsulate legacy functions in the architecture. The *service* layer provides basic services to the application and BSW modules. It contains amongst others the AUTOSAR OS, which is based on the specification *Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen (OSEK)* [OSE05].

2.1.2 Migration to AUTOSAR

AUTOSAR is now introduced in the *electrical/electronic (E/E)* architecture development processes of car manufacturers. The standard enables shorter design cycles and easier re-use of legacy software [DT11]. Nevertheless, a large fraction of legacy control software is not compliant to AUTOSAR standard. The migration is necessary to ensure interoperability and re-use in the future.

Migrating legacy control software to AUTOSAR is twofold. It can be distinguished in defining the communication, on application layer with the VFB (SW-Cs, ports, runnables, etc.), and configuring the BSW, i.e. OS tasks, runnable-to-task mapping, etc. The description and configuration are defined according to the AUTOSAR meta-model and stored in a standardised XML-file format. The application's source code is refactored. Concretely, the communication is replaced by RTE API calls and the original *operating system (OS)*, drivers, etc. are replaced by the BSW.

The migration to AUTOSAR becomes more relevant in the context of parallelization, because the SW-C structure of the application has a direct impact on the achievable degree of parallelism. The reason for this is runnables are distributed to cores indirectly by assigning a SW-C to a core. As a result, if a component contains two independent runnables no parallel execution is possible. The migration must therefore create a fine-grained VFB model, in which a SW-C contains only one runnable to allow for a maximal degree of parallelism in later steps.

Only few publications about the migration of legacy software to AUTOSAR are available, because this activity often concerns proprietary or confidential contents. Reports about the migration in the context of parallelization are rare. However, existing literature concerns two kinds of legacy software. The software can either be available as a model in a proprietary model-based environment like MATLAB/Simulink or it can be plain source code.

2.1.2.1 Migration of Model-based Legacy Software

A development environment for model-based automotive software often allows for direct export of an AUTOSAR XML description and source code. For example, the commercial tool Embedded Coder [The15a] for MATLAB and Simulink supports these features [The15b].

Manufacturers that maintain an own proprietary model can transform their model with the ATLAS *Transformation Language (ATL)* [Jou+06]. The language specifies the transformation from an arbitrary source model to and target model with a mixture of declarative and imperative constructs. The work by Selim et al. [Sel+12] describes the migration of the *General Motors (GM)* meta-model to the AUTOSAR meta-model via model transformation with ATL. Horizontal transformation manipulates models at

the same abstraction level, but possibly expressed in different formalisms. An example is the transformation of a MATLAB state machine into an UML state machine. Vertical transformation manipulates different abstraction levels. An example is generating a deployment model from a software and hardware architecture model.

The migration steps cover the application layer only and the configuration of the BSW is done in a separate step. The generated SW-C structure is equivalent to the model from which it is generated. Thus, the hierarchy can have multiple levels with independent runnables within the same SW-C. Model transformation techniques do not consider software parallelization. The transformation from a (proprietary) meta-model to the AUTOSAR meta-model is straightforward, because the source model contains explicit information about the control function, data dependencies, etc. and the transformation can be done without loss of information.

2.1.2.2 Migration from Plain Source Code

The migration from plain source code is complex and the expenditure is high. An in-depth analysis of data dependencies is needed. Common techniques are explained later in section 2.2.2. The result from this analysis must be interpreted to create an application model, the BSW must be configured, and the source code must be refactored to use the RTE API.

A helpful tool for creating a VFB model is the AUTOSAR Tool Platform (Artop) [Knü+10]. Artop is a textual description language, which can be used royalty free by all AUTOSAR members. The tool is based on Eclipse and provides base functionality for creating design and configuration tools for AUTOSAR. Several subprojects provide a textual modelling environment (ARText), for describing SW-Cs, timing, variant handling, etc., or a test environment (ARUnit), for implementing test cases.

Kum et al. [Kum+08] describe the migration of a legacy software including the AUTOSAR BSW. The methods separate the application's source code into parts: the control application, ECU devices, communication, other services, and peripherals. These parts are mapped to the equivalent part in AUTOSAR afterwards. For example, peripherals are mapped to the microcontroller abstraction in the BSW.

The control application is first divided into SW-Cs and second broken down into runnables. A data-flow between two runnables is interpreted as sender-receiver communication and a function call is interpreted as client-server communication. This interpretation is advantageous, because the RTE guarantees data consistency in both cases. However, the actual implementation of the process is not described.

Contrarily, the case study by Scheidemann et al. [SKS10a; SKS10b] is an example for the usage of ARText. Like the approach of Kum et al., a static data and control flow analysis is conducted first. The information about function calls, read and write access

to global variables, and data types of variables and parameters are used to map the communication to AUTOSAR communication paradigms in ARText.

The RTE (runnable-to-task mapping, client-server communication, and events) and communication stack are configured in a separate ECU development phase. The OS is replaced by an AUTOSAR compliant one and the parameters are determined from the former OS configuration. Contrarily, Scheidemann et al. already consider the multicore ECU and thus a mapping is generated (this is discussed in more detail in chapter 3) first. The BSW is configured in a semi-automatic way afterwards, because it depends on the mapping.

2.1.3 Case Study: Diesel Engine Control

The performance and the efficiency of the approach in this thesis need to be evaluated. Using a real application is advantageous, because this allows for drawing conclusions in a real deployment. To that end, a diesel *engine management system (EMS)* is used as example, because the application contains a large amount of highly connected runnables.

The examined EMS comprises roughly 1200 runnables that implement the behaviour of numerous SW-Cs. They exchange data via sender-receiver and IRV communication. The SW-C's internal states are updated at different rates, e.g. sensor values are polled with a greater or equal frequency than they are processed. Therefore, runnables with the same released period are mapped to the same task.

Figure 2.2 provides a simplified description of the task set in the diesel EMS. The nodes in this directed graph represent tasks and the edges represent communication between them.

The task τ_1 executes after an interrupt from the camshaft sensor (*crank-angle task*). Tasks τ_2 to τ_{12} execute with the period denoted by the label close to the node, task τ_5 has a period of one millisecond for example.

A solid edge represents explicit communication between two tasks, which is imposed by the runnables mapped to this task. As a result, communication between takes place with different frequencies. The dashed edges represent implicit communication. In this example, only the communication with the sporadic crank-angle task τ_1 is implicit.

The EMS also contains client-server communication. This requires a mechanism for maintaining memory coherency, when a server-runnable updates the internal state of the SW-C it belongs to. This is out of the scope of this thesis. To still conduct a performance evaluation, calls are enclosed in *ticket-locks* [ORS14]. They block other runnables until the execution of the server-runnable has finished. These locks can