Parallelization of Legacy Automotive Control Software for Multi-Core Platforms

vorgelegt von
M.Sc.
Martin Lowinski

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Clemens Gühmann
Gutachterin:	Prof. Dr. Sabine Glesner
Gutachter:	Prof. Dr. Rainer Leupers
Gutachter:	Prof. Dr. Ben Juurlink

Tag der wissenschaftlichen Aussprache: 10. Dezember 2018

Berlin 2019

Abstract

Automotive control-based applications become more and more sophisticated due to the continuous addition of new functionalities. At present, this functionality is implemented as runnables that are recurrently and sequentially executed inside software tasks. These computing-intensive tasks are typically concurrent to each other and hence can be executed in parallel on multi-core platforms.

Due to the increasing functionality, future tasks will exceed the computational power of a single core and have to be parallelized. When parallelizing a single task, it is critical that the order of execution, the communication and its timing of the legacy implementation is maintained. Otherwise, the behavior may change that could result in an incorrect functionality. These legacy requirements limit the concurrency inside a task such that an efficient parallelization is often hardly possible. Due to the high amount of functional interdependencies and the varying runnable execution times, synchronization is required to ensure the correct order of communication between parallelized tasks. However, any synchronization can potentially degrade the performance especially when the observed worst case execution times (oWCETs) of the runnables are overrun.

In this thesis, we present a parallelization approach that can cope with the requirements of automotive legacy software tasks. Our approach provides scalable and efficient heuristics to exploit the concurrency of real-world legacy tasks for parallelization. By optimizing the parallelization toward a novel robustness metric, the approach can cope with oWCETs and reduce the synchronization between resulting parallel tasks.

The concurrency inside legacy tasks is often limited due to various optimizations and construction principles for single-core platforms. Yet, these limitations are not always mandatory to the functional correctness. Our approach provides a semi-automated analysis to detect and eliminate these limitations and to increase the concurrency of a task that can be exploited for parallel execution. The typical bottleneck resource in any multi-core migration is the effort to ensure the functional correctness of the system by consulting functional development experts. This effort is reduced through our approach by ranking and proposing only the most promising task design changes to these experts for the validation of functional correctness.

Finally, we have implemented the proposed approach and evaluated it in several case studies with tasks from real-world automotive engine management systems.

Zusammenfassung

Aufgrund von neuen Funktionalitäten werden regelungstechnische Anwendungen im Automobilbereich stetig anspruchsvoller. Aktuell werden diese Funktionalitäten in Runnables implementiert, die periodisch und sequentiell innerhalb von Software Tasks ausgeführt werden. Diese rechenintensiven Tasks sind typischerweise zueinander nebenläufig und können somit auf Mehrkernsystemen parallel ausgeführt werden.

Infolge von zunehmender Funktionalität werden Tasks zukünftig die Rechenleistung eines einzelnen Rechenkerns übersteigen und müssen parallelisiert werden. Bei der Parallelisierung eines Tasks ist es entscheidend, dass die Ausführungsreihenfolge, die Kommunikation und dessen zeitliche Abfolge der vorhergehenden Implementierung beibehalten wird. Andernfalls könnte es das Verhalten so verändern, dass eine Funktionalität unzulässig wird. Diese Alt-Anforderungen beschränken die Nebenläufigkeit innerhalb eines Tasks, sodass eine effiziente Parallelisierung oft kaum möglich ist. Aufgrund von einem hohen Maß an funktionalen Abhängigkeiten und unterschiedlichen Ausführungszeiten der Runnables ist eine Synchronisation notwendig, um eine ordnungsgemäße Kommunikation zwischen den parallelisierten Tasks zu gewährleisten. Allerdings beinhaltet jede Synchronisation das Risiko die Leistung zu verschlechtern, insbesondere wenn die beobachteten WCETs der Runnables überschritten werden.

In dieser Dissertation präsentieren wir ein Vorgehen zur Parallelisierung, das den Anforderungen von Software Tasks aus Alt-Systemen im Automobilbereich genügt. Unser Vorgehen bietet skalierbare und effiziente Heuristiken um die Nebenläufigkeit von Tasks aus industriellen Alt-Systemen auszunutzen. Durch die Optimierung der Parallelisierung mittels einer neuen Robustheits-Metrik beherrscht unser Vorgehen beobachtete WCETs und reduziert die Synchronisation zwischen parallelisierten Tasks.

Oft ist die Nebenläufigkeit von Tasks aus Alt-Systemen jedoch durch Optimierungen für Single-Core Prozessoren beschränkt. Diese Einschränkungen sind allerdings nicht immer notwendig um die ordnungsgemäße Funktion zu gewährleisten. Unser Vorgehen bietet eine halbautomatische Analyse, um diese Einschränkungen zu finden und zu entfernen und damit die Nebenläufigkeit eines Tasks zu erhöhen, die widerum bei der parallelen Ausführung von Nutzen ist. Der typische Engpass bei jeder Migration auf Multi-Core Prozessoren ist der Arbeitsaufwand die funktionale Korrektheit zu gewährleisten indem Funktionsentwickler befragt werden. Dieser Arbeitsaufwand wird durch unser Vorgehen reduziert, da nur die vielversprechendsten Änderungen im Design eines Tasks diesen Experten zur Validierung der funktionalen Korrektheit vorgeschlagen werden.

Wir haben das vorgeschlagene Vorgehen implementiert und in mehreren Fallstudien mit Tasks aus Motorsteuergeräten aus dem Automobilbereich evaluiert.

Acknowledgments

I would like to say thanks to all the people that supported me during my time as a PhD student. First and foremost, I want to thank Prof. Dr. Sabine Glesner for supervising my thesis and the insightful discussions and always helpful advice. I'd also like to thank Prof. Dr. Leupers and Prof. Dr. Juurlink for advising and surveying this thesis. I am grateful for being a part of the SESE research group at the TU Berlin, although working from Stuttgart. Thank you for all your advices, helpful discussions and support.

As an industrial PhD student at the Robert Bosch GmbH in Renningen, I want to express my sincere gratitude to my supervisor Dr. Dirk Ziegenbein. His valuable and sharp insights from academia and industry has made my time at the ManyCore project a both challenging and enriching experience. My appreciation also goes to my Bosch colleagues and especially to Dr. Jochen Härdtlein and Dr. Björn Saballus who created this extended collaboration of academia and industry. While sharing a research vision with my fellow ManyCore PhD students, namely Alexander Biewer, Dr. Matthias Freier, Dr. Peter Munk and Felix Rützel, I am likewise thankful for the good times we had. Thanks also to all the members of the Bosch research project ManyCore and the CR/AEA department. It was a pleasure to work with you in this skillful, friendly and open-minded environment.

Last but not least, I am most grateful for my family and dear friends. Thank you for the continuous encouragement, the endless support, love and joyful moments in my life. Writing this thesis would not have been possible without you.

Contents

1 Introduction

Today's automotive applications employ highly sophisticated comfort, safety and power-train functionalities which will continue to grow in the future [Aoy12, BKPS07]. For example, the power-train has to fulfill constantly tightened emission limits. The resulting complex functionality is implemented in embedded real-time control software with an increasing demand for computing power. Parallelism provided by multi-core processors is the most promising solution to satisfy the computational demand [But12]. To leverage it efficiently, the legacy software, originally developed and optimized for single-core processors, needs to be parallelized.

1.1 Problem

Automotive legacy control software is composed of interdependent software units. These software units are allocated to recurrently executed and concurrent tasks. Using the concept of Logical Execution Time (LET) [HHK01], the tasks can run on single- or multi-core processors with a deterministic behavior independent of their distribution to cores. However, there has been the established construction principle to allocate all software units with the same recurrence pattern to a single task. This construction principle is optimized for single-core processors and reduces the resource overhead. With this principle, additional or more complex functionalities cause the number of software units of a task to grow. The computational demand of tasks thereby increases and may exceed the computational power of a single core [MHAK15]. Such a task has to be parallelized by remapping the software units to multiple parallel tasks.

The software units interact through sensors and actors with the physical environment [SZ16]. The interaction is based on messages that are communicated between the software units. Every communication creates data and ordering dependencies. While parallelizing, these dependencies have to be maintained to ensure the data flow and thus the correct functionality of the legacy software. Also, control-based software is very sensitive to timing [AEF+14]. A change in the timing of a dependency results either in earliness or lateness of the communicated data. Especially in the latter case, a delayed communication potentially alters the functionality of controllers. Depending on the controller design, the delay may violate latency or stability requirements and other performance criteria. But ensuring all dependencies and their respective timing limits the parallizability and is prohibitively expensive in terms of synchronization overhead.

For a better understanding of the relationship between dependencies, timing and parallelization we introduce two fundamental terms by an example. Assuming a task is periodically executed and has two software units. One software unit samples the engine temperature and communicates the temperature value to the other software unit that represents the engine thermal controller. Due to the optimization for single-core, both software units are allocated to the same task as they have the same recurrence pattern. Furthermore, the two software units are executed in the order such that the sampling software unit runs before the engine thermal controller and always receives the latest temperature value. The timing between the two software units results in a tight coupling and a reduced concurrency, the *available concurrency*. However, from the physical and control-engineering perspective the temperature value could be communicated with a delay without compromising the correctness of the system. In our example, the assumed requirement is that the engine thermal controller has to start the engine fan when the temperature hits a threshold within 5 seconds. Assuming that the task that contains both software units is executed every 100 ms, a delay in communication of 100 ms between the sampler and the controller does not violate this requirement. With a delay, i. e., a relaxed timing, the thermal controller receives the temperature value from the sampler of the previous task instance. Hence, the two software units are decoupled temporally inside the task. The effect is that they become concurrent to each other which again can be used for parallelization. The resulting concurrency is called *inherent concurrency* as it represents the physically required timing and coupling of the software units.

The key for an efficient parallelization of a task is the knowledge of the *inherent concurrency* as the physically required timing is often more relaxed than the *available concurrency*. A relaxed timing improves the concurrency between software units inside a task that in turn can be exploited for parallelization. While the dependencies are maintained, relaxing means to change the timing of the communication and thus possibly changing the functionality. The knowledge where relaxation is allowed is typically not documented and hence only domain experts of the system are able to determine if the functionality is still correct. There are various techniques for the experts to evaluate the impact of an altered timing on the functionality of the controller such as formal verification or simulation. Nevertheless, the impact on the controllers' functionality of a relaxed timing can only be evaluated for a small number of dependencies due to time and cost reasons.

1.2 Objectives

This thesis addresses the problem described above which is the parallelization of highly inter-dependent tasks from automotive control software. Our solution establishes an analysis to guide a domain expert in finding a suitable and functionally correct parallelization with little effort. We require our analysis to fulfill the following criteria:

- *Automotive Ecosystem:* The approach should use existing models and respect requirements as well as constraints of the automotive domain. A convenient integration into the development processes of automotive Electronic Control Units (ECUs) and its surrounding ecosystem is required.
- *Scalable & Efficient:* The approach should cope with large and complex software models from e. g., real-world automotive applications such as an Engine Management System (EMS).
- *Observed WCET:* The approach should cope with execution time bounds that are measured at certain working points and are thus only observed bounds, i. e., tight but unsafe. Due to the complexity of automotive applications and the lack of predictable hardware, exact execution time bounds are unavailable.
- *Improve Concurrency:* The legacy design of tasks in the automotive domain strongly limits the concurrency inside a task. The approach should therefore be able to increase the level of concurrency, but should always maintain correctness.
- *Reduces Interactions with Experts:* Interactions with domain experts should be reduced in all phases of the approach because the domain expert is a scarce resource and the invested time needs to be spent efficiently. Requests of information to the domain expert should be precise and simple to ease information retrieval.

1.3 Proposed Solution

To achieve the objectives given above, we propose a semi-automatic iterative approach to parallelize a task by leveraging domain knowledge of experts. The main idea is to relax specific dependencies, i. e., changing the timing of the communication which may alter the functional behavior but is validated to remain correct. The starting point of our approach is the abstract representation from the legacy implementation of a task with its containing software units, dependencies and timing.

In the first step we parallelize a task by exploiting the available concurrency and check if the resulting parallel tasks match the parallelization goal. We parallelize by remapping the software units to multiple task partitions within the boundaries of the available concurrency. The dependencies and the respective timing between the software units are hereby maintained to ensure the legacy functionality. For the remapping we have two techniques. The first technique uses domain-specific heuristics that are scalable and can cope with high numbers of dependencies. We tackle the problem of widely varying execution times and Observed Worst-Case Execution Time (oWCET) by introducing synchronization between task partitions only where absolutely necessary. This technique can therefore deal with the current state of the practice in the automotive domain where software is typically not designed to allow tight Worst-Case Execution Time (WCET) bounds and sporadic misses of task deadlines are often tolerable. The second technique exploits the functional independent or temporally decoupled software units of the available concurrency. According to LET, when two software units of the same task either do not communicate or communicate solely via a relaxed timing, each can be allocated to a separate task partition. This technique allows to create task partitions without any synchronization and thus without any overhead.

If the parallelization goal is not reached, the available concurrency is too limited, e. g., due to the high number of dependencies or the synchronization overhead. Thus, in a second step we iteratively advance further towards the inherent concurrency by relaxing dependencies.

In general, every dependency is a potential candidate for evaluation by an expert to check if its timing can be relaxed without compromising the correctness of the system. To reduce the amount of interactions with an expert, we specifically look for dependencies which are more *relevant* to solve the parallelization challenge than others. The central idea is to determine a single dependency as candidate for evaluation by an expert. Through static analysis of the software units' dependencies and their role in the control algorithms, every dependency is assessed by multiple criteria. To identify the relevance, there are two groups of criteria: The first group indicates the potential benefit of a dependency, if it is relaxed, toward the given parallelization goal. Using graph algorithms, this group analyzes properties of the communication between software units. The second group rates the success, i. e., how likely it is that the timing of a dependency can be relaxed without significantly degrading the functional behavior. This can be quantified by the analysis of the physical dynamics of a dependency. These two groups of criteria are evaluated for each dependency, aggregated, and tuned for the different parallelization strategies. The results create the basis to select the most relevant dependency candidate. As our approach is iterative, the most relevant dependency is presented to the expert on every iteration.

The expert then evaluates if the controller still behaves correctly despite the relaxed timing of the dependency. On success, i. e., when the timing can be relaxed safely, the relaxation refines the implemented concurrent design of the task. This refined design becomes the input for the next iteration in our approach to find a suitable parallelization of a task.

1.4 Main Contributions

In summary, the main contributions of this thesis are:

- We propose a general model-based semi-automated workflow to parallelize a task from automotive legacy control software.
- We present two techniques to create multiple task partitions from a single task while maintaining all dependencies of the legacy implementation.
  - The first most notable technique parallelizes using domain-specific scalable heuristics while keeping the synchronization to a minimum. By optimizing for a novel robustness metric, we can handle observed worst case execution times while sustaining a speedup despite synchronization.
  - The second technique exploits the characteristics of LET with functionally independent or temporally decoupled software units to parallelize a task without any synchronization overhead.
- To extract the inherent concurrency of a task, we propose an approach to determine the dependencies which are more relevant to ease the parallelization challenge and thus more promising candidates for evaluation by a domain expert than others. This approach reduces the interactions with the domain expert and requests are held simple.
- The whole approach was implemented using the open source platform AMALTHEA to fit into the complex tool chains in the automotive domain. This supports interoperability, extensibility and unifies data exchange between tool vendors, engineering companies and other suppliers in the toolchain.
- We evaluated our approach on a real-word Engine Management System as an example of a complex application in the automotive domain. We performed experiments to investigate the performance benefit of our parallelization approach that shows a significant speedup while maintaining correctness.

These contributions have been made as part of the ManyCore research project in the Corporate Sector Research and Advance Engineering of the Robert Bosch GmbH. The Robert Bosch GmbH is a multinational company for engineering and electronics and a leading supplier of automotive components, technologies and services. Within Bosch, the ManyCore project is researching on systems engineering technologies and methods for parallel automotive applications. This thesis also represents a contribution to the project and its vision for parallel and robust control applications.

1.5 Overview of this Thesis

This thesis is structured as follows: In Chapter 2, we give an introduction to the automotive systems domain with its real-time requirements, feedback control systems, related standards and multi-core platforms. Then, in Chapter 3, we review related work on parallelization approaches, in particular approaches that utilize domain knowledge. In Chapter 4, we describe the fundamental problems when searching for concurrency that can be exploited for parallelization and outline the solution idea. In Chapter 5, we present our general concept and describe the contained components in detail. Starting with the fundamental graph structures, this chapter presents the core workflow for the parallelization of a legacy task. This chapter also describes the integration of the concept into the automotive development processes. The Chapters 6 and 7 present the two main components for splitting a task into multiple task partitions and the analysis to find suitable candidates to improve concurrency. In Chapter 8, we show the benefit of our approach applied to real-world automotive case studies. In particular, we use an EMS to demonstrate the efficient parallelization of a task with little effort. We conclude in Chapter 9 and provide possible directions for future research.

2 Application Domain

This chapter introduces fundamental background concepts, terminologies and notations that are used in this thesis. We start with embedded real-time control systems, then discuss standards and models and finish with the typical development process in automotive.

Technological advances foster ubiquitous and omnipresent electronic devices that are integrated in products of today's everyday life. These devices interact with the outside world by measuring, controlling and actuating through the product itself to the physical environment around it. In the following we use the term *embedded system* for an electronic computing device or system that forms an indivisible part or is tightly integrated into a product and is designed for a dedicated function [Mar10]. In contrast to general purpose computers, embedded systems have to cope with conflicting priorities such as performance, resource requirements (e. g., size, weight, power) and costs. Also, ninety-eight percent of all produced microprocessors are integrated in embedded systems [Bar09] and hence play a huge role in the semiconductor industry. Besides products that integrate embedded systems such as fridges, ticket machines, or TVs, safety-critical embedded systems such as pacemakers, robots or the field of Highly Automated Driving (HAD) are a far greater challenge.

2.1 Automotive Embedded Systems

An especially challenging field for embedded systems is the automotive domain due to conflicting requirements of safety, performance and resource efficiency, low costs and coping with the increasing amount of systems and software [Bro05]. Figure 2.3 shows the car as a product with its many and diverse *system of embedded systems*. In a car, these embedded systems are typically integrated as an Electronic Control Unit (ECU). Therefore, embedded systems are not only a challenge due to being safety-critical, but also due to the complexity as the embedded systems need to cooperate to provide the dedicated function of a car. Automotive embedded systems have to be safety-critical because an error can cause high costs, severe injuries or in the worst case the loss of human lives. Examples that highlight this requirement are the airbag, Electronic Stability Control (ESC), Anti-lock Braking System (ABS) or radar-based systems. Resources for embedded systems in the automotive domain are limited because of efficiency and power dissipation reasons. For example, the camera for road sign detection is typically mounted behind the rear mirror. Therefore it cannot be actively cooled as it is part of the passenger compartment. The hardware parts for embedded systems are highly cost sensitive due to mass production. Small changes in the price per unit can have a huge impact on the production and business model. Today, a premium-class car contains 70 to 100 microprocessor-based Electronic Control Units (ECUs) and executes approximately one hundred million lines of code [Cha09]. An overview of such a system of embedded systems is illustrated in Figure 2.3. This huge amount originates from an exponential growth in software. The growth is caused by innovation through new functionalities that depend on complex algorithms and process huge amounts of data under real-time and safety-critical constraints [UBG+13]. One big driving force for this trend are the driving assistance systems. Starting with the simple cruise control system, these systems develop toward Highly Automated Driving (HAD) and ultimately autonomous driving.

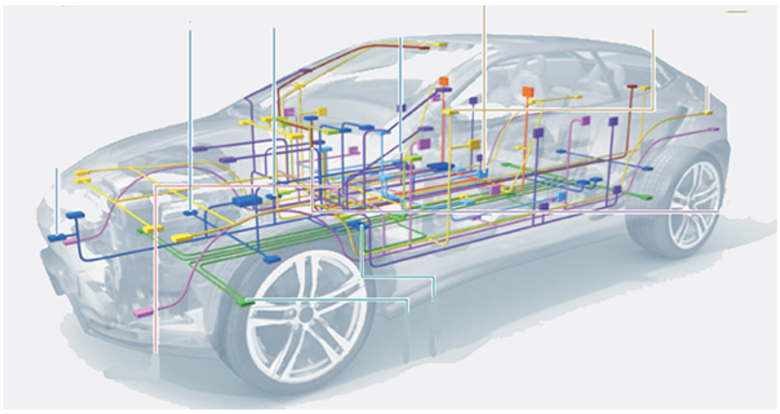


Figure 2.1: Overview of the system of embedded systems in a car [Don14]. Every box represents an Electronic Control Unit with a dedicated function inside the car and is connected to other ECUs.

2.1.1 Feedback Control Systems

A single embedded system inside a car always integrates hardware and software into a combined product that is interacting with and controlling its environment. In the following we call the integrated software an *application* because of its dedicated functionality. In embedded systems and especially in the automotive domain, the main purpose of applications is to control a chain of cause-and-effect with the environment. Hence they are *control applications* for feedback control systems.

Control applications have the task to influence a time-continuous process such that it behaves in a certain way. Hereby, a parameter has to be chosen that is adjustable from the outside and can achieve a certain goal. As this choice depends on the distance to the specified goal, this process becomes a feedback loop with a controller that has yet to be defined. Control applications interact with a *dynamic system* (also called: plant). A system becomes dynamic when the characteristic parameters are a function of time. There are two important kinds of characteristic parameters. The input parameters cause changes inside the system in a time-continuous manner while the output parameters characterize the behavior of the system to these input parameters. The goal-oriented interference with a dynamic system is called controlling. A controller calculates a control signal depending on a reference signal. The controller and the dynamic system are interacting continuously, hence it is called a control loop.

The more detailed interaction of a controller and a dynamic system in conjunction with its parameters is shown in Figure 2.2 [Lun16]. In this figure, the controller's task is to determine the input parameter $u(t)$ that is fed into the system such that the output parameter $y(t)$ ideally equals the reference value $w(t)$ at all times t. Hereby $y(t)$ is dependent on $u(t)$ and the disturbance of the dynamic system $d(t)$ that is manipulable. For the calculation of $u(t)$, the controller knows the reference value $w(t)$ that is to be achieved and the current output parameter $y(t)$ of the dynamic system. Based on these two parameters, the controller can determine the control error $e(t)$ which is important to calculate the input parameter $u(t)$ and a unique characteristic of feedback control systems. The time the controller needs to calculate $u(t)$ is called dead time.

As the controllers are integrated on ECUs that integrates a digital microcontroller or -processor, the processing is done discretely. Hence, a continuous input parameter has to be converted to a time- and value-discrete parameter and a discrete output parameter back to its external form of the dynamic system. The conversion from continuous parameters to time-discrete ones is called sampling; the parameters become value-discrete due to quantization errors in analog-digital converters.

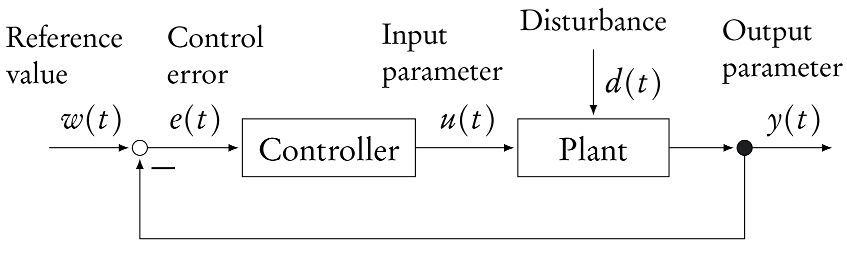


Figure 2.2: General concept of a feedback control system that shows the interaction of the controller, the plant and the corresponding parameters [Lun16].

The Engine Management System (EMS) is one of the most complex and computing intensive examples of an automotive control application [Cha09, But12]. An EMS is a control application and the software part of the ECU that controls a combustion engine. The automotive combustion engine is technically highly complex and sophisticated [Rei14]. Such an EMS constitutes the use case throughout this thesis. For less complex systems, the presented approach in this thesis is assumed to be applicable as well.

When looking at the evolving mechanism of the combustion engine itself, its essential goal is to generate a mechanical force by combustion of a fuel. At the time the Otto and Diesel combustion engines were invented, the dynamic system of combustion was controlled purely by mechanical components. However, the continuing development increased the complexity drastically over the years. This development was and is still motivated by the automotive industry and the government to increase efficiency and performance.

Today, the EMS is one of the most complex control applications as it has to control a wide variety of subsystems. Figure 2.3 gives an overview of the subsystems that an EMS for an up-to-date Diesel engine has to manage. Similar to the car itself, the EMS is again a system of systems. In the beginnings of the combustion engine, the throttle, injection, ignition and exhaust systems were the main components. Today, there are turbochargers, catalysts, additives injected into the exhaust and many more additional systems. Each system brings its own dynamic system and controlling challenge.

Internally, each subsystem of an EMS typically controls its own dynamic system. Yet, most of these systems are not isolated but highly dependent on each other. Additionally, the physical requirements of an engine demand high quality and stable controllers. Besides the physical requirements, many stakeholders, laws and regulations require improved fuel efficiencies, less CO_2 emission and other features. This again facilitates the complexity of the subsystem controlling in an EMS.

All controlling systems in an EMS have to follow a causal loop such that the combustion works properly. For example, the system for the throttle calculates the amount of air needed by the injection, the injection system calculates the amount of gas that has to be combusted, air and gas have to be ignited at the right time that is calculated by the ignition system and so on. This causal loop is a very sensitive process as it depends on a huge amount of parameters that are dictated from the dynamic system and the controllers. This leads to a tight coupling of the systems and a high amount of communication between them.

Inside a subsystem of an EMS, a single functionality is encapsulated into a *runnable* [ITE]. For example, one functionality of the subsystem for the injection is to calculate the amount of gas for the combustion. This calculation, i. e., the algorithm, is wrapped in a runnable. Other typical functionalities are the sampling of parameters from the environment or driving actuators. A runnable from the set of runnables $r \in \mathbf{R}$ is technically an encapsulated portion of sequential code, i.e., a void-void function. Besides the functionality, the characteristic properties of a runnable needed in this thesis are: Maximum sample interval, service, size and the Observed Worst-Case Execution Time

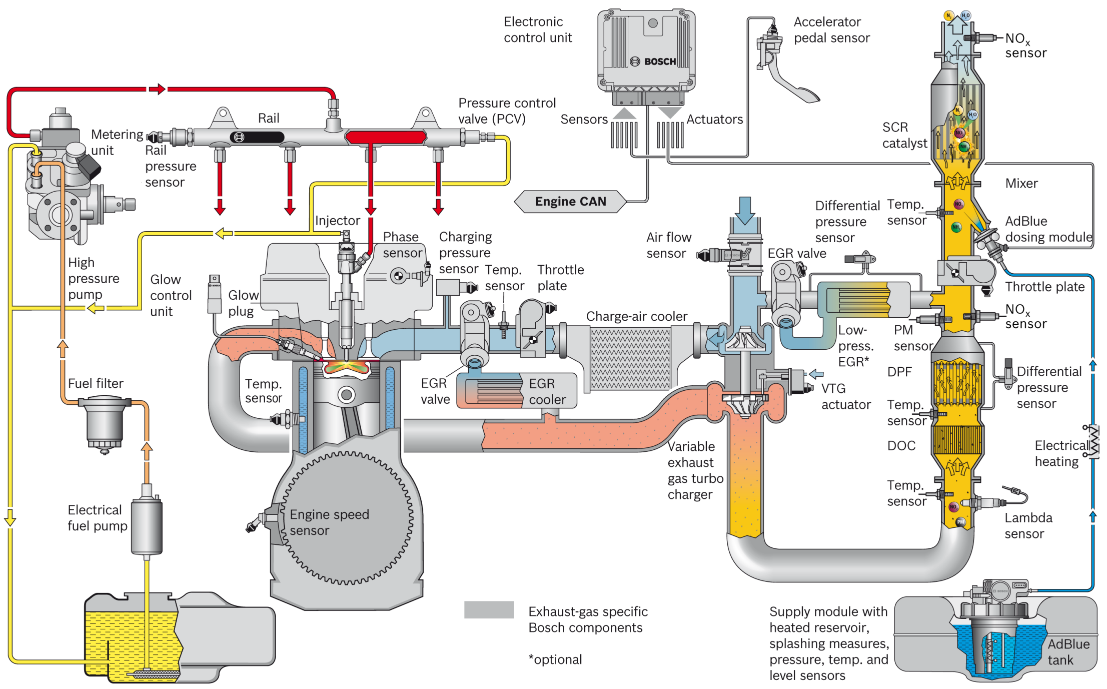


Figure 2.3: Overview of the multitude of diverse systems and their respective connection of an up-to-date Engine Management System for a diesel engine [Rob].

(oWCET). The maximum sample interval is determined by the Nyquist-Shannon sampling theorem [Lun16] and the environment. It is a lower bound for the frequency at which the runnable has to be executed. As a runnable represents a piece of sequential code, instructions in this code can call other runnables. When a runnable is only called from other runnables, it becomes a service. In automotive applications the resulting call hierarchy is limited in depth and recursive calls are typically not allowed. We assume for our purposes that this call hierarchy is flattened. I. e., the size and oWCET aggregate all service runnables that are called inside a runnable. The flattening of the runnable call hierarchy is often done by the compiler. The size of a runnable is the number of bits the runnable occupies in memory. The Observed Worst-Case Execution Time (oWCET) of a runnable denotes the upper bound of a runnable's execution that has been traced or measured. A comparison to the estimated and exact Worst-Case Execution Time (WCET) is illustrated in Figure 2.4. We assume that the execution time bounds of runnables are measured at a certain working point of the ECU and are thus *observed* bounds. For example, a working point could be a specific number of Rotations Per Minute (RPM) when in mode "driving". The current state of the practice in automotive is that software is typically not designed to allow tight WCET bounds, due to e. g., different operation points, control strategies and failure modes [BEGL05, MGL06, GE07]. Due to this complexity and the lack of predictable hardware, safe and tight execution time bounds are unavailable.

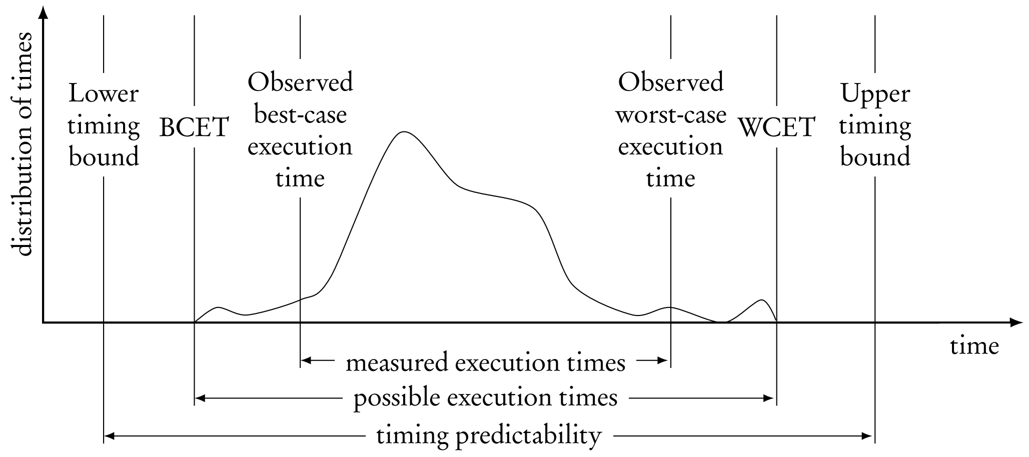


Figure 2.4: Execution time distribution of runnables with estimated, exact and observed best- and worst case boundaries [WEE+08].

Runnables have no input and output parameters (void-void), but communicate indirectly via *labels*. A label represents a data element in our application model. Labels are used as parameters, temporarily existing variables or represent constants and can be read and written from runnables at any point in time during their execution. A label $d \in \mathbf{D}$ can be of different data type, e. g., bit, array, or characteristic map and is stored in shared memory. Note that large data types such as a characteristic map are typically read-only and only segments are accessed. Besides the data type, other general properties of a label are: volatile(Boolean), constant(Boolean) and the initial value. Specific for the automotive domain is A2L which describes additional properties of labels for measurement and calibration purposes. The description standard A2L is extensively covered in Section 2.3.1.

When runnables access a label, these accesses can either be a read or write. Multiple reads or writes are aggregated using statistical values, i. e., how often a read or write is performed. The statistical values are typically extracted from source code using code analysis tools. Hence, when a runnable is executed, a read/write might be performed but does not necessarily have to. In our application model, these label accesses are unordered and do not follow a specific sequence within a runnable. Due to its complexity, automotive applications are highly communicative. Figure 2.5 shows a set of runnables from a real-world EMS, depicted as boxes and their respective communication, depicted as edges. Each edge represents a read-write relationship of two runnables via a label.

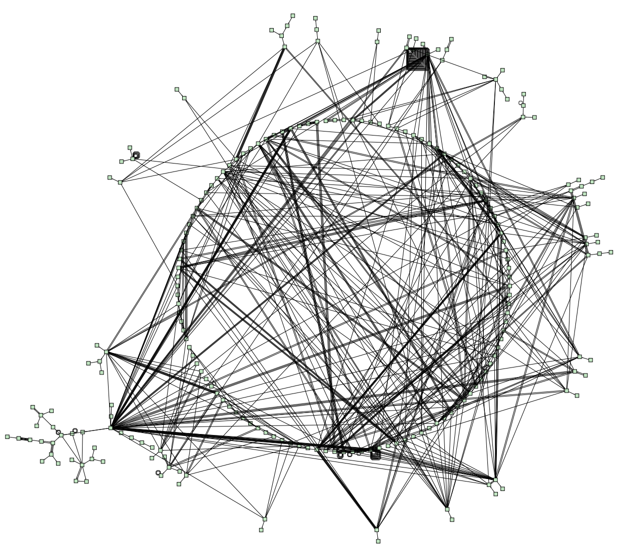


Figure 2.5: A typical, highly interdependent task with 229 runnables from a real-world automotive application. Each box represents a runnable and each edge a read-write communication via a label.

2.1.2 Real-Time Requirements

In the previous sections we have covered automotive applications in its embedded environment functionality-wise. In addition to the functionality, the timing is an important characteristic for automotive embedded systems. An application interacts with the environment through the hardware and connected peripherals such as sensors and actuators. When interacting, it has to respond to external stimuli within a certain amount of time, known as timing constraints. Thus, the correctness of a response does not depend solely on the correctness of its value, i. e., the logical result, but also on the time at which the value is made available. Responding too late or not at all is as bad as responding with a wrong value. The following definitions are based on Buttazzo's book on hard real-time computing systems [But11]. Buttazzo calls a system a *real-time system* when it "must react within precise time constraints to events in the environment". These precise time constraints, in the following called real-time requirements, are usually defined through experts of the domain by analyzing the functionality of the system and its environment. As stated before, real-time systems in the automotive domain are typically control systems. Hence, the control applications have to react to the environment ensuring certain real-time requirements. Hereby, a reaction typically implies a functionality that has to be provided to the environment and within the real-time requirements after a stimulus has occurred.

In a real-time system, this functionality is contained in a *task* t from the set of tasks $\mathbf{T}$ that is triggered by a stimulus and responds to this input by producing a result in time. Hence, a task is a container for functionalities, i. e., a set of runnables. A task is an abstract unit of scheduling and executes the containing runnables sequentially. The stimulus that triggers a task is mostly a recurrent stimulus. It could be an interrupt coming from a clock or from connected peripherals. For example, a typical trigger for a task that provides the functionality of sampling a sensor signal is a recurring interrupt from a clock. When and how a task is triggered depends on the control paradigm of the provided functionality. In either way, a task $t_i \in \mathbf{T}$ that is triggered by a stimulus releases a job $J_{i,k}$ from an infinite sequence of jobs $J_{i,k}, k \in \mathbb{N}$.

In automotive applications, tasks are typically recurring in the following way [But11]:

Periodic Task: A periodic task t_i is cyclically activated with a fixed time interval, the period P_{t_i}. The task releases jobs $J_{i,k}, k \in \mathbb{N}$ at a constant interval. Each job $J_{i,k}$ of task t_i is activated at $a_{i,k+1} = a_{i,k} + P_{t_i}$ with $a_{i,0} = 0$. In the automotive domain, the periods of the tasks are harmonic, i. e., a task's period is an integer multiple of the period of any other task. A representative example for runnables that are mapped to a periodic task is the sampling of a sensor signal.

Sporadic Task: A sporadic task is activated and releases jobs at a priori unknown times but its activations are separated by a minimum inter-arrival time. An example for runnables that are mapped to sporadic tasks are runnables depending on a specific angle of the crank in an automotive engine.

Our approach is independent of the job release intervals and can thus handle all kinds of tasks (i. e., also aperiodic tasks that release jobs with no pattern and irregularly). In examples and use cases we mainly demonstrate the approach on periodic and sporadic tasks.

Besides the stimuli of tasks and the recurrent creation of jobs, there are also requirements in real-time systems when a task has to respond. As stated before, the correctness of the response from tasks is based on the logical result of the computation and also on the point in time when the response is available. This specific point in time is called a deadline. A task t_i has a relative deadline D_{t_i} when a task must be completed and has to provide the desired response. The deadline D_{t_i} is relative and thus an offset to the activation time of the task $a_{i,k}$. Missing the deadline has different consequences and depends on the system and its environment.

In automotive real-time applications, we usually have the following kinds of deadlines [But11]:

Firm Deadlines: The response, that is produced after the deadline, is discarded or useless for the system. In automotive applications, sporadic deadline misses are often tolerable [ZH15]. An example is the EMS where e. g., the injection controller maintains stability despite missed deadlines and thus sampling losses due to the physical inertia of the combustion engine[1]. Systems with firm deadlines are also called weakly hard real-time systems.

Hard Deadlines: People are hurt when a deadline is missed. The most widely known example for such requirements is the airbag system. When the deadline of the airbag task is missed, the inflation module cannot provide the cushioning and restraint for the occupants during a crash event.

The approach presented in this thesis is applicable to tasks with both kinds of deadlines, hard and firm. Also, for our applications we assume constraint deadlines which are typical for automotive systems. In such systems, the relation between the period P_{t_i} and the deadline D_{t_i} of a task t_i is $D_{t_i} \leq P_{t_i}$. Figure 2.6 summarizes the notation of the presented real-time properties.

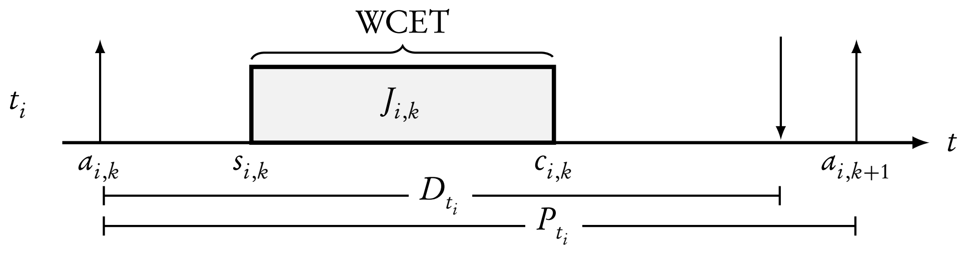


Figure 2.6: Real-time properties of a task t_i with a deadline D_{t_i} and a period P_{t_i}. The task is activated at $a_{i,k}$ and releases a job $J_{i,k}$ that starts its execution at $s_{i,k}$ until it is finished at $c_{i,k}$.

A task $t_i \in \mathbf{T}$ encapsulates runnables and is an abstract unit of scheduling. Runnables are thus mapped to a task by the function $m : \mathbf{R} \rightarrow \mathbf{T}$. In the automotive domain there is the established construction principle to map all runnables with the same recurrence pattern to a single task. This is most resource efficient on single-core processors and allows static control of dependencies between runnables with the same recurrence pattern. Inside a task, the runnables are executed sequentially, yet the runnables can be concurrent to each other. Let t_i be a task, then the order of execution of runnables is defined as the totally ordered set of precedences $\mathbf{P}(\mathbf{R}_{t_i}, \prec)$. A task can be divided in its structure by remapping the runnables to task partitions. We denote the j-th task partition by $t_{i,j}$ from the set of task partitions $\mathbf{T}_i$ of a task t_i. This means that each task t_i can be splitted into an arbitrary number of task partitions $\mathbf{T}_i$. A task partition requires the same properties as the task itself. Hence, each task partition in $\mathbf{T}_i$ has to have the same recurrence pattern and deadline as the original task t_i.

Besides being a container for runnables, a task is also the unit of scheduling for the underlying operating system. For automotive systems the operating system is typically

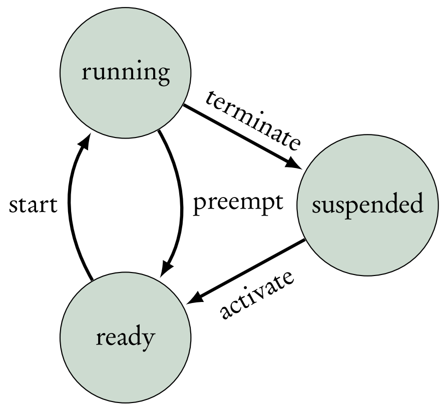
[1]Tobuschat et al. [TEHZ16] presented a timing feasibility test that exploits the robustness of control applications.

OSEK-compatible [OSE05]. OSEK (dt. Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug) is a standard for an open-ended architecture for distributed control units, e. g., ECUs in automotive vehicles. Among others, the objective of OSEK is to create independence from the hardware and implementation to provide portability and reusability. The scheduling algorithm provided by the OSEK Operating System (OS) is static, preemptive and uses priorities [ZH15]. In the automotive domain, the priority of a task t_i is based on the period P_{t_i} (also called: rate) of the task. The smallest period being the highest priority. This is known as a rate monotonic scheduler. Specifically, the OSEK-compatible operating system for the EMS discussed throughout this thesis uses an (offline) partitioned fixed-priority preemptive scheduling policy with a deadline-monotonic priority assignment scheme.

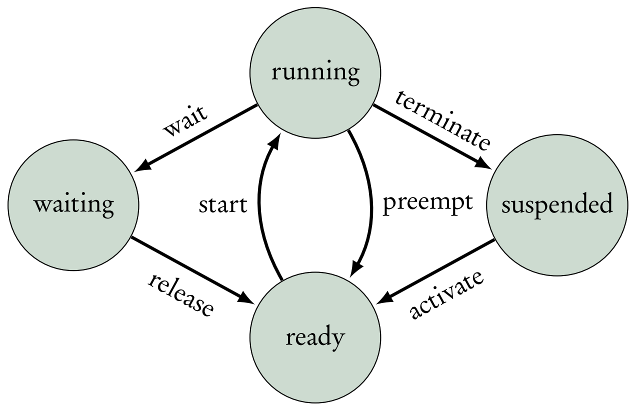
The concept of the OSEK OS knows two different task models for scheduling: The basic and extended task model. The task models differ in the state machine, as illustrated in Figure 2.7. The state machine of the basic task model in Figure 2.7a consists of only three states: running, ready, suspended. When a task is in the state running, the task is assigned to a core and instructions are executed. In the state ready, all functional requirements for running the task are met but another task is assigned to the predestined core, hence it is waiting for a scheduling decision. When a task is not activated, it is in the state suspended. In this task model, a task can never be blocked as there is no waiting state. Thus, it is primarily used for systems or functionalities that have no synchronization or complex I/O. In the extended task model, shown in Figure 2.7b, there are the same three states from the basic task model, plus an additional waiting state. In this state, a task is waiting of at least one event. This task model allows that tasks can be blocked and is hence used for complex systems with interfering resource accesses and communication. There is also an additional overhead when using the extended task model compared to the basic task model due to the event processing feature. It also increases the complexity when developing applications that use the extended task model because applications have to be analyzed for possible deadlocks. These are also the reasons why the EMS is using the basic task model.

Event chains are another approach to specify and describe real-time requirements. When scheduling real-time tasks and their runnables, there are also real-time requirements that follow the propagation of a signal through the system. A typical example for such a requirement is the braking system with brake-by-wire [BP03]. By law it is required that the actuators for the brakes respond to the brake pedal within a certain timeframe. As there can be many ECUs, subsystems, tasks and runnables between sensing the brake pedal and the actuation for the brakes, these timing requirements have to be specified over a chain of events. These *event chains* allow to specify timing information and requirements on causal loops of events through the system.

Event chains can reference highly diverse events but have to contain at least two of them: the stimulus event and the response event, e. g., a sensor to actor. An event can



(a) Basic Task Model



(b) Extended Task Model

Figure 2.7: State machines of the basic and extended task model defined in the OSEK OS in comparison [OSE05].

be the activation or termination of a runnable, it can be an interrupt, or an external stimulus from the driver. But the stimulus is always the first event of an event chain and the response is always the last event. All other events that are defined in between are called event segments and are optional. Event chains can have different end-to-end timing constraints. For example, a reaction constraint or an age constraint. Reaction constraints are one of the most widely used constraints for control systems and define how long after a stimulus a corresponding response must occur with a minimum and maximum value. Figure 2.8 gives an example of an event chain between a sensor and an actuator with a reaction constraint over multiple tasks.

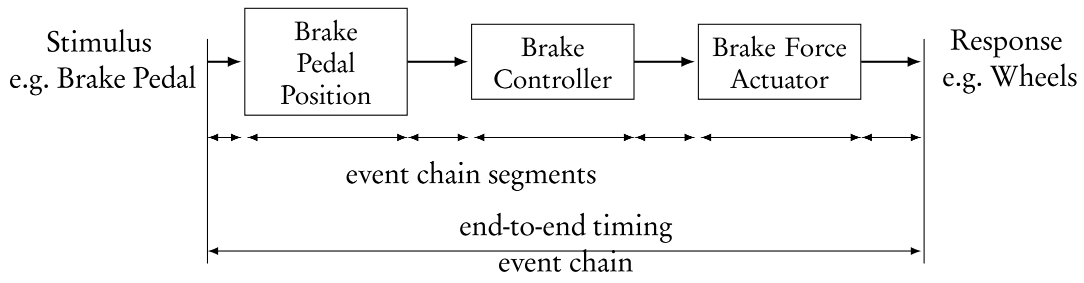


Figure 2.8: Example of a brake-by-wire event chain. A list of event chain segments from a stimulus to a response create an event chain.

Event chains and their corresponding timing constraints can be formally specified using the Timing Augmented Description Language (TADL) (presented in [FRNJ08], specified in [C$^+$09]). Event chains can be modeled on different levels of abstraction, from the vehicle level to the design level. In this thesis we focus on tasks and hence on event chains on the design level.

2.1.3 Multi-core Platforms

In the beginning of this chapter we argued that the software of automotive embedded systems follows an exponential growth. Modern automotive control algorithms become increasingly complex as a result of more comfort, safety and power-train functionalities. This software has to be integrated into ECUs. Hence, there is an increasing computational demand from the software to the hardware resources of an ECU. Processors in the consumer electronics industry already faced the technology change from single-core platforms to multi- and many-core platforms. This same change in technology is currently challenging the automotive domain [BDN+15], delayed due to longer development cycles and certification processes. Although the demand for more computing power has always been there, for consumer electronics as well as for automotive electronics, the boundary conditions for this technology change are different. Still, the primary motivation roots in the physics of our processors. This technology change is often called "the free lunch is over" [Sut05] and essentially describes the upcoming *power wall*. In the automotive context, the power wall (also known as the end of Dennard Scaling) is the main reason why single-core processors cannot reasonably cope with the computational demand. When increasing the clock speed of a processor, power consumption increases as well and leads to an advanced heat generation. Current automotive-grade cooling solutions are not able to dissipate this amount of heat and also the environment around an ECU does not allow advanced cooling mechanisms such as liquid cooling. Processors have to cope with difficult and alternating environment conditions, e. g., high temperature in an engine compartment. This excludes consumer electronics processors with high clock rates to be used in automotive in many cases. Besides that, advanced cooling solutions would be very expensive for automotive grade solutions. Furthermore, as transistors get smaller due to improved manufacturing, power density increases even more. Another way to provide computational power is to support Instruction-Level Parallelism (ILP). In processors of consumer electronics, ILP is a common feature. Yet, ILP increases the processor complexity and thus the power consumption while its performance gains are diminishing.

In the following we describe the characteristics of a typical architecture of multi-core platforms for the automotive domain. An automotive multi-core platform typically consists of a number of identical cores that are connected by a common shared resource. In contrast to single-core processors, the architecture of a core on a multi-core platform is much simpler. The cores have short in-order instruction pipelines, a reduced instruction set and preferably no speculative execution. The automotive domain has high predictability requirements and needs guarantees to satisfy timing constraints. Hence, components that improve the average-case performance are generally not integrated into automotive processors [CFG+10]. There is typically no hardware prefetching integrated. Prefetchers work in the background, consume interconnection network bandwidth and make latency computation difficult. While prefetching is great from a performance perspective, it should be avoided when deterministic systems are needed as within the automotive domain. Determinism is also the reason why automotive multi-core platforms have no caches or only caches for constants and code. Modern caches are very difficult to analyze due to high associativity and replacement strategies. Also, cache coherence becomes inefficient when the number of cores increases. Hence, a flat cache-less memory hierarchy is typically employed in automotive multi-core platforms which is also attractive when considering chip area and power consumption. However, besides the a processing unit on which a runnable is executed, a core contains also local memory for program code and temporary data. To obtain data from other cores or the global memory and flash, there is a common shared resource which is typically a bus or a cross bar. A bus architecture is often used because the needed chip area is small. In comparison, a cross bar provides a higher throughput but needs more chip area than a bus as it connects all components with each other. This is also the limiting factor when the number of cores increases. Example manufacturer for automotive multi-core platforms used in our EMS case study are JDP and Infineon with 3–6 cores. The block diagram in Figure 2.9 shows the architecture of a typical automotive multi-core platform.

It is important to note that our approach can also be applied to many-core platforms. Many-core platforms bypass the limiting cache-coherence mechanisms by e. g., using core-local memories as scratchpads in conjunction with a Network-on-Chip. Although the interconnect is different to multi-core platforms, its influence to our method is negligible.

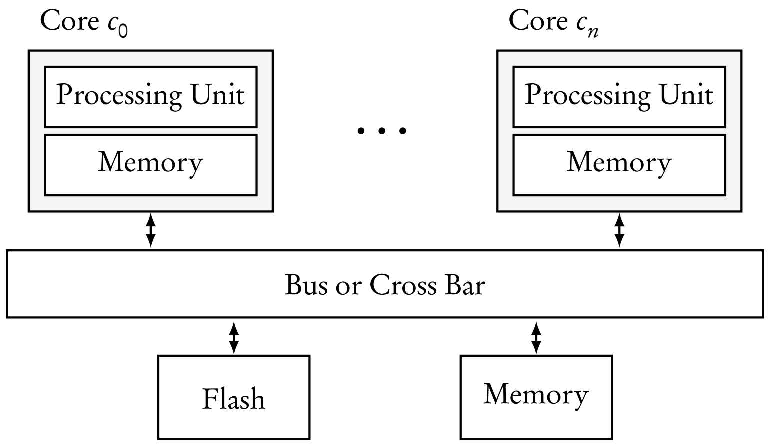


Figure 2.9: Block diagram of a typical automotive multi-core platform consisting of n cores, flash, memory and a bus or cross bar.

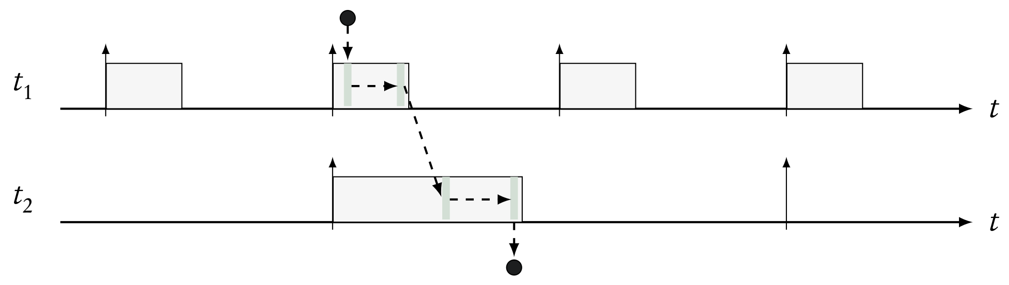
2.1.4 Model of Computation and Communication

As automotive control software is highly communicative, there is a tight link between the computation and communication. Especially the communication timing and its effect on control applications is of interest to us. While the runnables inside tasks encapsulate the functionality and hence the computational part, the communication between runnables always takes place indirectly via labels. There are two kinds of communication among runnables that any communication semantic of a Model of Computation (MoC) has to address. This semantic depends on the task the runnables are mapped to. When a runnable communicates to another runnable inside the *same* task it is called *intra-task* communication. When a runnable communicates to another runnable inside a *different* task it is called *inter-task* communication.

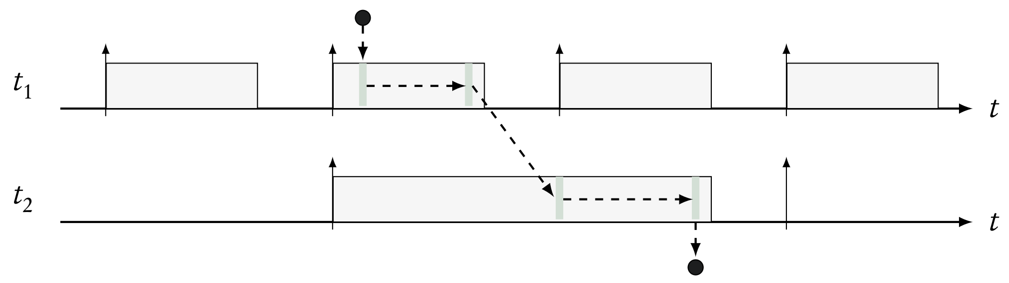
The most widely-used and established communication semantic in automotive is Last Is Best (LIB). LIB relies on the fact that all communication between runnables goes through labels that are shared in global memory. The data that is written to a label is available globally as soon as the writing instruction has finished. In other words, while a task is executing a set of runnables, the values of labels change continuously. Hence, by writing to a label its new value is immediately available to all other runnables as well as reading a label always transfers the latest values from shared memory. This applies to intra-task as well as to inter-task communication. The advantage of LIB is that it has generally short latencies after being optimized for the given platform. LIB was originally used and still is for applications running on single-core platforms. On multi-core platforms, this communication concept has some consequences.

The consequence of LIB on multi-core platforms is that runnables get *either* stale data (or there are even sampling losses) or fresh data. This communication semantic is not deterministic and depends on the distribution scenario, workload and order of execution inside a task [ZH15]. Figure 2.10 illustrates two cases on a multi-core platform in which runnables of a fast task t_1 communicate with runnables of a slower task t_2 using the LIB semantic under different workload scenarios. The figure shows that the point in time where the communication is executed is dependent on the execution time of the runnables and hence the task. In the average case, the communication of both tasks is executed within the period of task t_2 while in the worst case it takes almost twice the time. The drawback is that controllers have to be designed to cope with different timeliness of the data. Controllers have to be tested if they are stable and adhere to required performance criteria (e. g., robustness). Such a controller design is costly in terms of development effort and resource efficiency. For example, controllers would need to run at a higher rate to be stable enough. Also, data consistency has to be enforced by critical sections due to concurrent accesses from different cores. These sections have to be added manually and can lead to race conditions.

In contrast to LIB, the concept of Logical Execution Time (LET) [HHK01] provides a deterministic communication semantic among runnables in tasks. The advantage



(a) LIB communication on multi-core from fast to slow in the average case.



(b) LIB communication on multi-core from fast to slow in the worst case.

Figure 2.10: LIB communication on multi-core platforms from fast to slow tasks under different workload scenarios. The runnables indicated as green boxes communicate inside a task as well as across tasks. The communication between runnables is illustrated as a dashed arrow, start and end of all communication is depicted as black dots.

of Logical Execution Time (LET) is that its read and write semantic enforces a deterministic data exchange, for single- and multi-core platforms. As stated before, this is a very useful property for designing and composing control software[2]. The described communication semantic is also known as Timed Implicit Communication in the AUTOSAR environment [KQnBS15]. In the concept of LET, the same result/response is produced from the same input/stimulus independent of distribution, workload, or task execution timing (meaning e. g., the offset). Hence, LET can provide task-level determinism and guaranteed data consistency[3]. The benefit of LET is that it enables portability and allows to map tasks to different cores under the assumption that the WCET of a task is less than its LET interval [GSVK+06]. The only drawback is that the communication latency is fixed to the worst-case of LIB.

LET as used for deterministic multi-core platforms in the automotive domain separates the computation from the communication. The logical specification of a LET task consists of the following [GSVK+06]:

[2]Presented at the Embedded Multicore Conference (EMCC) by Jochen Haerdtlein

[3]The data consistency is dependent on the implementation of the copy operation at release and termination event.

- A sequential piece of code that has its own local memory space.
- A set of program variables (input variables), i. e., the labels that are read by the task.
- A set of program variables (output variables), i. e., the labels that are updated by the task.
- Some timing constraints, i. e., a release event and a termination event.

While the program variables are accessible by any task and thus global, the local memory space of a task is only accessible by itself. The timing constraints define the LET of a task, cf. Figure 2.11. Within the LET interval, a task can start later than the release event, complete before the termination event and can be preempted and resumed at any time inbetween. When a LET starts (the release event), all program variables that are input variables are read from global memory and copied to the local memory of the task. Likewise, at the end of a LET (the termination event), the program variables in the global memory are updated by the state of the output variables in local memory of the task. The process of copying the program variables to and from the global memory happens in logical zero time.

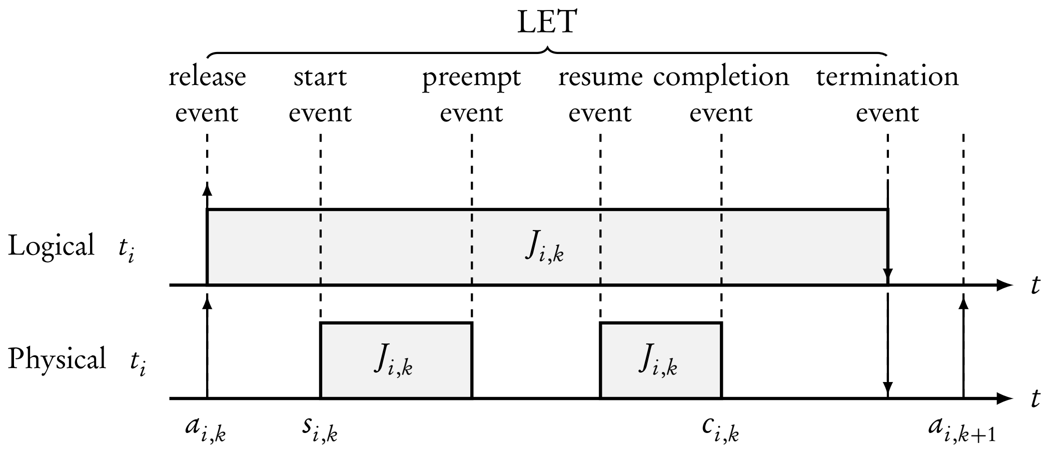
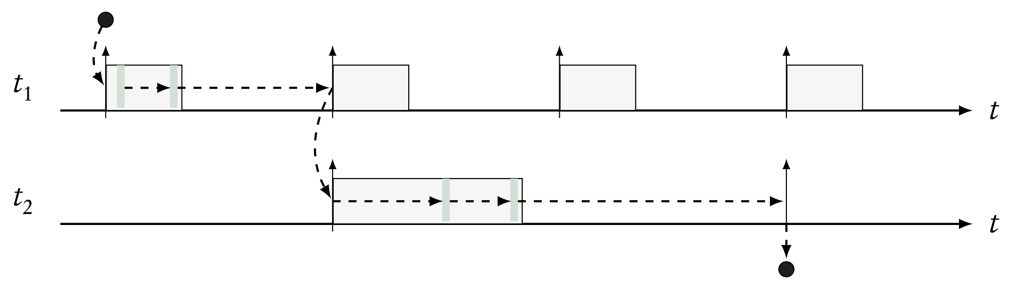


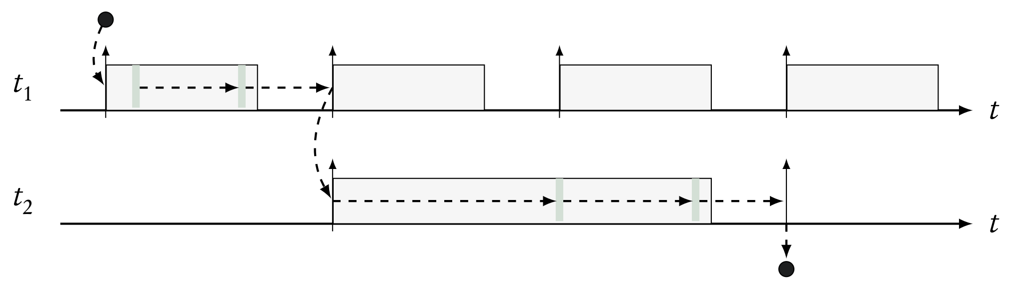
Figure 2.11: Definition and notation of a task's Logical Execution Time in relation to its physical execution.

In case of intra-task communication, the runnables simply read and write to and from the program variables located in the task's local memory. Within the task's LET this altered data is only available to the task itself.

The inter-task communication between jobs happens logically instantaneous at fixed points in time: at the beginning and end of each LET. Figure 2.12 shows two cases on multi-core platforms under different workload scenarios that follow the LET semantic when communicating. In both cases the boundaries represent the beginning and end of a LET for the corresponding task. At each write, the label is only altered in the local memory of the task. As soon as the LET is finished the global memory is updated with the state of the local memory of the task. At this point in time, the altered values of the labels are available to other tasks. Figure 2.12 also illustrates that this behavior is independent of the workload.



(a) LET communication on multi-core from fast to slow in the average case.



(b) LET on multi-core from fast to slow in the worst case.

Figure 2.12: LET communication on multi-core platforms from fast to slow tasks under different workload scenarios. The runnables indicated as green boxes communicate inside a task as well as across tasks. The LET interval of a task is the time between subsequent activations. The communication is illustrated as a dashed arrow, start and end of all communication is depicted as black dots.

For our approach we use LET as a model of computation and communication with the following notation, cf. Figure 2.11. Let t_i be a task with a deadline $D_{t_i} = P_{t_i}$, then for a job $J_{i,k}$, the LET starts with its activation $a_{i,k}$ and ends latest with the next activation $a_{i,k+1}$. For tasks with a constraint deadline $D_{t_i} \leq P_{t_i}$ holds and the termination event is the deadline D_{t_i}. For our approach, the point in time when updates of program variables are made is highly important, i. e., when the updated labels are available and to whom they are available. We call this point in time the publication event.

For intra-task communication the communication is defined as follows. Let m be a mapping such that runnables r_a and r_b are mapped to the same task $m(r_a) = t_i$, $m(r_b) = t_i$ and runnable r_a writes to label d_v that is read by runnable r_b. I. e., r_a communicates with r_b and $r_a \prec r_b$. In this case, as soon as runnable r_a finished writing to label d_v in the task's local memory, the updated value of d_v is immediately available to r_b and all other runnables that precede r_a within the task. The updated value of d_v is published during the execution of r_a to all runnables $r_k \in \mathbf{R}_{t_i}, r_a \prec r_k$. For runnables that precede runnable r_a in the job instance $J_{i,k}$, the updated value is available in the following job instance $J_{i,k+1}$. This is similar to the inter-task communication, which we explain in the following.

The inter-task communication with LET is defined in the following way. Let m be a mapping such that runnable r_a is mapped to producer task $m(r_a) = t_p$ and runnable r_b to consumer task $m(r_b) = t_c$. Also, runnable r_a writes to the label d_v that is read by runnable r_b, i. e., r_a communicates with r_b. Again, we assume harmonic periods, i. e., the period of one task is always an integer multiple of the period of any other task. As soon as runnable r_a finishes writing to label d_v, only the corresponding program variable located in the task's local memory contains the updated value of d_v. This value is published instantly at the earliest of the termination event of the producer task t_p. Hence, the updated value is available to r_b, all following jobs $J_{p,k+1}$ of the same task and all other runnables outside the producer task t_p.

As already mentioned, LET fixes the communication latency to LIB worst-case. Hence, the transition from LIB to LET introduces a delay in communication for most cases. When doing such a transition, the application has to be checked for increased latencies that exceed the worst-case scenario of the LIB implementation or affected critical paths that do not allow additional latencies. Our approach is based on applications that are fully transitioned to LET. We assume that all additional latencies, critical paths, or event chain constraints have been checked and do not violate any requirements of the system.

2.2 Software Development Process

In the automotive industry, the development of systems, hardware and software typically follows the V-model [SZ16]. The V-model [KKuBdBfIidB98] is a phase-based development process model and an extension of the waterfall methodology. It is defined in multiple standards that differ in abstraction and domain e. g., ISO 12207, V-Modell 97, V-Modell XT. The idea of the V-model is the separation of the specification, implementation and integration phases, cf. Figure 2.13. The model requires that results of a phase are complete before entering the next phase. For example, the requirements of design or architecture clearly have to be defined before going into the next phase, the implementation. Hence, the V-model only defines activities and results and excludes the order of the activities. The V-model is also divided by layers such that each phase during the specification has its counterpart within the same layer during integration.

For example, for the architecture design there is the counterpart that provides unit-tests during integration.

The benefit of the V-model is that it improves the process of planning the development with its clear process lifecycle. This is especially important for the automotive industry with its heavy supply chain. Figure 2.13 shows the V-model when developing a car. The Original Equipment Manufacturers (OEMs) are typically responsible for the top layers of the V-model and set up the requirements for a car and integrate the products from the suppliers, while the suppliers are responsible for the bottom layers of the V-model. The phases of the V-model can also be aligned to other major standards, e. g., Automotive SPICE, the domain specific variant of the ISO/IEC 15504 standards to assess the development process. Besides the OEMs, also the suppliers typically use the V-model to develop the supplied products that are integrated by the OEM. In fact, the V-model can be used for various products and in different shapes and sizes. When focusing on a single ECU, the development process follows the Double-V-Model having one development process for the hardware part of the product and one for the software part. In this approach we refer to the V-model as the software development process of an ECU application.

2.2.1 Model-based Development

As the systems and software of ECUs in the automotive domain gain more and more complexity over the last decades, so has the process of developing these systems. In the early days, these systems were developed within the domain of mechanical engineering. Today, automotive systems development is a highly interdisciplinary challenge at a unique complexity level. In the previous sections we showed that standards such as AUTOSAR can contribute and ease the system development by abstraction and au-

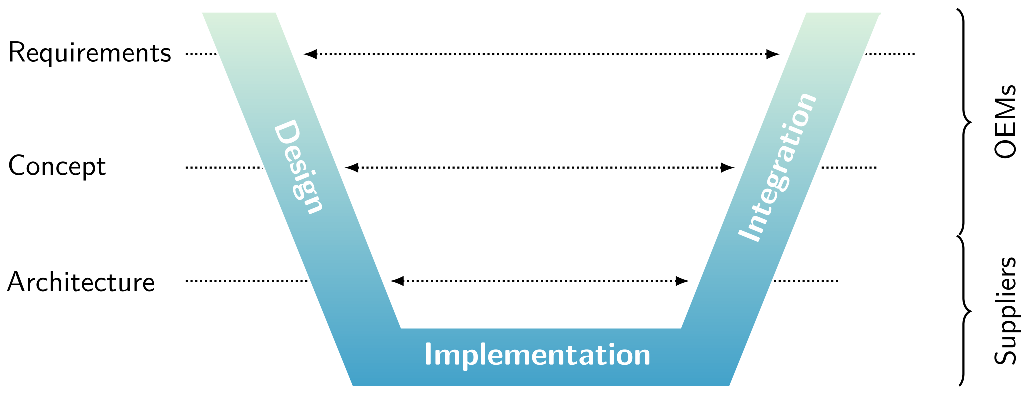


Figure 2.13: Example of a simple V-model for the development of a car that is devided into design, implementation and integration [SZ16].

tomation. Another concept that targets the problem of efficient systems development is the model-based approach. The model-based approach also abstracts and automates the systems' development process, ideally to cope with complexity. And because of its benefits, model-based development has become a standard methodology in the automotive industry [BKK+13]. Modeling aims at improving the effectiveness of engineering by using models as important artifacts in the development process and not solely for documentation purposes. Modeling enables early verification and validation activities on various stages which reduces costs, supports generation of lower-level artifacts (e. g., source code) and can be used for communication with other multidisciplinary stakeholders [SZ16].

AMALTHEA [ITE] is a tool platform for embedded multi- and many-core software systems engineering. It is an ITEA2 funded project, created in 2013, and primarily focuses on, but not limited to, automotive systems. The main goal of AMALTHEA is to enable efficient software engineering especially for multi-core platforms. As a model-based development platform, it promotes and simplifies data exchange in cross-organizational projects. Figure 2.14 illustrates this benefit. Complex tool chain elements such as simulation and validation can be managed easily. Hence, products and knowledge from tool vendors and other suppliers can be integrated efficiently. This allows to explore the decision space extensively and to support decisions in designing automotive systems with accurate information [WKH+15].

The AMALTHEA system model contains information on the software, hardware, timing behavior and constraints for the embedded system under development. It hereby extends AUTOSAR by dynamic architecture details. AMALTHEA and AUTOSAR share the same concept of tasks, runnables, components and interfaces, yet with a different implementation. Also events, event chains and timing constraints are implemented in both platforms as they share a common origin.

2.2.2 Legacy Software

Since the beginning of E/E (electric/electronic) systems in cars, embedding additional systems with software has been a major trend for the automotive industry. While the number of individual systems and ECUs has increased, also the communication infrastructure had to grow with it. In 2010, innovations in the form of electronics and software inside an automotive are at a staggering 90 % [Fü10]. Premium-class cars integrate close to one hundred million lines of code in 2009 [Cha09]. Whereas a low-end car already embeds 30–50 ECUs, a premium-class car consists of up to 70 ECUs that are connected via 5 different system busses [Fü10]. When focusing on a single ECU, the automotive supplier Bosch has been experiencing a 15 % increase of load per year in an ECU for an EMS on average. This number is obviously depending on the project and segment, but the increase in software on ECUs is nevertheless steady and significant. However, decades of ECU software development have also left a huge amount

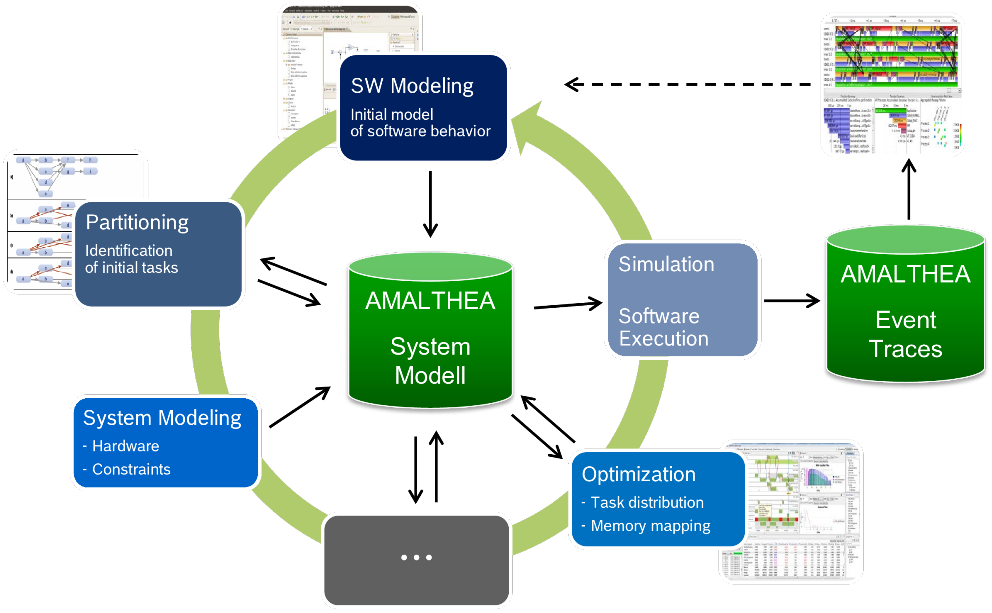


Figure 2.14: Model-based development using the AMALTHEA platform [ITE]. The common AMALTHEA system model is the foundation for all activities on the model.

of legacy software. This legacy software is built for single-core platforms and cannot be executed on multi-core platforms without issues. It needs to be migrated to multi-core platforms. Yet, a complete redesign and reengineering of the legacy software for multi-core platforms is prohibitive due to cost and time-to-market reasons. The transition from single-core to multi-core platforms also carries a paradigm change which we explain in the next paragraph.

Previously with single-core platforms, communication followed the blackboard principle. Memory accesses were virtually for free because the single global memory had a very low access delay. Hence, the challenge when integrating software was solely the scheduling of the computations. There were little to no consistency issues due to a single master that handled all communications. As a rule of thumb, it was more efficient to store or communicate data than to compute it. On multi-core platforms, communication between the different cores is expensive. Furthermore, there is the need to synchronize access to shared resources which brings a high overhead with increasing number of cores. Hence, the waiting time for resources and communication increases.

As mentioned before, scheduling the computations is a major challenge on single-core platforms. Hence, construction principles were created to efficiently map runnables to tasks. An important and established construction principle is to allocate runnables with the same activation pattern to a single task. This is most resource efficient for single-core platforms and allows static control of dependencies between runnables with the same activation pattern. Also, runnables follow a fixed and optimized order within a task. When migrating this single-core software to multi-core platforms, the fixed order within a task limits the parallel execution. The fixed order ignores the increased communication costs on multi-core platforms and hence causes a decreased utilization due to higher waiting times.

Another issue when migrating is that preemptions via priorities are used heavily on single-core platforms to get a computing resource. Priorities are used as a tool to enforce implicit assumptions for data consistency requirements or scheduling policies. This is also the reason why there are no precedence constraints between tasks of different periods. Yet, when migrating to multi-core platforms, tasks with the same activation pattern may run in parallel. They can even jitter in their activation because of different clocks on the cores. However, the fundamental assumptions that originally created an optimized scheduling policy with priorities are unspecified. This can lead to data inconsistencies, increased overhead due to additional preemptions or synchronization, or even functional issues when precedence constraints are not enforced.

In summary, migrating legacy software that is optimized for single-core platforms to multi-core platforms by distributing the functionality is a complex problem. The legacy software controls inherently concurrent physical systems through highly optimized task schemes and scheduling. However, the foundations for these optimizations are not explicitly known. Yet, these foundations are needed to migrate legacy control software to multi-core platforms due to the strong link between computation and communication.

2.3 Standards

There are numerous contributors for hardware, software and other parts for embedded automotive systems. In fact, the list of suppliers that provide parts of a car to an OEM seems endless. Only the adoption of established standards can streamline this huge variety of contributions and the corresponding development effort. The standards that govern the automotive industry are mainly influenced by four factors: Technology such as AUTOSAR [AUT], processes like the ISO 26262 [Int11] for functional safety of road vehicles, organization such as the improvement program Capability-Maturity-Model-Integration (CMMI) [cmm] and regulations e. g., for emissions.

In the following sections we give a brief overview of standards important to this thesis and how they interact with each other. We focus only on the essential ones and describe the foundations they provide to this approach. Also, many standards have a close relationship and build on each other. For a better understanding of the context and landscape of automotive standards, Figure 2.15 presents a historical overview of the set of standards described in the following.

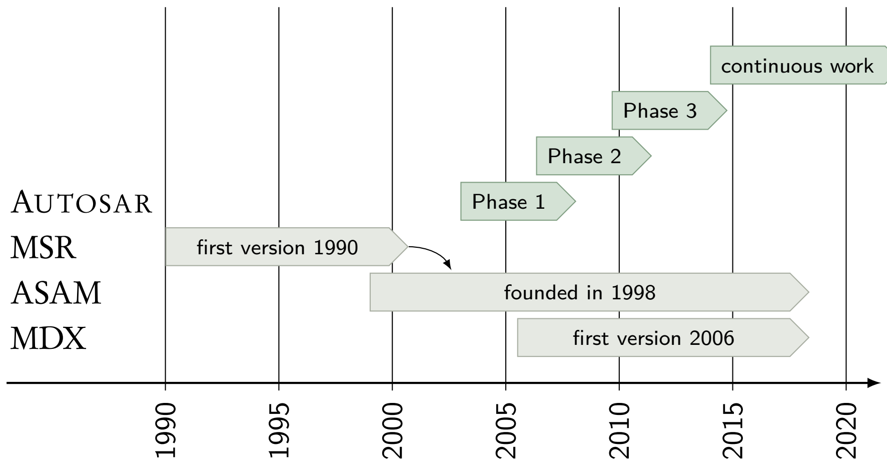


Figure 2.15: Timeline of the set of standards used for this thesis. Starting in 1990 with MSR up to today's standards such as AUTOSAR and ASAM [AUT, ASA15b, MSR].

2.3.1 ASAM

In 1990, the Manufacturer Supplier Relationship (MSR) [MSR] was founded as a joint project of several major automotive organizations and suppliers in the German industry. Its goal has been to improve the collaboration among the members of the project for the electronic systems development. It should help describe methods and realize structured and tool-based interfaces among the project partners. The standards developed by MSR are used for describing automotive electronics and are based on SGML and later on XML.

MSR was then merged into ASAM MSR in 2001. The Association for Standardization of Automation and Measuring Systems (ASAM) [ASA] was founded in 1998 and is an incorporated association under German law. It was primarily founded by international automotive manufacturers, suppliers and engineering service providers and has more than 140 members today. The standards created by ASAM define protocols, data models, file formats and Application Programming Interfaces (APIs) for development and testing of automotive ECUs. ASAM also works closely with the ISO and AUTOSAR organization.

MDX

MDX was initially published in 2006 by ASAM. The Model Data Exchange Format (MDX) [ASA15b] is an XML-based description of interfaces of functions, their data (variables and calibration parameters) and scheduling information for ECU software. The benefit of MDX is the integration of functions as a black box into the overall

ECU software. The functions can be integrated as object code without having access to the source code which is usually the intellectual property of the various automotive suppliers. ASAM MDX is used by the suppliers and OEMs for data management and documentation purposes. It is hereby typically implemented as an exchange format for the in-house development tools.

MDX belongs to the group of the Measurement, Calibration and Diagnostic (MCD) standards. Related standards in this group are ASAM MCD-2 MC (covered in Section 2.3.1), ASAM CDF and ASAM MDF which define calibration parameters and measurement variables. ASAM MCD-2 MC and ASAM MDX use the same semantics for the same elements, yet they are applied for different use cases. The AUTOSAR Software Component Template (SWC-T) (AUTOSAR is described in Section 2.3.2) is strongly influenced by ASAM MDX and target a similar use case but in a different ecosystem. While MDX contains the data constructions, ASAM FSX contains the behavioral description of software functions.

There are various use cases and file types defined in ASAM (MSR/MDX). For our approach, we use PaVaSt and DySched which both are originally developed by Bosch and merged into ASAM (MSR/MDX).

PaVaSt The Parameter Variable Structure (PaVaSt) is a technical data specification used for software integrated into ECUs. It specifies calibration parameters, variables, messages, classes, system-constants and more. All objects that require to generate declarations, that are relevant for the calibration engineer and which are needed for interface validation are handled by PaVaSt. For each function in PaVaSt there is a description of its interface, used data and relationships. In MDX it is represented as the SW—DATA—DICTIONARY—SPEC element.

For our approach, PaVaSt is important as it is the data specification for the labels used in our application model. The PaVaSt Data Dictionary contains the data structures as well as required auxiliary objects for each data element within the ECU. We focus on the following dictionary elements:

- SW—VARIABLES: Specifies the characteristics of variables and means of communication, e. g., the value, the type, its scope etc.
- COMPU—METHODS: The method to apply when converting a physical value into an ECU internal value and vice versa.

These dictionary elements are the source for the label's properties in our application model.

DySched The Dynamic Scheduling (DySched) describes the mapping of functionalities, i. e., runnables, to tasks and in which order they are executed. In MDX, the element SW—SCHEDULING—SPEC is a superset of DySched and includes the description of the tasks, the mapping and order of the runnables inside a task. Together it creates the scheduling architecture of an application. For our application model, the task set **T** and the mapping function $m : \mathbf{R} \rightarrow \mathbf{T}$ are specified by DySched.

MCD-2 MC

ASAM MCD-2 MC [ASA15a] also belongs to the group of MCD standards. The standard is titled “Data Model for ECU Measurement and Calibration” and contains data types, record layouts, dimensions and memory locations of ECU variables in its corresponding A2L file format. This information is needed for the calibration of the ECU software. ASAM MCD-2 MC supports the separation of the process of measurement and calibration with the process of software development. It provides a description of the ECU targeted specifically for calibration engineers.

While being independent from ECU-internal data formats, ASAM MCD-2 MC describes the physical understanding of the parameters in an ECU. Using this understanding, our approach can improve the parallelization process.

2.3.2 AUTOSAR

The AUTomotive Open System ARchitecture (AUTOSAR) [AUT] is a consortium and worldwide automotive development partnership. It was founded in 2003 and has been developed in multiple phases since then. Members of the consortium are OEMs, suppliers of different Tier stages and also tool suppliers. AUTOSAR aims to create an open and standardized software architecture for automotive ECUs. The specifications established by the consortium describe the whole vehicle ECU development with its architecture, methodology and application interfaces. This is also the major difference to ASAM MSR/MDX that focuses only on a single ECU. AUTOSAR pursues the objective to increase reuse and exchangeability between OEMs and suppliers, to manage complexity of ECU architectures and to improve scalability and transferability of software components. Its ultimate goal is to establish a collaboration between various partners throughout the whole product life cycle.

AUTOSAR defines a three-layered architecture, illustrated in Figure 2.16. This architecture allows to develop software components that can be integrated into vehicles of different OEMs, variants or product generations. This leads to a high reliability of the overall vehicle’s system with cost reduction and capacity benefits. Note that the AUTOSAR operating system is primarily based on OSEK OS.

The three layers of the AUTOSAR architecture are (bottom-top):

- Basic Software (BSW): This layer represents a common definition for hardware related software. These are for example components such as controller peripherals, various drivers and abstractions for memory or I/O.

- Runtime Environment (RTE): This layer abstracts from the inter- and intra-ECU communication and acts as a middleware between the ASW and the BSW. As a middleware the RTE is responsible to provide a uniform environment for the ASW such that any implementation is independent from communication mechanisms and the hardware.
- Application Software (ASW): This layer includes the application that consists of AUTOSAR software components and AUTOSAR ports. The software components are an atomic piece of software and the ports define the interface for communication among the components.

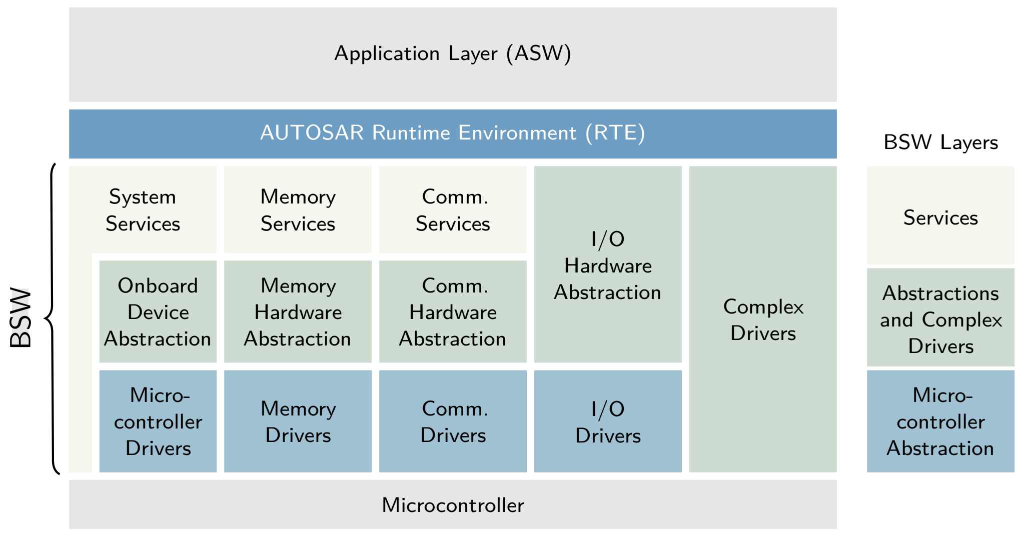


Figure 2.16: AUTOSAR architecture overview of the basic software, the runtime environment and the application software [AUT].

This standardization of interfaces across manufacturers and suppliers and between the different software layers builds the foundation for achieving the goals of AUTOSAR.

3 Related Work

Parallel Architectures have been around for decades in many different forms [Sar89, CSG99]. The spectrum of related research on the topic of software parallelization is therefore very broad. Parallelism can be applied at very different levels of hardware and software designs and strongly relates to many architectural concepts. In the following two sections we survey and discuss existing coarse-grained parallelization approaches specifically for complex embedded systems. The focus of the first section is on fully automated approaches, while the second section discusses approaches that interact with experts from the corresponding domain to integrate additional domain knowledge.

3.1 Automatic Parallelization

The primary problem for an efficient parallelization is to partition the application into smaller processes that can be assigned to different processor cores. Finding the optimal assignment of processes to a limited number of partitions is a computational challenge because the corresponding problem is NP-complete [Kar72, GJ90]. As a consequence, heuristics are needed to find near-optimal solutions for real-world applications in reasonable time. The sequential software applications are hereby commonly represented in the form of a Directed Acyclic Graph (DAG). The nodes of the DAG represent the smallest entity that is executed sequentially while the edges indicate communication or causal dependencies among the nodes. Typically, the node weight denotes the execution time and the edge weight the communication costs. Over the last decades, a wide range of partitioning heuristics using DAGs have been proposed. These heuristics utilize various techniques such as branch-and-bound, integer-programming, graph-theory, randomization, genetic and other evolutionary methods. Kwok and Ahmad [KA99] and McCreary et al. [MKTM94] give an overview and compare a remarkable set of algorithms.

In the last years, more advanced graph representations have been developed. Cordes et al. [CMM10] presented an approach to extract coarse-grained task level parallelism from sequential embedded software written in C. It is based on the intermediate representation of the Hierarchical Task Graph (HLT). It provides an abstraction to make the parallelization step manageable and is similar to the representation by Girkar and

Polychronopoulos [GP94]. A HLT consists of simple nodes as basic statements, hierarchical nodes for loop or function bodies and communication in- and out-nodes. The edge weights represent the communication costs and the average execution time of a statement is the node weight. The execution time of a statement is measured via a cycle accurate simulation of the target platform. Cordes et al. published approaches for homogeneous [CMM10, CM12] as well as heterogenous architectures [CNEM13, CENM13] that are either single or multi-objective. The approaches with a single objective to minimize the execution time incorporate Instruction-Level Parallelism (ILP). Genetic algorithms were used to find a parallel solution with multiple objectives such as execution time, energy consumption, communication costs or load balancing. Experiments on picture and video codecs as well as several industrial benchmarks have validated the approaches.

The previously surveyed approaches by Kwok and Ahmad, McCreary et al. and Cordes et al. were presented for embedded systems in general but cannot be directly applied to automotive systems. In the automotive domain, parallelization is typically applied on the task-level where runnables are mapped to multiple tasks that in turn are assigned to processor cores. In particular, when parallelizing existing automotive applications, the legacy design has to be considered to avoid functional changes. We survey automotive-grade solutions in this section that respect automotive architectures and their ecosystem.

The approach by Höttger et al. [HKI15] derives dependencies from read and write accesses using the application model AMALTHEA [ITE]. These dependencies and together with the specified runnables form a Weighted Directed Acyclic Graph (WDAG). This WDAG contains the communication costs as well as the computation costs from the application model. The presented approach assumes tight and safe upper bounds of the execution times. Höttger et al. presented two model-based parallelization heuristics that consider the critical path of the system and earliest start scheduling. Both heuristics maintain the behavior by respecting the ordering constraints that are derived from the application model. The presented approaches specifically address automotive embedded multi-core systems and were evaluated on a real-world automotive application e. g., Engine Management System (EMS). However, the work by Höttger et al. assumes tight and safe Worst-Case Execution Times (WCETs) that are unavailable to us due to the complexity of our EMS and the lack of predictable hardware.

Several authors proposed parallelization approaches based on the AUTOSAR application model which we discuss in the following. An overview of the general challenges related to AUTOSAR when migrating to multi-core platforms is presented by Becker et al. [BDN^{+}15].

During the earlier phases of AUTOSAR, Long et al. [LLP^{+}09] presented a ruleset based on the mapping rules of the runtime environment to map runnables to tasks. The rules have the objective to reduce the communication of AUTOSAR application.

The work also provides an equation to evaluate the efficiency of the mapping solution but no mapping algorithm was proposed.

The work by Zhang et al. [ZG11] proposes a runnable to task mapping for AUTOSAR software components using a variant of a genetic algorithms. Yet, the process of mapping the runnables does not consider multi-core platforms.

A heuristic algorithm to map runnables to tasks and tasks to cores was presented by Monot et al. [MNBSL12]. The motivation to this work is the integration of AUTOSAR software from different sources onto one Electronic Control Unit (ECU). The runnables are grouped into clusters and allocated to cores while considering locality constraints and load balancing. In a second step, the runnables of a core are sequenced by assigning the least loaded slot. Whereas the approach respects locality constraints due to e. g., shared variables, it does not consider inter-runnable data dependencies. Based on the analysis of our EMS software, these data dependencies between runnables are however the main limitation when parallelizing. With our approach we can find the most limiting inter-runnable dependencies and evaluate its usefulness together with an expert.

RunPar presented by Panić et al. [PKQn$^+$14] focuses on the allocation problem of runnables from AUTOSAR applications on homogeneous multi-core processors and optimizes the load balancing among the cores. It utilizes the idea of runnables being the unit of scheduling to allocate dependent and independent runnables to the least utilized core while maintaining the execution order of the legacy task. The RunPar assignment is based on a variant of the worst-fit decreasing heuristic and assigns runnables of the same task to different cores. As RunPar assumes safely bounded WCET, the runnables are statically scheduled such that no run-time synchronization is needed. The result follows the task ordering of the behavior of the legacy application and does not allow runnables from different tasks to be executed in parallel. Hence, the same functional behavior is guaranteed on single- as well as on multi-core platforms and the AUTOSAR application configuration is maintained. The presented approach was evaluated on a real-world automotive EMS. In contrast to the assumption of safely bounded WCET in this approach, we only have unsafe Observed Worst-Case Execution Times (oWCETs) of our EMS such that run-time synchronization is needed. However, our approach can reduce the overhead due to run-time synchronization without compromising the functionality.

The presented approach by Faragardi et al. [FLN13, FLSN14b, FLSN14a] is also using the AUTOSAR ecosystem. Based on a runnable interaction graph, a set of runnables, their precedences and the rates of communicated data are modeled [FLN13]. This graph is used to map the containing runnables onto multi-core platforms while minimizing the overall communication costs and ensuring all timing and precedence constraints. The presented model and mapping heuristic [FLN13] was extended by an evolutionary algorithm inspired from simulated annealing to allocate the tasks to cores [FLSN14b].

It assumes homogeneous multi-core platforms and tight and safe WCET bounds which is different from our approach that has to cope with oWCET. In these approaches, the task creation as well as the allocation of the tasks to cores complement each other and refine the solutions. The experiments to evaluate the approach were performed using a set of randomly generated applications which are supposed to be executed on a multi-core platform with 4–6 cores.

Similar to the procedure by Faragardi et al. is the work by Saidi et al. [SCCM15]. The AUTOSAR runnables are not mapped to tasks that are in turn allocated to cores, but are directly mapped to cores. Based on this mapping, the runnables are assigned to tasks which are then internally sequenced. The authors developed an Integer Linear Programming (ILP) formulation of the mapping problem that minimizes inter-core communication and balances the processor load. The presented approach was evaluated using a steer-by-wire application with 22 runnables but the authors did not consider the scalability with more complex applications. Our approach can handle one of the most complex automotive applications, an EMS.

The approach presented by Hennig et al. [HvHM+16] is very similar to our approach. Their work is also motivated by the increased computational workload of tasks and builds on Logical Execution Time (LET). Their approach extracts a dataflow graph from an existing AUTOSAR task that contains runnables and forward as well as backward dependencies in the form of edges. This dataflow graph is similar to our dependency graph as we share the same semantic for the runnables and dependencies. However, we only allow the manipulation of the graph's dependencies when the manipulation has been verified by a domain expert. In the next step, clusters are identified in the dataflow graph. The runnables within a cluster must not have any forward dependencies between them. They are only allowed to have forward dependencies to runnables of another cluster. This enables the free distribution of runnables of a cluster to different cores. The runnables are distributed and sequenced according to the legacy specification using Timing Description Language (TDL). While the work by Hennig et al. is similar to ours, it does however not cope with oWCET.

3.2 Parallelization using Domain Knowledge

It is critical to maintain all dependencies and their respective timing while parallelizing. Unfortunately, the amount of dependencies in automotive application is huge. As a consequence, these dependencies limit the solution space when parallelizing a legacy task. Yet, some dependencies are based on past design decisions in regards to the legacy system and not physically required. This provides the opportunity to find and review these dependencies with the goal to remove the limitations on the parallelization. In the following we survey related work that introduces domain knowledge to improve the parallelization process.

There is a lot of research in the area of parallelization by using software engineering methods that introduces domain knowledge. The general and well-established PCAM approach (partitioning, communication, agglomeration, and mapping) was proposed by Foster [Fos95]. The idea is to decompose the problem into as many parts as possible, then communication is introduced, the combined result is evaluated according to trade-offs and finally mapped to the processor platform. Similar to this approach is the well-known method described by Culler [CSG99]. It also starts with the decomposition of the computational problem into small parts, then combines these parts into tasks, introduces communication and synchronization mechanisms and maps the result to the processor platform.

The general idea of parallelization presented in these textbooks by Foster and Culler can be found in many publications to some degree. However, most of these publications on parallelization presented approaches that are performed manually. Due to the huge amount of legacy software in our domain, a manual parallelization is too expensive. In the following we discuss parallelization approaches that are semi-automatic and hence supported by procedures that are executed automatically. The parallelization can either be guided by a software engineer respectively domain expert or the domain knowledge is requested during the approach.

As part of the EU funded parMERASA project [UBG+13] the patternsupported parallelization approach was developed [JGU13b]. The approach is an extension of the one described by Jahr et al. [JGU13a]. It provides a methodical model-based migration from sequential legacy software to a parallel representation for multi-core platforms that is executed by an engineer or software developer. One of the main goals is that existing legacy code is migrated and hence improved in performance by applying parallel design patterns. The pattern-supported parallelization approach is based on two principals: the construction cycle of Foster [Fos95] and the usage of patterns as building blocks by Mattson et al. [MMS01]. In contrast to the work by Foster and Mattson et al. that targets high performance computing, the pattern-supported parallelization approach focuses on embedded systems. In its first phase, the platform independent Activity and Pattern Diagram (APD) is constructed from the sequential legacy implementation. The APDs are related to UML Activity Diagrams and contain sequential code blocks and parallel design patterns. APDs are also similar to Petri nets and the already discussed HTGs by Cordes et al. [CMM10]. This first phase exposes the maximum degree of parallelism in form of the APDs. In the second phase, the APDs are aggregated and mapped to provide a optimal degree of parallelism for the target platform while respecting the corresponding trade-offs. Subsequently, the resulting parallel design patterns of the second phase are implemented. The pattern-supported parallelization approach was evaluated on an Unmanned Aerial Vehicle (UAV) case study [JGU13a], an EMS and a construction machinery [JFG+14, JFGU14]. While this pattern-supported parallelization approach is semi-automatic, the interactions with the domain expert is reduced only by the help of well-known parallel design patterns. When using this approach the expert is still the major component when parallelizing an application.

The approach presented by Ceng et al. [CCS+08] is called the MPSoC Application Programming Studio (MAPS). It provides an integrated framework to parallelize legacy applications written in C for multi-core platforms. MAPS assists a developer in developing parallel C applications by combining machine analysis and domain knowledge to suggest partitions to the developer. Starting with the sequential legacy code, the approach provides partitioning suggestions by detecting typical parallelism patterns to the developer. These suggestions are generated by a clustering heuristic on a weighted statement control data flow graph. The generated partitions can be modified, discarded or extended by other partitions, the developer hereby decides the granularity level of the partitions. Based on these partitions and the sequential legacy code, MAPS emits parallelized C code. The approach by Ceng et al. is also commercially available and developed further through Silexica. Their product SLX Parallelizer still relies on interactions with the developer, yet in combination with the SLX mapper the parallelization becomes more and more automated. In contrast to our approach, MAPS suggests complete partitioning solutions for evaluation with the expert while our approach reduces it to one single dependency that has to be evaluated.

3.3 Summary

As elaborated in this chapter, a considerable amount of work has been done to develop approaches that extract task-level parallelism from sequential legacy applications in an automated as well as semi-automated way. However, all of the presented approaches have their limitations. The majority of the published approaches do not consider the conditions in the automotive domain.

Most related approaches assume tight and safe bounds on WCET that are typically not obtainable for automotive applications such as an engine control. To our knowledge, the partitioning of a task with varying execution times and observed WCET has not yet been studied. In contrast to exact WCET bounds, observed WCETs require synchronization at run-time. However, any synchronization may degrade the statically computed speedup at run-time due to runnable execution times exceeding the oWCET. In addition, most approaches assume that there is enough concurrency available inside a task that is worth exploiting. In the automotive domain however, many dependencies and their respective timing are leftovers from the legacy design that drastically limit the concurrency that could be exploited. Whereas there are notable approaches that introduce domain knowledge into the parallelization process, the developers still have to identify the parallelism inside the application to some degree. Hence, a developer has to have a not only good understanding of the functionality of the application, but also on the concurrent architectural design.

To the best of our knowledge, a comprehensive parallelization approach that supports the aforementioned automotive state-of-the-practice and legacy design applications does not exist. With our approach, we provide such a parallelization method for tasks. A comparison of the discussed related work with respect to the objectives of this thesis is presented in Table Table 3.1 on the following page.

Table 3.1: Comparison of approaches that leverage domain knowledge to parallelize automotive legacy tasks with respect to the proposed objectives and solution in this thesis.

Objective	Cordes et al. [CMM10, CM12, CNEM13, CENM13]	Höttger et al. [HKI15]	Faragardi et al. [FLN13, FLSN14b, FLSN14a]	Panić et al. [PKQn+14], Hennig et al. [HvHM+16]	Saidi et al. [SCCM15]	Ceng et al. [CCS+08]	parMERASA [UBG+13, JGU13b, JGU13a, JFG+14, JFGU14]	This work
Considers the automotive ecosystem	—	✓	✓	✓	✓	✓	✓	✓
Considers scalability and efficiency	✓	✓	✓	✓	—	✓	✓	✓
Copes with oWCET	—	—	—	—	—	✓	—	✓
Improves concurrency	—	—	—	—	—	✓	✓	✓
Reduces interactions with experts[1]						—	—	✓

[1] Only applies to approaches that improve the concurrency using domain knowledge

4 Solution Space

As automotive applications are often complex and originate from a concisely standardized development process, the understanding of a task's design is crucial. In particular, the development process for single-core platforms is the reason for a legacy design that brings limitations to the parallelization approach. These limitations define therefore the boundaries of the solution space when parallelizing a task.

This chapter describes the solution space when solving the challenge of parallelizing a task and describes the solution idea of our approach. It is structured in five sections as outlined below. The first two sections describe the automotive development process and its implications toward the legacy design of tasks. The last three sections describe the solution idea of our approach in respect to the legacy design of tasks.

Legacy Design of Tasks The huge amount of communication, implicit requirements and construction principles have formed today's process of designing automotive application. This section shows how an application and its tasks were designed for a single-core platform in retrospective.

Evolving Tasks As legacy applications continue to evolve, their integration on single- and multi-core platforms becomes a challenge. This section elaborates possibilities on how additional functionality can be added to tasks without exceeding the computational power of a single core.

Splitting In this section, we introduce our approach of task splitting to parallelize a legacy task. Task splitting exploits the concurrency inside a task. The concurrency is bound by the data dependencies, their respective timing as well as the ordering of runnables. We show two paths to exploit the task's concurrency for an efficient task splitting.

Relaxation The strict timing of data dependencies is not always physically required. For some dependencies, adjusting its timing would ease the task splitting but could potentially alter the behavior of the application. In this section we introduce our approach to detect these dependencies and modify its corresponding timing while maintaining the functionality of the legacy implementation.

Parallelization Strategies In this section we propose two strategies to parallelize a legacy task. These strategies combine the splitting and relaxation approach and support a typical automotive development process.

4.1 Legacy Design of Tasks

We start by briefly elaborating the process of developing a control application for an automotive Electronic Control Unit (ECU) that controls a dynamic system as introduced in Section 2.1. This lays the foundation to understand the solution space with its conditions and boundaries. It is important to further understand the architecture of legacy tasks found in the automotive domain and its implications toward parallelization.

Developing a controller for a dynamic system is a complex process in the automotive domain. Due to its complexity, the automotive industry uses specific development processes such as the V-model from Section 2.2. In this section we focus on the design phases on the concept and architecture level. We assume that the hardware is given in the form of a single-core platform and all software requirements are specified.

In the V-model, a controller is gradually designed, implemented and integrated. The specification created in the design phase includes the requirements of the control loop, the modeled environment and specified performance goals. This specification is then implemented into a software and integrated into the application that is deployed into the system. In the beginning, the engineer for the controller creates the conceptual design of a controller. The engineer defines requirements, performance goals and the mathematical models of the control loop. Based on this information, the runnables of the controller are defined. A controller typically samples the continuous state via a sensor, processes the data discretely and adjusts actors to interact with the physical and continuous environment in a recurrent way. We illustrate this design process using an example that is consecutively modified throughout this section. We start with a very simple controller that includes three separate runnables in the conceptual design. One for the sampling, one for the processing and one for the actuation. Besides the runnables that encapsulate functionality, the resulting conceptual design also includes the communication between them.

In the next step, this conceptual design is further refined into an architecture design. The goal of this refinement is to provide a complete architecture that can be implemented in the subsequent phase of the V-model. For our approach, the main interest in the architecture design are the tasks. A task is a container for runnables and releases recurring jobs as an abstract scheduling unit. When refining the conceptual design to an architecture design, the runnables have to be mapped to a task as well as the order of execution inside the task has to be determined. The mapping and execution order depends on the execution rate requirements of the runnables and on the communication. When mapping runnables to a task, the maximum sample interval plays a key role as there is the established construction principle to allocate all runnables with the same maximum sample interval to a single task. Mapping a runnable to a task with a shorter sample interval is also possible and can simplify and improve the controller design. However, smaller sample intervals are bounded by the available computing power. Also, with smaller sample intervals, numerical problems may occur due to e. g., missing floating point units in the underlying hardware. For our simple controller example with three runnables that sample, process and actuate, we assume that all three runnables have the same maximum sample interval.

Following these established construction principles, the architecture design for such a controller is shown in Figure 4.1. All three runnables of the controller are executed inside a single task t_1 with the same sample interval τ: First, the runnable r_s samples the continuous state. Based on the resulting value, denoted as label d_s, the runnable r_c computes the value for the actor, typically together with the controller state of the previous job, denoted as d_c. The result in form of the value of label d_a is then given to the actor runnable r_a that adjusts the actor to the continuous environment.

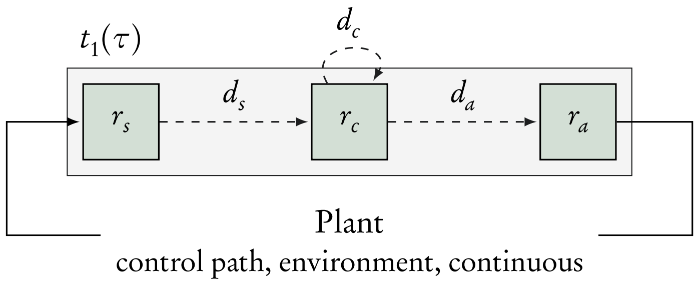


Figure 4.1: Architecture design of an atomic controller with a sampling, a processing and an actuation runnable mapped to a single task t_1.

In our automotive systems, simple controllers such as the one presented are uncommon and are typically more complex. Often, sensor signals have to be filtered due to noise and other interferences before they can be used and processed. The raw signals are typically sampled (runnable r_s) and filtered (runnable r_f) at a smaller interval $\tau/n, n \geq 1$ than the processing and actuating runnable r_c and r_a. Afterwards, the filtered value is communicated to the processing runnable at the corresponding rate, e. g., every n times. This change in the conceptual design has to be reflected by the architecture.

When refining such a controller into its architecture design, there are two different sample intervals, namely the smaller interval τ/n for the sampling and filtering, and τ for the processing and actuation. Figure 4.2 shows the architecture design of this controller. Again following the construction principles, the design for this single controller then contains two tasks that are communicating with each other. Another example for controllers that are divided into tasks running at different sample intervals is when using a solver. Often a solver for differential equations is needed which is specified in the mathematical model of the controller. Hence, in this case the solver is also separated and executed at a much longer interval.

Many controllers in the automotive domain also must learn to adapt to the environment. Thus, in addition to the main control algorithm, there are algorithms that

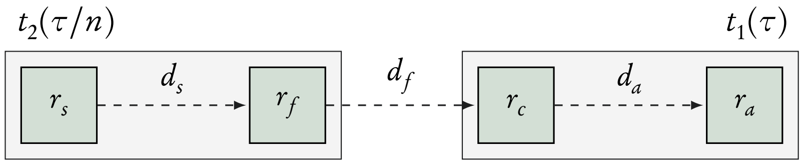


Figure 4.2: Architecture design of a controller with a filtering runnable for the raw sensor signal. The filtering runnable is mapped to a task t_2 with a smaller period than task t_1 that contains the processing and actuating runnables.

approximate changes in the environment. The most straight forward objective to a learning controller is to optimize the quality of the control algorithm. But the control algorithm can also adapt to the deterioration of materials, sensors and actuators. Another typical use case is the error detection that can influence the main control algorithm over a long period of time.

Such an adaptation can usually be executed at a much longer interval $\tau \times n, n \geq 1$ than the main control algorithm. Hence, there is a runnable added to the conceptual design of the filtering controller that is responsible for the adaptation and receives all the needed state and signal information of the system. Figure 4.3 shows the architecture design of this controller which includes a filtered sensor signal and an adaptation. The main controller runnable r_c receives a filtered sensor signal d_f and sends the actuation result d_a to an actuator. The main controller r_c is executed at the sample interval τ, while the sensor signal is sampled in runnable r_s and processed through a filtering runnable r_f every τ/n. The controller sends its state d_c and learns/adapts the parameter d_l from the adaptation r_l every $n \times \tau$. As we can see from Figure 4.3, the design on the architecture level has increased by one runnable that is mapped to one additional task. The extra runnable is responsible for the adaptation of the main control algorithm. Hence, it collects all available state and signal information of the system that adds a huge amount of communication to the design.

Controllers can further be related and connected in very different ways. Controllers can be part of an hierarchical design such that a parent controller provides the reference input for various child controllers. A real-world example for such a hierarchy is the cruise control that is regulating the driving torque controller. A hierarchy of controllers on the architectural level brings additional communications typically from different tasks. In other controller designs, the controllers may be siblings and connected with each other on the same level. There can be controllers that influence the plant through different actuators but are strongly coupled within the physical environment and hence use the same input signals. For example, the front camera delivers the pictures for the Automatic Interval Control System (AICC) to control the distance to the preceding car, as well as delivers the pictures for the lane assist controller. Other controllers may use the same actuators but have a very distinct functionality. For example, the Anti-lock

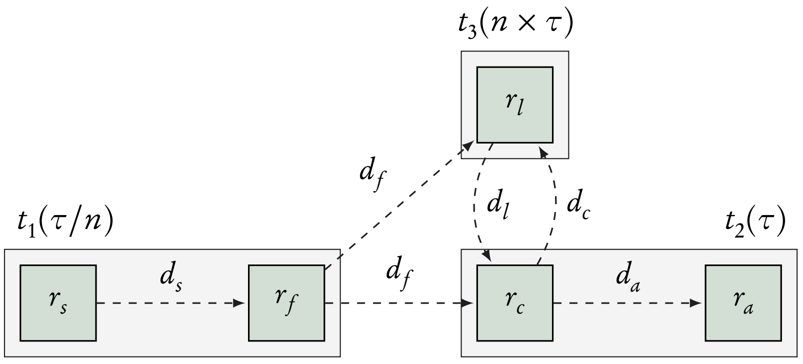


Figure 4.3: Architecture design of a controller with a filtering runnable for the raw sensor signal and an adaptation that maintains a higher level view of the system. Due to the different sample intervals of the runnables, this architecture contains three different tasks.

Braking System (ABS), Traction Control System (TRC), or AICC all use the brakes as actuators. Hence, the controllers have to exchange information, e. g., sensor values, estimates, states, strategies etc. but typically have different sample intervals and are thus assigned to different tasks.

Due to this process of designing controllers with the corresponding construction principles, the resulting control applications are very communicative, highly interdependent, tightly coupled and timing sensitive. Besides this strong functional interdependence, there are additional architectural characteristics to the legacy design of embedded control applications. The communication follows the blackboard principle. Runnables communicate via shared variables with a very low access delay, i. e., a memory access is virtually for free. Also, the order of the runnables inside a task provides a static control of dependencies. And on the task-level, preemptions and priorities enforce implicit assumptions for data consistency requirements or scheduling policies.

4.2 Evolving Tasks

The controller designs are continually evolving as functionality is added while the described construction principles remain unchanged. Every year, there is an 15 % increase of the workload for an Engine Management System (EMS) on average[1]. To illustrate the effect of this evolution, we assume a task t_1 with a set of runnables $\mathbf{R}_{t_1}$ is allocated to a core c_1. The task t_1 is a periodic task with the period P_{t_1} and the relative deadline $D_{t_1} = P_{t_1}$. From simulations and measurements we know that the Observed Worst-Case Execution Time (oWCET) of t_1 is smaller than the deadline D_{t_1} on the assigned

[1]This trend is based on experience at the Robert Bosch GmbH but generally supported [MHAK15].

core. Specifically, the time to execute the workload of task t_1 on the assigned core c_1 at the observed worst-case is $oWCET(t_1) = 0.9 \times P_{t_1}$. Due to the nature of the oWCET, there is typically a safety margin of e. g., 10 % of execution time before the task would violate its deadline. Clearly, when the workload of task t_1 increases by 15 %, which equals only one year of development, the task exceeds the computational power of the assigned core. In this case, the task t_1 would violate its real-time requirements when executed on core c_1.

There are obviously two possibilities to cope with this problem, one is to adjust the hardware, the other one to optimize the software.

From the hardware point of view, the workload of a task would complete earlier and within its real-time requirements when the performance of the task's core is increased. The performance can simply be increased by raising the clock speed, but also optimizations of the core pipeline or architectural changes such as decreasing the access latency to the memory can boost a core's performance. However, major performance gains for multi-core platforms by tuning the architecture of a single core are not to be expected [CSG99]. As explained in Section 2.1.3, the trend for multi-core platforms is to integrate more cores on a processor to boost its overall performance. Certainly, the performance of a single core will also see improvements, but it will not satisfy the high and continuously increasing demand from software.

Optimizing the software is another way to solve the workload problem. However, optimizations on the runnables with the goal to reduce its oWCET is not a valid option for us. Optimizing the runtime of all legacy runnables in a task is a huge effort and benefits are typically nonrecurring or decreasing on each revision. Also, such an approach does not match our objective to provide a scalable and efficient method for real-world automotive applications. Hence, we generally assume that runnables are the smallest entity and cannot be optimized, changed or divided further.

4.3 Splitting

In our approach we focus on the idea of *splitting* tasks to solve the workload problem and hence transforming a single heavy task into multiple light tasks. As mentioned before, a runnable is the smallest entity that has to be executed sequentially and a task specifies the runnables' order of execution. Despite this fixed order, the runnables provide concurrency inside a task and do not necessarily have to run as specified. By detecting this concurrency among the runnables and mapping the runnables to multiple *task partitions*, the task partitions also become concurrent to each other. These concurrent task partitions have the same real-time properties and requirements as the heavy task but can run in parallel. When these n light task partitions are assigned to n different cores on a multi-core processor, there is theoretically n times the workload capacity available than with the single heavy task.

As elaborated before, the main reasons to do task splitting roots in the design of legacy automotive real-time applications, the continuing addition of functionality and the technology change from single-core to multi-core platforms. These roots however limit the concurrency that can be exploited via task splitting. The following coherent limitations when exploiting the concurrency of legacy tasks are order, timing and synchronization.

The order in which runnables are executed inside a task originate from manifold requirements. These requirements have evolved over time and are typically not documented or modeled. When splitting a task, these requirements could be violated. There are three basic requirements that have the potential to be violated:

Precedence relations. Although runnables theoretically neither have an input nor an output, they communicate heavily via shared memory. To illustrate this behavior, let two runnables communicate with each other, one runnable filters data and the other runnable processes this filtered data inside a task. It is safe to assume that the data has to be filtered before it can be processed. Hence, the filtering runnable should precede the processing runnable in the order specified by the task. But such precedence relations are unfortunately often not documented. When splitting a task with undocumented precedence relations, these implicit requirements could be violated.

Race conditions. Similar to the order of execution inside a task, there is also an order of communication. In general, an altered order of communication may result in an incorrect computation. The order of communication is implicitly given by the execution order of the runnables inside a task. Yet, this order may be altered due to task splitting as there is typically no specification to respect while splitting.

Mutual exclusion. This is often used to prevent race conditions. For example, there can be a requirement that two runnables should not be executed in parallel at any point in time. When the two runnables are in two different task partitions, the runnables can potentially be executed simultaneously and violate this requirement.

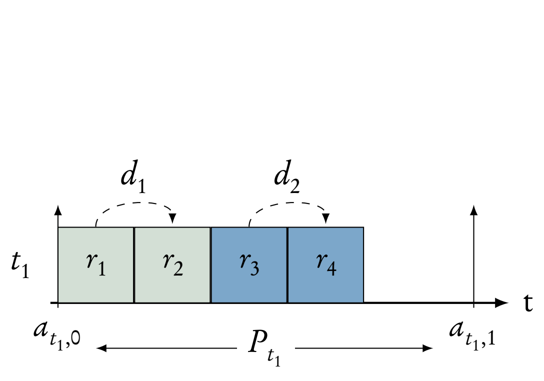
Timing is another limitation when exploiting the concurrency of legacy tasks. Despite that the result of a runnable is correct, it can be time-wise incorrect. Control-based applications are highly sensitive to timing. Any change in the timing of a communication between runnables might change the functionality of the corresponding controllers. Depending on the robustness of the controller's algorithms, a change in timing can violate latency or stability requirements. Such timing requirements can be defined using event-chains but are unfortunately often unknown. However when splitting a task, the timing requirements have to be fulfilled nevertheless.

Synchronization mechanisms to allocate resources to prevent unwanted side effects are another obstacle when detecting concurrency in tasks. Automotive applications use synchronization mechanisms typically against side effects such as e. g., data, mode or other inconsistencies. When a task is split, such synchronization can be problematic. In case of a central locking mechanism it creates an additional overhead. When fine-grained locking is used it is possible that deadlocks are created.

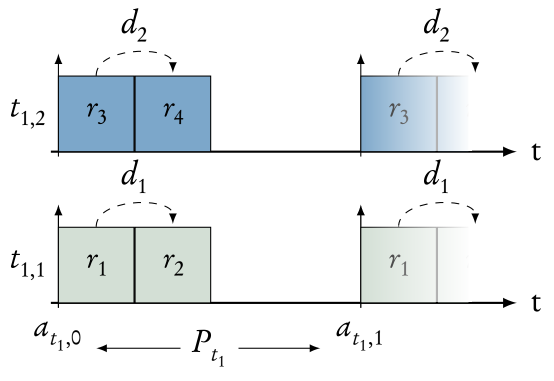
These three limitations, order, timing and synchronization, interact with each other and influence the functionality. To enforce the functionality of the legacy task, the order of execution as well as the communication has to be maintained. Hence, the concurrency of runnables inside a task depends heavily on the communication and order of execution. In our approach, we split a task into multiple task partitions by exploiting the concurrency of runnables inside a task while maintaining the communication and order of execution.

In this thesis we propose two alternative splitting modes, the unsynchronized and the synchronized splitting mode. The *unsynchronized splitting* mode exploits the *available* concurrency of a task. The concurrency is called available because it represents the legacy and natural real-time design of the runnables inside a task. The goal of this mode is to create task partitions that are fully decoupled from each other and strictly follow the Logical Execution Time (LET) semantic of a task. Figure 4.4 shows a simple example where our approach detects concurrency between runnables in a task. The task has four runnables and contains only two communication relationships between r_1 to r_2 via the label d_1 and between r_3 to r_4 via the label d_2. For simplicity reasons, we assume that there are no additional functional requirements for the order of execution. Based on the information on the communication, the order of execution and functional requirements, the approach creates precedence relations. When ensuring these precedence relations while splitting, the functionality of the legacy task is maintained. In this example, the two groups of runnables r_1, r_2 and r_3, r_4 are detected to be concurrent to each other. I. e., the legacy functionality is maintained independent of the distribution of the two groups of runnables as long as the order inside the group is preserved. The two groups of runnables are hence called task partitions and can be executed in parallel. Unsynchronized splitting detects the available concurrency inside a task and transitions a task into concurrent task partitions without changing the functionality. The single objective for unsynchronized splitting is to reach the parallelization goal as close as possible. For our approach, a parallelization goal for a selected task t_i comprises the number of resulting task partitions as well as the speedup. This approach is presented in detail in Section 6.2.

As shown in the previous section, the tasks in our domain are highly communicative that is addressed by the *synchronized splitting* mode. Due to the high amount of communication, there is often not much concurrency to find and to exploit in the unsynchronized splitting mode. Because of the huge amount of communication, the resulting task partitions would have to communicate inter-task-wise. To which extend the task partitions have to communicate depends on the mapping of the runnables to



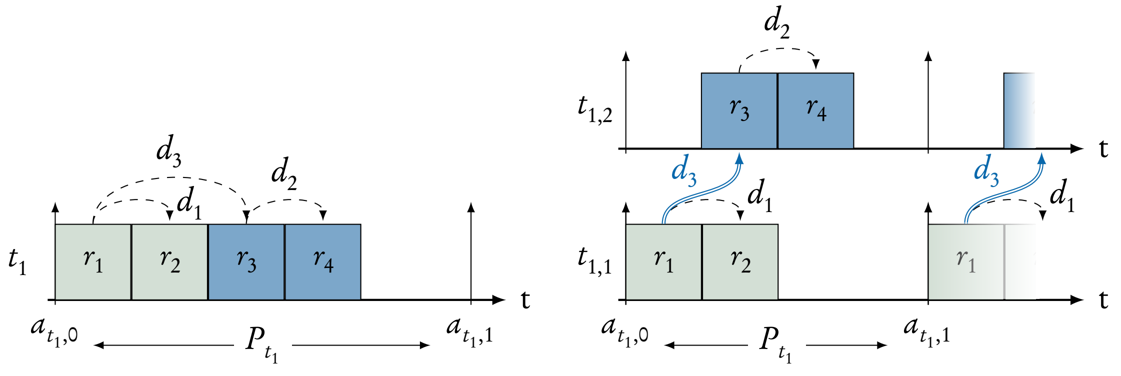
(a) Legacy task t_1.



(b) Splitted into $t_{1,1}$ and $t_{1,2}$.

Figure 4.4: Splitting a legacy task t_1 by detecting concurrency and mapping runnables to multiple task partition $t_{1,1}$ (green) and $t_{1,2}$ (blue) with no intra-task dependencies.

the task partitions. When task partitions communicate with each other, the communication has to be coordinated. This coordination is viable to ensure that the runnables communicate in the right order and at the right time. The functionality of a task is not only depending on the communication and order of execution, but also on the timing of the communication. A change in the timing in either direction may change the functionality. For the synchronized splitting mode, we allow the analysis to create coordination rules for certain communication among the task partitions. This allows to exploit a higher level of available concurrency. Still, when splitting a task into multiple task partitions, this stategy exploits the available concurrency of runnables inside a task and maintains the communication and order of execution. In the following we call this coordination *synchronization*. Figure 4.5 shows an example with a task of four runnables. This time, there is an additional communication between r_1 and r_3 via the label d_3. Assuming that the parallelization goal is two equally balanced task partitions, there is no concurrent solution that does not include inter-task communication. In this case, the analysis of the synchronized splitting mode can find a solution that creates concurrent task partitions with the appropriate synchronization. For our example, the solution is similar to the one from unsynchronized splitting, but with synchronization between r_1 and r_3. This mode adds another objective to the parallelization goal, namely the overhead due to the synchronization. There are two problems with synchronization: (1) The constant overhead of the coordination between the runnables. (2) The synchronization in combination with oWCET is a brittle connection: When a runnable is synchronized with another and takes longer to compute, then the other runnable and all following runnables are also influenced by that. This may result in a situation different from the solution that was determined by the approach. Our approach that can handle synchronization together with oWCET is presented in detail in Section 6.3.



(a) Legacy task t_1.

(b) Splitted into $t_{1,1}$ and $t_{1,2}$.

Figure 4.5: Splitting a legacy task t_1 by detecting concurrency and mapping runnables to multiple task partition $t_{1,1}$ (green) and $t_{1,2}$ (blue) with synchronization between r_1 and r_3.

4.4 Relaxation

Both modes, the synchronized and unsynchronized splitting, exploit the available concurrency of the real-time design in a legacy task. However, the available concurrency may not be sufficient to reach the parallelization goal. Again, this is due to the typically huge amount of communication that constrains the concurrency. The challenge of this section is how to *improve* the concurrency of the task's design while maintaining the functional behavior. The main idea to solve this challenge is to change the timing of a communication. Our thesis is that the physically required communication timing often allows a higher delay than specified by the legacy task. Controllers are typically more robust and can cope with a delayed communication without breaking the functionality. However, the physically required communication timing is often not known or too complex to document.

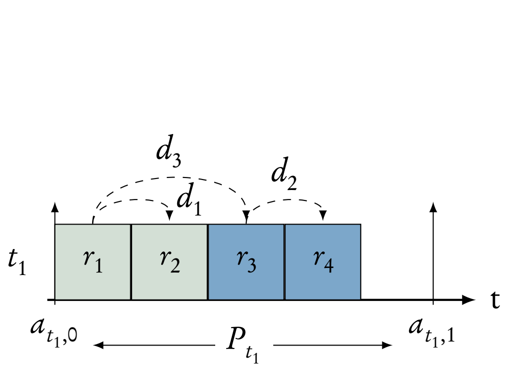
For the purpose of this thesis we define the term *relaxing* a communication as to allow a delay in the transmission. Relaxing the timing of a communication can improve the concurrency of a task, Figure 4.6 illustrates this. The legacy task t_1 again contains four runnables and three communication relationships. Assuming that the parallelization goal is two equally balanced task partitions, there is no splitting solution that does not include synchronized inter-task communication. Fortunately, when a communication is relaxed and thus allows a delay, the concurrency of the task improves. In Figure 4.6, the communication from runnable r_1 to r_3 via label d_3 is allowed to be delayed. Now, runnable r_3 is also allowed to read the value of label d_3 in the subsequent task instance of t_1. This enables us to remove the synchronization from the example in Figure 4.5. Overall, relaxation can improve the concurrency that can be exploited by our splitting approach to create parallel task partitions without any synchronization.

But the main problem still persists, a delayed communication of data potentially alters the functionality of a controller. It can violate latency or stability requirements and other performance criteria. Hence, this change in functionality has to be evaluated. In an ideal world, all latency requirements of the controllers are documented such that potential latency violations can be checked automatically. As stated before and based on our experience, this is not the case for real-world automotive applications. Instead, domain experts have to manually evaluate the change in functionality on the controller design level. There are various techniques to evaluate the impact of an altered communication timing on the functionality of the controller such as formal verification or simulation [ZH15]. But the massive number of communications prohibits the evaluation of all communication by a domain expert. It is simply unfeasible due to time and cost reasons and also contradicts to our objective of an efficient analysis that should be automated as much as possible. Due to this evaluation overhead, only a small number of comunications can be evaluated. To narrow down the search space to find the right communications that are suitable for relaxation and a subsequent evaluation, we have two objectives: (1) Relaxing the communication has the potential to improve the concurrency of the task such that the parallelization goal is met. (2) It is likely that a domain expert would approve a delayed communication without violating any requirements. According to these objectives, our approach proposes the most suitable communication to a domain expert for evaluation. That way, we can increase the concurrency inside a task. The basis for this analysis is the available concurrency and works toward the *inherent* concurrency. The ultimate goal of our approach is to gradually find and iterate toward the inherent concurrency, i. e., the concurrency that is pureley motivated by the physics instead of the legacy design.

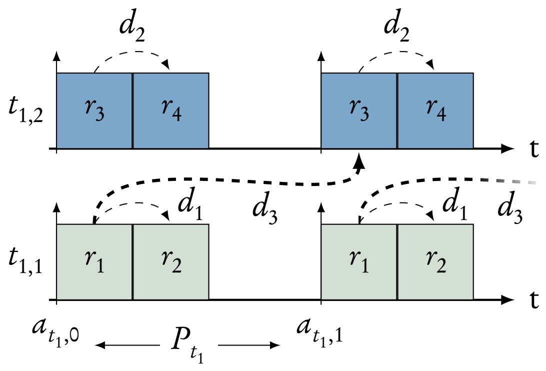
4.5 Parallelization Strategies

In our approach we want to support different stages of the automotive development lifecycle. For this purpose, we created the two parallelization strategies *early* and *late*. In the following we explain the two strategies, their corresponding task splitting approaches and where relaxation can be performed.

In the *early* parallelization strategy, the approach may alter the behavior but should always maintain correctness. This is the case in early development stages, hence the name. In the early stages of the development lifecycle, runnables with their functionality and communications are designed, but the timing is undecided for most communications.



(a) Legacy task t_1.



(b) Parallelized into $t_{1,1}$ and $t_{1,2}$.

Figure 4.6: Splitting of a task t_i by exploiting the relaxed communication from runnable r_1 to r_3 via label d_3.

There is also no implementation of the current design, just a reference architecture from previous lifecycles. Using the early strategy, both splitting modes can be performed on a task. When using the early strategy, our approach is hooked into the design phase on the concept level as shown in the V-model of Figure 4.7. Our approach provides the benefit that architecture decisions regarding the concurrency can be evaluated early in the development lifecycle. At this point, our approach eases the process of evaluating communication timing decisions for domain experts. Challenges such as evaluating the impact of the timing on the task's concurrency when integrating can be solved with little effort. The overall goal of this strategy is to improve the concurrency of a task's design.

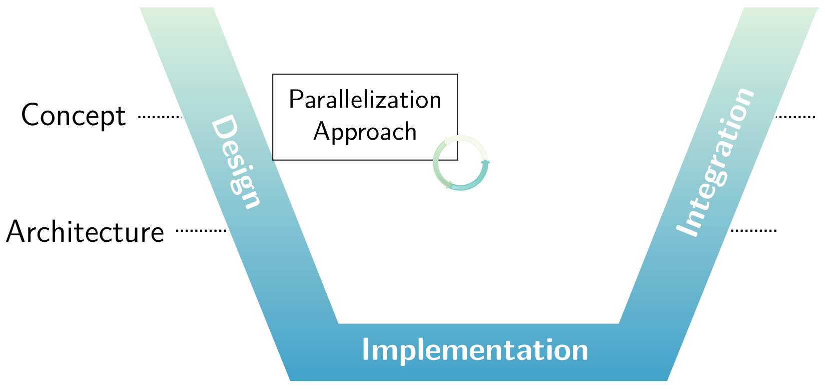


Figure 4.7: Our approach in the product development lifecycle when using the early strategy.

In contrast to the early strategy, there is the *late* parallelization strategy. As the name suggests, this is used in late stages of the development lifecycle, as illustrated in the V-model of Figure 4.8. The typical problem at this stage is that tasks exceed the computational power of a single core due to its increased functionality. At this time, architecture changes are very expensive because the development process is set back to the design phase. Changes of the hardware are similarly connected to high costs. Thus, this strategy should exactly maintain the communications and the respective timing of the legacy implementation. The solution provided by our approach is the splitting of a single task that exceeds the computational power of a single core. In Figure 4.8, our approach is performed in the integration phase on the architecture level. The implementation of the task's design is given e. g., runnables, communications with fully specified timing and the execution order. Again, both splitting modes, unsynchronized and synchronized can be used in this strategy. Results are fed back into the reference architecture for future lifecycles while results from previous lifecycles can improve the splitting. Although it is not mandatory, this strategy may be assisted by an expert if the parallelization goal cannot be reached. In that case, our approach can improve the concurrency by relaxing single yet important communications inside a task. As any change in a communication's timing would result in an expensive evaluation, it is not considered best practice. The overall goal of this strategy is to split a task while exactly maintaining the communication and the respective timing of the legacy implementation. Changes in the timing and thus in the functionality should be prevented whenever possible.

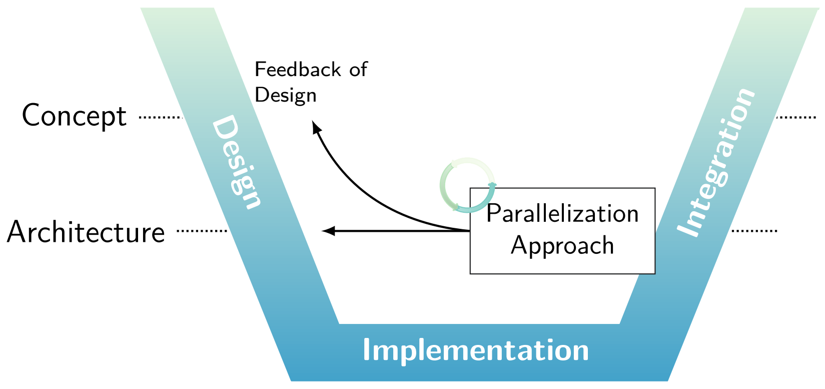


Figure 4.8: Our approach in the product development lifecycle when using the late strategy.

4.6 Summary

In this chapter we have shown that the continuing evolution of the controller design has influenced the architecture of today's tasks. We have elaborated how the huge amount of communication, implicit requirements and construction principles for single-core platforms define the boundaries in the solution space when parallelizing a task. Figure 4.9 illustrates the resulting solution space together with our parallelization approach for tasks. In this figure, the x-axis shows the level of concurrency that can be extracted through task splitting while the y-axis shows the independence of the resulting task partitions. As the concurrency of a task is the foundation for parallelizing a task, we distinguish between the available and the inherent concurrency. The available concurrency is the concurrency that is implicitly given by the legacy design of the task and can be extracted using synchronized and unsynchronized splitting. The selection of the splitting mode depends on the targeted level of independence and available concurrency of the parallel tasks. Synchronized splitting can exploit a higher level of concurrency but at the costs of independence while unsynchronized splitting creates fully independent tasks exploiting less concurrency. In addition, our approach of relaxation can further improve a task's concurrency as it alters the design to reflect the inherent concurrency. The inherent concurrency represents the concurrency that is purely motivated by physics. It is not yet specified in the task's design but can be found by our approach using the assistance of a domain expert.

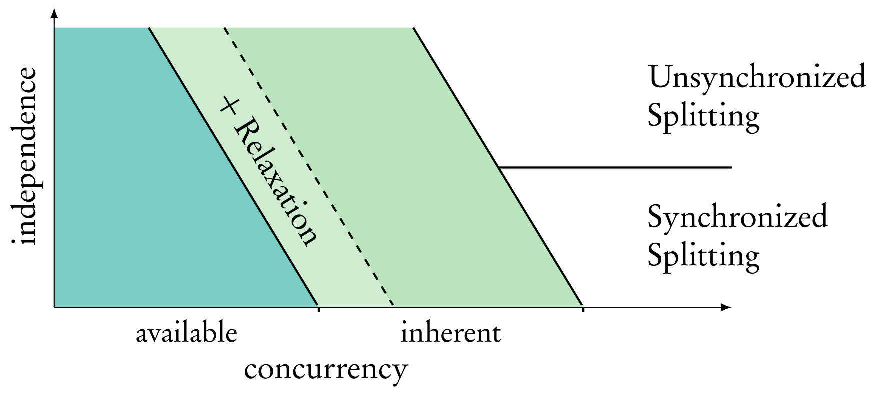


Figure 4.9: The solution space when splitting a task using our approach. The x-axis shows the level of concurrency that can be exploited while the y-axis shows the independence of the resulting task partitions.

5 Parallelization Concept

In this chapter we present a novel parallelization concept for embedded real-time and control-based legacy tasks in detail. Figure 5.1 shows a high-level view of the concept with its four components and the workflow path from start to exit. Three components are automatic components and one is a manual component that interacts with a domain expert. We outline the purpose of each component in the following.

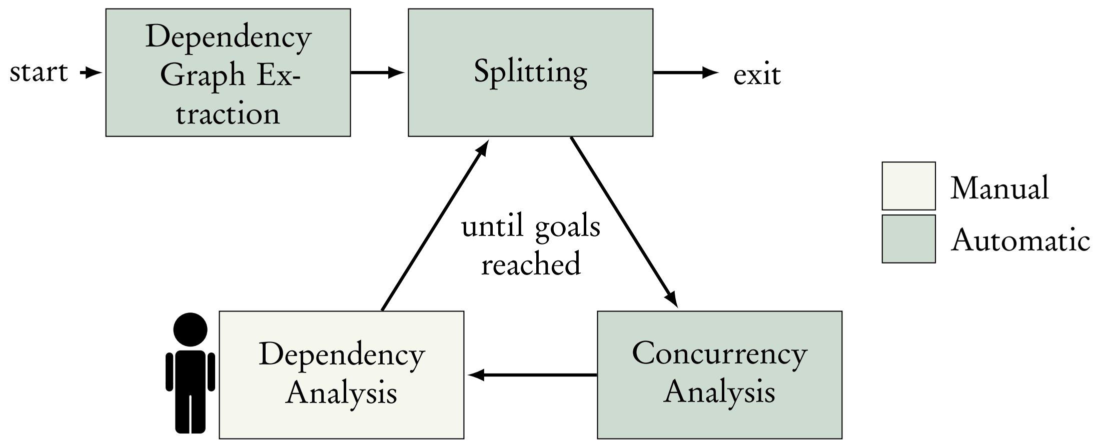


Figure 5.1: High-level view of our parallelization concept with four components and all possible workflow paths.

The *Dependency Graph Extraction* component is a preprocessing component and mandatory for each workflow. This component extracts dependencies from the legacy application model and creates the dependency graph for a legacy task. The dependency graph is the fundamental graph structure of our approach and used by all other components. It abstracts from the fine grained application model with the purpose to specify all needed information that is related to concurrency and parallelism.

The *Splitting* component evaluates the level of parallelism based on the dependency graph by identifying the available concurrency and creates pareto-optimal splitting solutions. It maps runnables of the legacy task to multiple task partitions while considering the overhead due to communication. It also creates all necessary communication mechanisms such that the task partitions can be executed within the Logical Execution

Time (LET) model. The component is able to split complex legacy tasks with a high number of interdependent runnables while maintaining the functional behavior of the legacy implementation. The result of the splitting component is an application model with pareto-optimal solutions of parallel task partitions. The resulting parallel task partitions are freely distributable.

The *Concurrency Analysis* component identifies potential inherent concurrency of a task using the pareto-optimal solutions given by the splitting component. The idea is to find dependencies that can potentially improve the concurrency of the task when they are relaxed. This component is looking for the most suitable dependencies for which the timing can be relaxed without compromising the correctness of the application. Hereby, the dependency search considers the impact toward the concurrency as well as the probability that the timing is allowed to be relaxed. During this process, the inherent concurrency of the physical environment is exposed. As a result, the concurrency analysis component specifies dependency candidates from the dependency graph that have to be evaluated by a domain expert.

The *Dependency Analysis* component leverages domain knowledge of expert(s) to evaluate if the proposed dependencies can be relaxed. The input for this component are the most suitable dependencies found by the concurrency analysis to increase the concurrency of the task. This is the only component that relies on the manual interaction with one or more experts to check if a relaxed timing of the proposed dependencies is allowed.

We propose workflows that define how the components are used in our parallelization concept. Our parallelization concept is flexible such that different workflows can be constructed depending on the parallelization strategy. For the automotive domain we define two essential workflows that are used in the early and late parallelization strategy. The workflow for the late strategy uses the following components in that order: Dependency Graph Extraction and Splitting. The workflow for the early strategy additionally uses the Concurrency Analysis and Dependency Analysis. Together with the Splitting component, these three components create the *expert-in-the-loop workflow*. This workflow determines the dependencies in which the expert and the machine are in an interactive loop until the parallelization goals are met.

Independent of the strategy, the goals that guide the workflow are as follows. The *number of task partitions* together with the *speedup* specify the main parallelization goal. In addition, there is a Boolean goal if the resulting task partitions must contain synchronization. If synchronization between the task partitions is allowed, the associated costs add further requirements to the solution. These costs can be measured using our brittleness metric. The goals are explained further in the chapters that describe the components.

All components of our parallelization concept have been implemented using the model-driven tool-platform AMALTHEA [ITE]. In AMALTHEA, each component is implemented as an AMALTHEA workflow element. AMALTHEA workflow elements are Java classes with an AMALTHEA model as in- and output.

The remainder of the chapter is organized as follows: Section 5.1 describes the integration of the proposed concept into the automotive software development process. Section 5.2 defines graph structures that are used as an intermediate representation of a task's design throughout our parallelization concept. In Section 5.3 we describe the Dependency Graph Extraction component which is the entry point to the concept for each workflow.

5.1 Software Development Process

In this section we elaborate how our parallelization concept is integrated into the automotive software development process. The integration into this process is independent of the used strategy and current position in the development lifecycle. Figure 5.2 illustrates the integration using a bipartite graph with artifacts and processes. At the top and in the beginning, there is the artifact Executable Units Sources & Specification. This artifact contains all related resources for the functionality that is contained in runnables. This contains but is not limited to source code in various languages, object code from other vendors, used libraries, data specification and configuration files. It also includes AUTOSAR related specifications. Besides the functionality, there is the Task Specification artifact. It describes the mapping of runnables to tasks and also the execution order inside a task. This information can be described using specifications from the widely-used automotive standard Model Data Exchange Format (MDX). Both artifacts are input to the Model Generator that creates an application model from the sources and specifications. In our case, this is a proprietary software which generates the AMALTHEA application model. The non-functional AMALTHEA application model is enriched by information from the Execution Time Analysis. It adds the observed execution times of runnables in the best, average and worst-case. This execution time data can have different sources and can e. g., come from simulations, a reference architecture or measurements. The application model is one of the two inputs for our parallelization concept. The second input are the Goals, i. e., the requirements on the result of our method.

Output of our parallelization concept are two artifacts: the Modified Application model and additional Synchronization Code. Based on the modified application model that contains the specification for the parallelized tasks, the Runtime Generator creates and builds an Operating System (OS) Library. The Modified Application model also contains communication timing information that may have changed during the parallelization. Hence, it is input for the final Target Build. The Target Build compiles all sources according to the configurations and specifications provided by the Executable Units Sources & Specification artifact. Note that this artifact is not altered by our parallelization concept. In addition, potentially generated Synchronization Code has to be included in the compilation process of the Target Build. Using all four artifacts, the Target Build can create a binary for the target platform which can be integrated into the Electronic Control Unit (ECU).

5.2 Graph Structures

In this section we formalize the parallelization concept. The concept is relying on graph structures as an intermediate representation of the design of a task. The graph structures contain all information related to concurrency that can be extracted from the application model.

All graph structures depend on a basic set of definitions which we introduce in the following. A single graph always corresponds to a single legacy task $t \in \mathbf{T}$. This task contains of a set of runnables $\mathbf{R}_t$. The containing runnables depend on each other that is specified by the set of dependencies $\mathbf{Q}$. A dependency $q_k \in \mathbf{Q}$ always has a source runnable $src(q_k) = r_a$ and a destination runnable $dst(q_k) = r_b$. As there can be multiple dependencies between two runnables r_a and r_b, this specific set is defined as $\mathbf{Q}^{a,b} \Leftrightarrow \forall q_k \in \mathbf{Q}^{a,b} : src(q_k) = r_a \wedge dst(q_k) = r_b$. A specific dependency between two runnables r_a and r_b is denoted as $q_k^{a,b} \in \mathbf{Q}^{a,b}$.

Figure 5.3 shows the parallelization concept and the graphs used by the components. We define two graph structures for a task: The Dependency Graph and the Precedence Graph. The Dependency Graph constitutes the design of a task including all concurrency-related information. The graph shows how the runnables of a task are communicating or are otherwise dependent on each other. The Precedence Graph is derived from the dependency graph and the basis for task splitting. This kind of graph shows the requirements of runnables that have to be executed before other runnables. Both graph structures can be enriched by the Partitioning Information.

5.2.1 Dependency Graph

The *Dependency Graph* (DG) is a directed cyclic multigraph representation of a task $t \in \mathbf{T}$. It is constructed from the legacy application model using the Dependency Graph Extraction component. A dependency graph for a task $t \in \mathbf{T}$ is denoted as

$$DG(t) := < \mathbf{V}, \mathbf{E}_C, \mathbf{E}_F, w, \gamma, \sigma > \quad .$$

Each node in the set of nodes represents a runnable $\mathbf{V} = \mathbf{R}_t$. Hereby every node has a weight that is represented by the function $w : \mathbf{V} \rightarrow oWCET \in \mathbf{N}$. We mainly use the Observed Worst-Case Execution Time (oWCET) as a weight for a runnable, but best-case and average-case execution times are also possible and supported by our

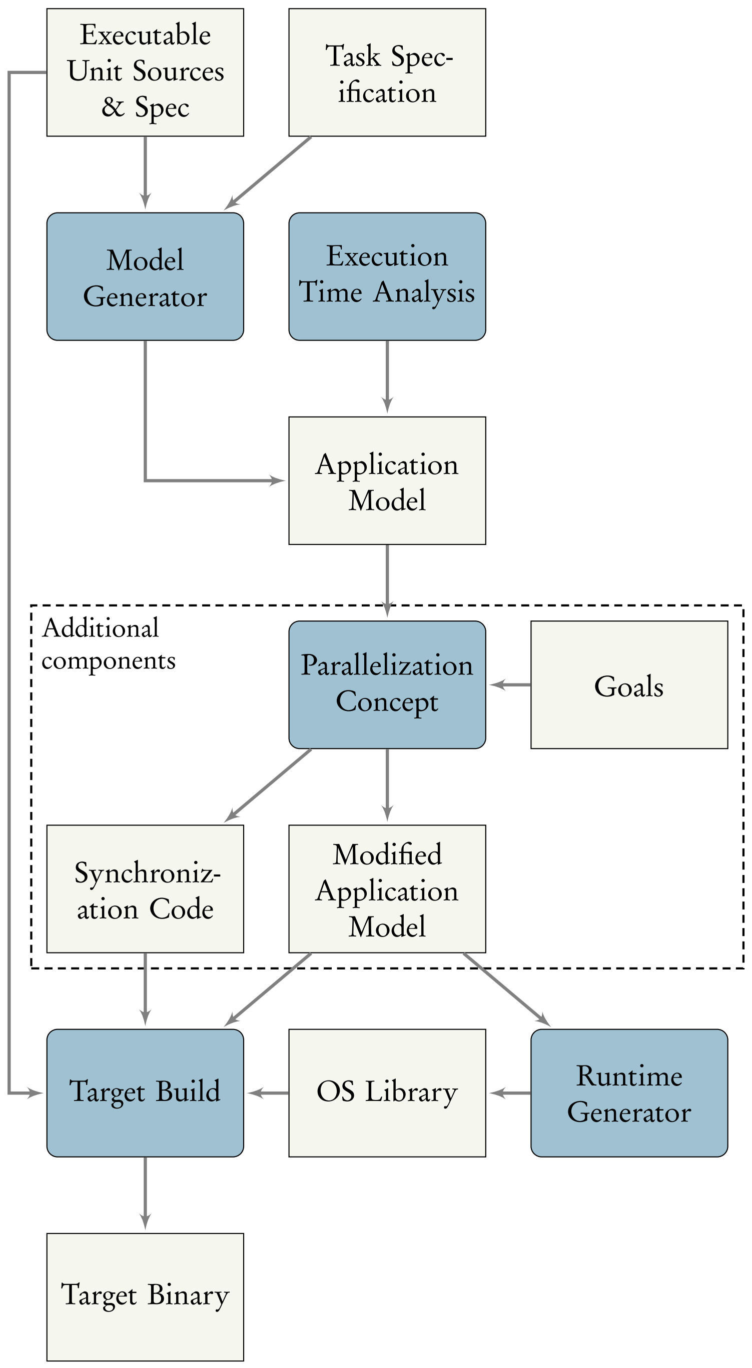


Figure 5.2: Toolchain integration of our parallelization concept

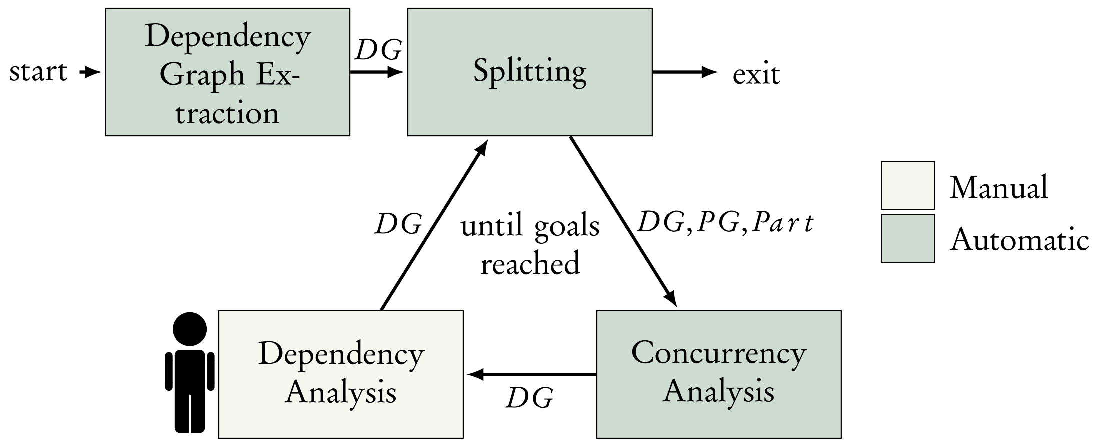


Figure 5.3: High-level view of the parallelization concept with used graphs.

approach. There are two sets of edges that connect the nodes, the set of functional edges $\mathbf{E}_F$ and the multiset of communication edges $\mathbf{E}_C$.

The set of functional edges $\mathbf{E}_F = \mathbf{Q}_F \subseteq (\mathbf{V} \times \mathbf{V})$ contains functional dependencies between two runnables $r_a, r_b \in \mathbf{R}_t$ that is denoted as $q_k \in \mathbf{Q}_F$. A functional dependency is user-specified and defined by an expert of the application. The dependency $q_k \in \mathbf{Q}_F$ indicates the requirement that $src(q_k) = r_a$ runs before $dst(q_k) = r_b$ is executed to ensure a functional requirement e. g., the correct order of interaction with peripherals.

The multiset of communication edges $\mathbf{E}_C = \mathbf{Q}_C \subseteq (\mathbf{V} \times \mathbf{V})$ defines communication dependencies between two runnables $r_a, r_b \in \mathbf{R}_t$ and is denoted as $q_k \in \mathbf{Q}_C$. A communication dependency $q_k \in \mathbf{Q}_C$ indicates the requirement that $src(q_k) = r_a$ sends data to $dst(q_k) = r_b$. All communication dependencies are distinct through the corresponding label $label(q_k) = d_v \in \mathbf{D}$. Besides the corresponding label, a communication dependency has an associated criticality $\gamma : \mathbf{Q}_C \rightarrow \{0, 1, 0..1\}$ that specifies the communication timing in terms of LET. A criticality of 0 requires a strict communication without delay (also called strictly undelayed). Hereby the destination runnable has to consume the data from the source runnable within the same job instance. In contrast, a criticality of 1 requires that the communication must be delayed (also called strictly delayed). The destination runnable has to consume the data from the source runnable from the previous job instance. Furthermore, there is the criticality 0..1 which requires a communication with or without a delay (also called relaxed). Hence, an undelayed and delayed communication timing is allowed.

For all pairs of runnables in the graph, there is an associated evaluation flag/status $\sigma : \langle r_i, r_j \rangle \rightarrow \{t, f, r\}$. It indicates if all dependencies between the two runnables were evaluated (t), not evaluated (f), or requested for evaluation (r) by an expert. This is especially important for the Concurrency Analysis and the subsequent Dependency Analysis.

By the definition of the dependency graph, the graph can contain cycles. A cycle with a path length of one means that a runnable communicates with itself. In this case the communication dependencies between the same runnables have to be strictly delayed: $\forall q_k \in \mathbf{Q}_C^{a,b}, r_a = r_b : \gamma(q_k) = 1$.

The dependency graph DG of a task is created by the Dependency Graph Extraction component. It is used as the in- and output for the Splitting as well as the Concurrency and Dependency Analysis components. For a better understanding we illustrate a dependency graph using the following notation. The nodes $\mathbf{V} = \mathbf{R}_t$ are shown as boxes while the size of the box indicates the nodes weight w. For illustration reasons, many examples have the same node weight. Directed edges $\mathbf{E} = \mathbf{Q}_C \cup \mathbf{Q}_F$ show a dependency between two runnables. A dashed edge is used for a communication dependency $q_k \in \mathbf{Q}_C$ and a solid edge for a functional dependency $q_k \in \mathbf{Q}_F$. The label $label(q_k \in \mathbf{Q}_C)$ as well as the criticality γ is annotated to each communication dependency.

Table 5.1 summarizes the notation of a dependency graph and Figure 5.4 illustrates an example graph.

Table 5.1: Notations of the Dependency Graph

Description	Symbol	Visualization
Node	$v \in \mathbf{V}$	boxes
Functional edge	$e \in \mathbf{E}_F$	directed solid edges
Communication edge	$e \in \mathbf{E}_C$	directed dashed edges
Node weight	$w(v \in \mathbf{V})$	size of node box
Criticality	$\gamma(e \in \mathbf{E}_C$	edge annotation
Label	$label(e \in \mathbf{E}_C)$	edge annotation
Flag	$\sigma(\langle v_i, v_j \rangle \in \mathbf{V} \times \mathbf{V})$	—

5.2.2 Precedence Graph

The *Precedence Graph* (PG) is a directed graph of a task $t \in \mathbf{T}$. It is constructed from the dependency graph $DG(t)$ by the Splitting component. How the precedence graph is generated is covered in Section 6.1. A precedence graph for a task $t \in \mathbf{T}$ is denoted as

$$PG(t) := < \mathbf{V}, \mathbf{E}_P, w, o > \quad .$$

While the nodes follow the same definition as in the dependency graph, the edges have a different meaning. The set of edges specify precedences $\mathbf{E}_P = \mathbf{C} \subseteq (\mathbf{V} \times \mathbf{V})$. A precedence $c_k \in \mathbf{C}$ indicates the requirement that $src(c_k) = r_a$ runs before $dst(c_k) = r_b$ is executed. All precedences have an origin $o : \mathbf{C} \rightarrow \{f, u, d\}$ that denotes the reason why

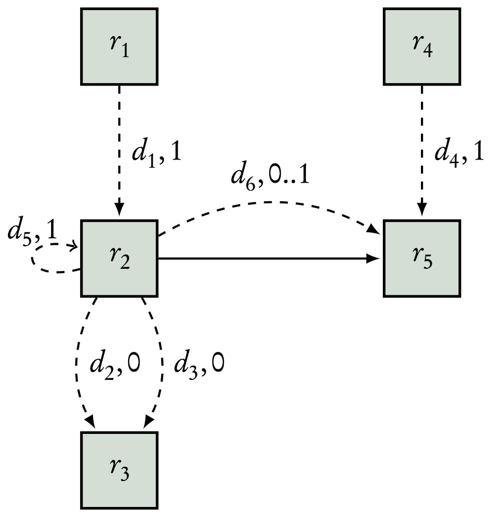


Figure 5.4: Example Dependency Graph with boxes as runnables, dashed edges as communication dependencies and solid edges for functional dependencies.

this precedence exists. It can either be because of functional requirements f, because of strictly undelayed communication u or due to strictly delayed communication d. There can only be one precedence between two runnables. Hence we often notate a precedence from r_a to r_b as $c^{a,b}$. As summarized in Table 5.2 and illustrated in Figure 5.5, the graph notation is similar to the dependency graph. Again, the nodes show the runnables of the task. The solid edges $\mathbf{E}_P$ illustrate the precedences $c_k \in \mathbf{C}$ with its origin o annotated to the edges. The precedences of the graph correspond to the order of the runnables inside a single job instance of a task. When the graph contains cycles, no order of execution can be created. Without an order of execution, the task cannot be implemented. Hence, the precedence graph does not allow cycles.

Table 5.2: Notations of the Precedence Graph

Description	Symbol	Visualization
Node	$v \in \mathbf{V}$	boxes
Precedence edge	$e \in \mathbf{E}_P$	directed solid edges
Node weight	$w(v \in \mathbf{V})$	size of node box
Precedence origin	$o(e \in \mathbf{E}_P)$	—

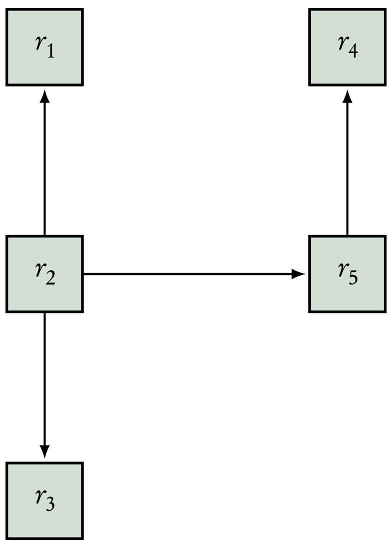


Figure 5.5: Example Precedence Graph with boxes as runnables and solid edges for precedences.

5.2.3 Partitioning Information

The *Partitioning Information* adds properties to a dependency or precedence graph. The partitioning information is created by the Splitting component and represents a solution to the parallelization problem. This information is also used during the Concurrency Analysis to identify potential inherent concurrency. The Partitioning Information *Part* is defined as follows:

$$Part := < m, \mathbf{P}, \mathscr{S} >$$

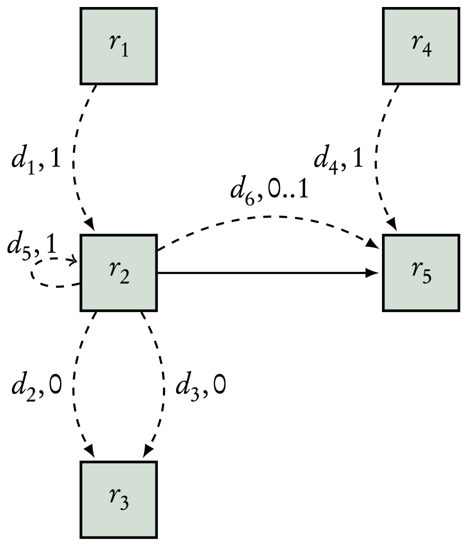
Figure 5.6 shows an example dependency graph with and without the additional properties from the partitioning information.

The first additional property is the mapping function $m : \mathbf{V} \rightarrow \mathbf{T}$ that describes the affiliation of each node $v \in \mathbf{V}$ to a task partition $t_{t,j} \in \mathbf{T}_t$. The mapping function m is shown as boxes around nodes that are labeled with the corresponding task partition.

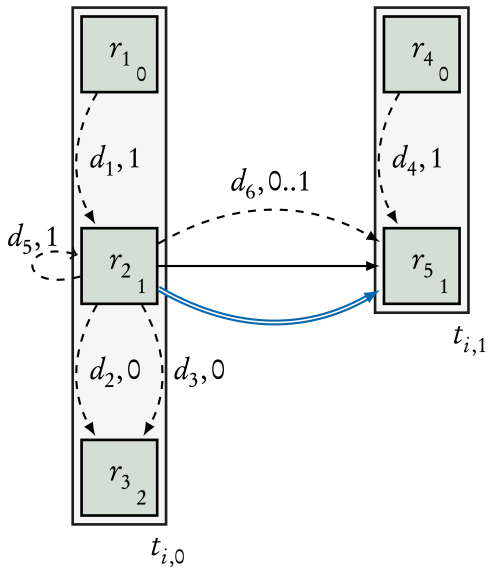
The strict total order $\mathbf{P}(\mathbf{V}, \prec)$ represents the order of execution for all runnables of a task. This totally ordered set $\mathbf{P}$ is represented as indices to the nodes.

There may be the need for synchronization between task partitions which is denoted by $\mathscr{S} \subseteq (\mathbf{V} \times \mathbf{V})$ with a single synchronization $s \in \mathscr{S}$. Similar as with dependencies, the source runnable of a synchronization is denoted as $src(s) = r_a$ and destination runnable as $dst(s) = r_b$. It requires a runnable r_a with a synchronization $s^{a,b}$ to only be activated after r_b has finished execution. As a synchronization is invoked at a specific point in time, it is also called a synchronization point. The synchronization points $\mathscr{S}$ are illustrated as double edges in the graph.

This illustration of the partitioning information can be used as an overlay for the dependency and precedence graph. Hence, this overlay creates a partitioned dependency graph $PDG(t \in \mathbf{T}) := < DG(t), m, \mathbf{P}, \mathscr{S} >$ as shown in Figure 5.6b and a partitioned precedence graph $PPG(t \in \mathbf{T}) := < PG(t), m, \mathbf{P}, \mathscr{S} >$.



(a) Example Dependency Graph



(b) Example Partitioned Dependency Graph

Figure 5.6: Example dependency graph with and without partitioning information.

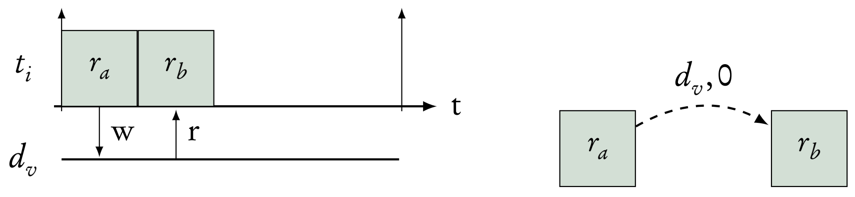
5.3 Dependency Graph Extraction

The Dependency Graph Extraction component is the entry point of every workflow in our approach. Input for this component is the application model and a selected task t. From the legacy application model we know the set of runnables $\mathbf{R}_t$, all corresponding label accesses and the order of execution for the given task. Using this data, the component extracts all information that relates to the concurrency of a task. We call this information the dependencies. There are two kinds of dependencies that relate to the concurrency inside a task, namely the communication and functional dependencies. In the following sections we elaborate the process of extracting both kinds of dependencies.

5.3.1 Communication Dependencies

The huge amount of communication between the runnables of a task is a serious limitation for concurrency. To maintain the same behavior of the legacy implementation, the data has to be send and received by the runnables in the correct order but also from the same job instance. We model these requirements using communication dependencies between runnables of the task. A communication dependency is directed, has a source and destination runnable and a criticality. When a communication dependency q_k is extracted from the legacy application model, its flag is set to $\sigma(q_k) = f$ as no domain expert has evaluated it before. The extraction depends on the label accesses and the order of execution. Hence, there are three different kinds of communication dependencies, the forward, backward and loop communication dependency.

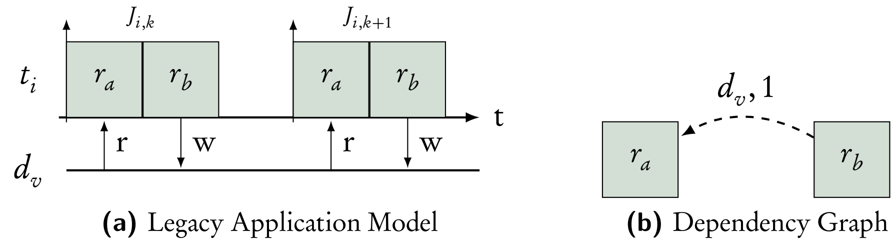
A communication dependency is forward if runnable r_a writes to a label d_v and is executed before runnable r_b reads from it. In other words, r_a produces data which r_b consumes afterwards, as shown in Figure 5.7a. It is not important when r_b consumes the label's value, as long as it is after r_a finished execution within the same job instance. In terms of LET, runnable r_a writes to a copy of label d_v that is located in the local memory of the job instance. The written value is immediately available to all other runnables inside the same job. Hence, runnable r_b that is executed after r_a can read the updated value within the same job instance. The timing of the legacy application model states that the communication has to be strictly undelayed. In this case, the extraction component adds a communication dependency q_k with $label(q_k) = d_v$ and $\gamma(q_k) = 0$ to the dependency graph. The source runnable of communication dependency q_k is r_a and the destination runnable is r_b. Figure 5.7 illustrates the extraction from the legacy application model to the dependency graph.



(a) Legacy Application Model **(b)** Dependency Graph

Figure 5.7: Extraction of a forward communication dependency from two runnables. Runnable r_a writes to label d_v while r_a reads from the label afterwards.

In case runnable r_a writes to a label d_v and is executed after r_b, then it is a backward communication dependency. This dependency has to ensure that r_b reads the value of d_v before r_a overwrites it. This is illustrated in Figure 5.8a. In terms of LET, runnable r_a writes its value to a copy of label d_v in the local memory of the job instance $J_{i,k}$. But runnable r_b was already executed in $J_{i,k}$ such that the runnable has to read the updated value of d_v in the next job instance $J_{i,k+1}$. Hence, all label copies in the local memory of the job update the global labels at the end of $J_{i,k}$. At this point in time, all runnables of the application can read the value written by r_a. Subsequently, runnable r_b reads the updated value in job instance $J_{i,k+1}$. Between the start of job $J_{i,k+1}$ and execution of runnable r_b, runnable r_a is not allowed to be executed as it would overwrite the label's value. The timing of the legacy application model explicitly states that the communication has to be strictly delayed. In this case, the extraction component adds a communication dependency q_k with *label*$(q_k) = d_v$ and $\gamma(q_k) = 1$ to the dependency graph. The source runnable of communication dependency q_k is r_b and the destination runnable is r_a. Figure 5.8 illustrates the extraction from the legacy application model to the dependency graph.

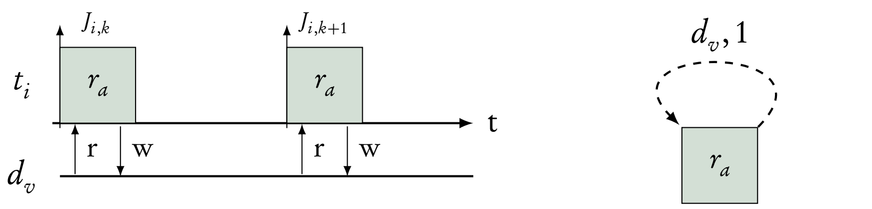


(a) Legacy Application Model **(b)** Dependency Graph

Figure 5.8: Extraction of a backward communication dependency from two runnables. Runnable r_b writes to label d_v in job $J_{i,k}$ which r_a reads from in the subsequent job instance $J_{i,k+1}$.

When a runnable communicates via a label with itself, it is called a loop communication dependency. Hence a runnable r_a performs a write access to label d_v as well as a read access to the same label. This is shown in Figure 5.9a. Obviously, such a communication makes only sense with a strictly delayed timing. Hence, runnable r_a updates the label in job instance $J_{i,k}$ to make use of it in the subsequent job instance $J_{i,k+1}$. This communication behavior is very similar to the one of backward communication dependencies. In this case, the component adds a communication dependency q_k with *label*$(q_k) = d_v$ and $\gamma(q_k) = 1$ to the dependency graph. The source and destination runnable of communication dependency q_k is runnable r_a. Figure 5.9 illustrates the extraction from the legacy application model to the dependency graph.

To extract these communication dependencies $\mathbf{Q}_C$ for the dependency graph, we created Algorithm 1. The input for the algorithm is a single task t from the legacy application model. The algorithm iterates through the call sequence of the task and checks if a combination of runnables (line 2–3) reads and writes to the same label (line 4). If they do, the order of execution is important and decides the direction and criticality of the dependency as explained before.



(a) Legacy Application Model **(b)** Dependency Graph

Figure 5.9: Extraction of a loop communication dependency from a single runnable. Runnable r_a writes from and also reads to label d_v.

Algorithm 1 Extraction of communication dependencies from a legacy task.

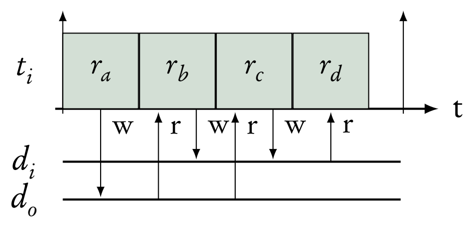
```
 1: procedure EXTRACTCOMMDEPS($t_k \in \mathbf{T}$)
 2:     $\mathbf{Q}_C \leftarrow \varnothing$
 3:     $n \leftarrow numRunnables(t_i)$
 4:     for all $i \leftarrow 1, n$ and $r_i \in t_k$ do
 5:         for all $j \leftarrow i, n$ and $r_j \in t_k$ do
 6:             if $\exists d_v \in \mathbf{D} : r_i$ writes to $d_v$ and $r_j$ reads from $d_v$ then
 7:                 if $r_i \prec r_j$ then
 8:                     add $q_k$ with $src(q_k) = r_i, dst(q_k) = r_j, \gamma(q_k) = 0$ to $\mathbf{Q}_C$
 9:                 else if $r_i \succ r_j$ then
10:                     add $q_k$ with $src(q_k) = r_j, dst(q_k) = r_i, \gamma(q_k) = 1$ to $\mathbf{Q}_C$
11:                 else if $r_j == r_i$ then
12:                     add $q_k$ with $src(q_k) = r_i, dst(q_k) = r_i, \gamma(q_k) = 1$ to $\mathbf{Q}_C$
13:                 end if
14:             end if
15:         end for
16:     end for
17:     return $\mathbf{Q}_C$
18: end procedure
```

5.3.2 Functional Dependencies

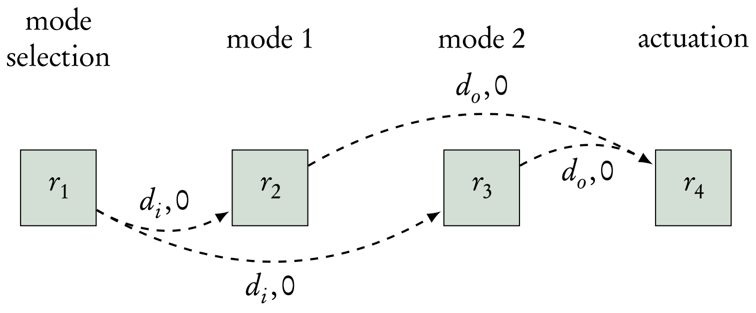
Besides communication dependencies, there are other dependencies that correlate with the concurrency inside a task. We call these dependencies functional. Functional dependencies specify a precedence relationship between a source and a destination runnable. Hereby, the source runnable has to finish execution before the destination runnable is allowed to start its execution due to some functional requirement. It is important to note that such requirements are specified by an expert. These requirements are verified, hence the status flag for these dependencies is set to $\sigma(q_k) = t$ and required for a correct functional behavior. Due to their relation to the functionality, they typically originate from earlier phases (e. g., requirements) of the application design. However, from our experience and case studies based on variations of a single Engine Management System (EMS), the amount of functional dependencies specified that way has been very small. Most functional dependencies that occurred in our case studies stemmed from the error management system of the EMS. Nevertheless, if functional dependencies are specified, they have to be added to the dependency graph as well. In this case, the extraction component adds a functional dependency $q_k \in \mathbf{Q}_F$ for a precedence requirement between r_a and r_b to the dependency graph. The source runnable of this dependency q_k is r_a and the destination runnable is r_b.

A special case where functional dependencies are needed and have to be extracted from the legacy application model is when having multiple writers. A motivational example for this special case is shown in Figure 5.10. The example shows four runnables with two runnables containing functionality for a separate mode. Runnable r_2 for mode 1 and r_3 for mode 2. Both runnables read the selected mode that is set by runnable r_1, process the data according to their mode and write the result back to the same label that is read by the actuation runnable r_4. Here, a functional dependency has to be created to ensure the order of writing from the legacy application model, shown in Figure 5.10a. Without such a dependency, as shown in Figure 5.10c, r_3 could write to d_o before r_2 and may alter the behavior of the system. In the legacy application, r_2 always writes to d_o before r_3 does. This behavior has to be represented in the dependency graph. Hence, we create a functional dependency between r_2 and r_3 to ensure the order of writing. This is called a multi-writer protection and shown in Figure 5.10c.

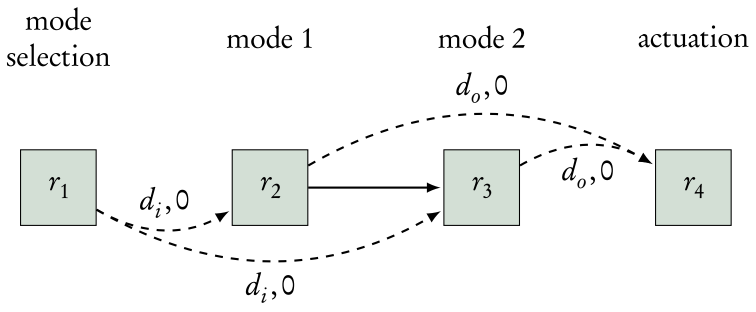
In general, there are multi-writers when two runnables r_a and r_b write to the same label d_v inside a task and another runnable reads from that label d_v that is executed after the two writers. Such a multi-writer communication is usually considered bad practice. The reason is that it is unspecified which writing runnable actually writes to the label during execution. Neither runnable could write to the label, the latter runnable could overwrite the label or the label could be written by only the first runnable that is executed. Thus, it creates some kind of race condition which results in an unspecified design. Multi-writers are therefore typically forbidden by design guidelines. Because it is unknown to us which scenario is the actual preferred scenario, a functional dependency has to be created to preserve the order of writing. The general case to extract these functional dependencies for multi-writers is as follows. Let r_n be the last writer to d_v in a sequence of writing runnables $r_{n-k} \dots r_n$. Then a functional dependency is needed from every writer r_{n-k} that is executed before r_n to r_n. Algorithm 2 shows how functional multi-writer dependencies are extracted from the legacy application model.



(a) Legacy application model.



(b) Dependency Graph without multi-writer protection.



(c) Dependency Graph with multi-writer protection.

Figure 5.10: Extraction of functional dependencies for multi-writers using the example of a mode selection.

5.4 Summary

In this chapter we have presented our novel parallelization concept and its integration into existing automotive development toolchains. We introduced the concept components that create workflows to parallelize a legacy task. Starting point of a workflow is the Dependency Graph Extraction that creates a dependency graph. The dependency graph is the fundamental graph structure that contains all information related to concurrency and abstracts from the application model. Based on this graph, the Splitting generates a precedence graph to evaluate the level of parallelism by exploiting the available concurrency in the task's design. The result of the Splitting is the Partitioning

Algorithm 2 Extraction of multi-writer dependencies from a legacy task.

```
1: procedure ExtractFuncDeps($t_i \in \mathbf{T}$)
2:     $\mathbf{Q}_F \leftarrow \emptyset$
3:     $n \leftarrow numRunnables(t_i)$
4:     for all $i \leftarrow 1, n$ and $r_i \in t_k$ do
5:         for all $j \leftarrow 1, n$ and $r_j \in t_k$ do
6:             if $\exists d_v \in \mathbf{D} : r_i \prec r_j$ and both $r_i$ and $r_j$ write to $d_v$ then
7:                 add $q_k$ with $src(q_k) = r_i, dst(q_k) = r_j$ to $\mathbf{Q}_F$
8:             end if
9:         end for
10:    end for
11:    return $\mathbf{Q}_F$
12: end procedure
```

Information that represents a parallel solution. Our concept can further increase the task's concurrency through the relaxation of suitable dependencies. These suitable dependencies are determined by the Concurrency Analysis and evaluated by a domain expert in the Dependency Analysis. The workflow is iterative and finishes when the parallelization goals for the given legacy task are met.

6 Splitting

Splitting a task is the core component of our parallelization concept. It maps runnables to parallel task partitions based on the concurrency of the task's design. Our splitting component provides two modes, namely unsynchronized and synchronized, to split a task. When splitting unsynchronized, the resulting task partitions are fully decoupled whereas the task partitions are still dependent on each other when splitting synchronized.

Unsynchronized

The resulting task partitions when performing unsynchronized splitting are fully decoupled such that each task partition communicates based on the LET semantic. Each task partition complies with the Logical Execution Time (LET) requirements and is therefore a LET task that can be distributed freely. Figure 6.1 illustrates this. Any communication of a task partition is fully managed by LET. The dependencies from r_1 to r_2 and from r_3 to r_4 require a strictly undelayed communication via the labels d_1 and d_2. Respecting the LET semantic, this data is therefore transfered via the task's local copies of the labels d_1 and d_2. For any dependencies that require stricly delayed communication between runnables of the same task, the data of the corresponding label is transfered at the LET boundaries. At the LET boundaries, the communicated data becomes globally available and can be read by any other runnable independent of the task. In case of a strictly delayed communication dependency, the receiving runnable reads the data in the subsequent job instance. Similar to the strictly delayed intra-task communication, the single dependency from r_2 to r_3 also requires a strictly delayed communication but to a different task. This dependency represents an inter-task communication. The data of label d_3 is also transfered at the LET boundaries, yet the receiving runnable r_3 reads the data in the subsequent job instance inside of task $t_{1,2}$. Hence, both task partitions are executed within their respective LET. Note that the unsynchronized splitting mode does not allow any strictly undelayed communication between the task partitions.

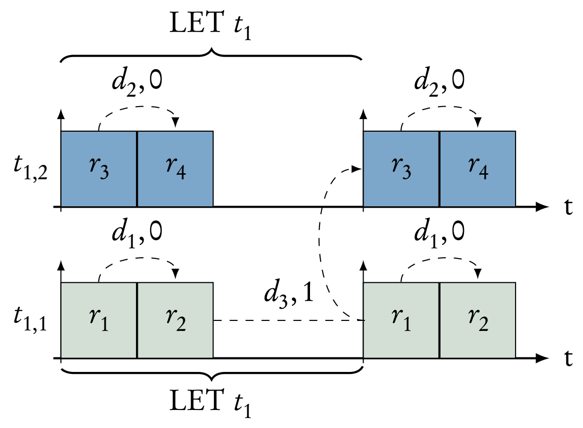


Figure 6.1: Example of resulting task partitions when splitting unsynchronized in terms of LET.

Synchronized

The resulting task partitions for synchronized splitting can be executed in parallel but are still dependent on each other. While all task partitions together form a LET task, the inter-task partition communication does not comply with the LET semantic due to synchronization between the partitions. Figure 6.2 shows the resulting task partitions after splitting in synchronized mode. Any intra-task communication of the task partitions that is either strictly delayed or undelayed is handled by LET. This is similar to unsynchronized splitting. The major difference is that strictly undelayed communication is also allowed between task partitions. Both task partitions hereby share the same local memory of the legacy task (here: t_1). This is not LET compliant and any such communication needs to be synchronized, yet a higher level of concurrency can be exploited in this splitting mode. In the illustrated example, the requirement of the legacy task is that runnable r_2 communicates strictly undelayed with r_4 via d_4. Thus, when mapping the two runnables to different task partitions, this communication has to be synchronized to ensure this requirement. Due to synchronization, the task partitions are dependent on each other to some degree. This is especially important when a preemptive task scheduler is used. Should the task partition that contains the source runnable of a synchronization be preempted, the other task partition has to wait as well.

When specifying the parallelization goals for our concept, one can define which splitting mode should be performed. In general and for both splitting modes, LET ensures that all task partitions communicate consistently and deterministically. Note that only the strictly undelayed and delayed communication dependencies are of interest to this component. Relaxed communication dependencies may or may not be delayed

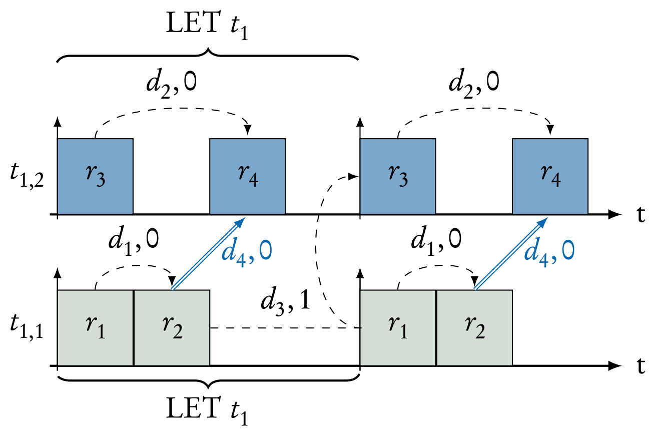


Figure 6.2: Example of resulting task partitions when splitting synchronized in terms of LET.

and thus do not provide any requirement to the splitting. The result produced by the Splitting component is the partitioning information *Part*.

In the following we start by presenting the generation of the precedence graph. The precedence graph is generated from the dependency graph and the basis for any splitting mode. Then we describe the unsynchronized and synchronized splitting modes in detail.

6.1 Precedence Graph Generation

The first step of the splitting component is to create the precedence graph from the dependency graph. The graph transformation $generate : DG \rightarrow PG$ is performed by Algorithm 3 and works as follows: First, all nodes $\mathbf{V}$ and node weights w are copied from the dependency graph to the precedence graph. Then the algorithm iterates over all communication dependencies that is the set of edges $\mathbf{E}_C$ in the dependency graph. It creates precedences for each dependency $q_c \in \mathbf{E}_C$ depending on its criticality $\gamma(q_c)$. If the dependency is strictly undelayed, the order has to be ensured such that the destination runnable always receives the updated value within the same job instance. Hence, a precedence $c^{a,b}$ from the sending runnable r_a to the receiving r_b is created. On the other hand when the dependency is strictly delayed, a precedence has to ensure that the destination runnable r_b receives the updated value from runnable r_a in the following job instance. This is ensured by creating a precedence $c^{b,a}$, i. e., runnable r_b always has to run before r_a. With a relaxed criticality, the communicated data can be delayed. When a delay is allowed then the destination runnable becomes independent of the order of execution inside the task. It can either receive the data from the current job or from the previous job instance. Hence, no precedence is created for a relaxed communication dependency. By the definition of functional dependencies these are precedence requirements and hence are added to the precedence graph as is.

Algorithm 3 Generation of the precedence graph from a dependency graph.

```
1: procedure GENERATE($DG := < \mathbf{V}, \mathbf{E}_C, \mathbf{E}_F, w, \gamma, \sigma >$)
2:     $\mathbf{E}_P \leftarrow \varnothing$
3:     for all $q_c \in \mathbf{E}_C$ do
4:         if $\gamma(q_c) = 0$ then
5:             add $c^{a,b}$ with $r_a = src(q_c), r_b = dst(q_c)$ and $o(c^{a,b}) = u$ to $\mathbf{E}_P$
6:         else if $\gamma(q_c) = 1$ then
7:             add $c^{b,a}$ with $r_a = src(q_c), r_b = dst(q_c)$ and $o(c^{a,b}) = d$ to $\mathbf{E}_P$
8:         else if $\gamma(q_c) = 0..1$ then
9:             no precedence needed
10:        end if
11:    end for
12:    for all $q_f \in \mathbf{E}_F$ do
13:        add $c^{a,b}$ with $r_a = src(q_f), r_b = dst(q_f)$ and $o(c^{a,b}) = f$ to $\mathbf{E}_P$
14:    end for
15:    return $PG := < \mathbf{V}, \mathbf{E}_P, w, o >$
16: end procedure
```

Figure 6.3 shows an example of the precedence graph generation process. There are five communication dependencies and one functional dependency in the given dependency graph of Figure 6.3a. The two dependencies $q^{1,2}$ and $q^{2,4}$ are specified with the timing strictly undelayed. Hence, this order has to be maintained and two precedences with the same direction are added to the precedence graph in Figure 6.3b. The dependency of source and destination r_2 is a strictly delayed loop communication dependency. No precedence is required for this dependency. The same holds for relaxed communication dependencies such as the one between r_1 and r_4. Such a dependency does not create any precedence requirements. The dependency between r_3 and r_4 is strictly delayed. Hence, the order of communication is that r_4 always has to receive the value written by r_3 from the preceding job instance. This requirement is enforced by a precedence in the opposite direction of the dependency. Besides the communication, the functional dependency between r_1 and r_4 is added to the precedence graph as is. One might note that the precedence $c^{1,4}$ is redundant because the order is already specified by $c^{1,2}$ and $c^{2,4}$. The reason is that the precedence relation is transitive. The resulting set of precedences creates a directed acyclic precedence graph, as shown in Figure 6.3b.

There can only be one precedence between two runnables. If a precedence dependency is added that has the opposite direction, the component cannot split the task. Hence, the precedence graph becomes invalid when there are edges in the opposite

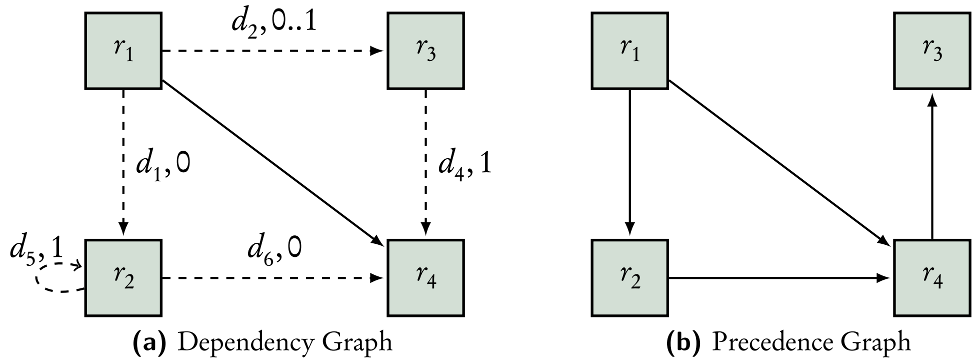


Figure 6.3: Example precedence graph generation. Based on the dependency graph with two strictly undelayed, two strictly delayed, one relaxed communication dependency and one functional dependency, 4 precedences are generated.

direction for each pair of runnables. If there are precedences in the opposite direction, the precedence graph becomes cyclic. A cyclic precedence graph is not schedulable and thus cannot be used for splitting as the component uses scheduling methods.

For the splitting of a task, the precedence graph has to be free of cycles. Precedence graphs that are extracted during late phases of the development lifecycle are always acyclic. When extracting the dependency graph from the application model, the existing order of execution already prevents the generation of cycles. However, cycles may be created during early phases of the development, i. e., when communications between runnables are specified and there isn't an execution order yet. During the precedence graph generation we check for such cycles and provide feedback on how to resolve cycles with reduced effort. Cycles can be resolved in different ways by the domain expert e. g., by relaxation, changing communications or adjusting the execution order.

To detect cycles, we use some properties of the directed graph [MS08]. In a directed graph $G = (V, E)$, a path in graph G is a sequence of nodes to get from an origin node to a destination node by traversing the direction of edges in the graph. A sequence is denoted as a list of nodes e. g., $[r_a, r_b, \ldots]$. A path becomes elementary if no node appears twice in this sequence. A cycle is then a path where the first and last nodes are identical and is elementary if no node but the first and last appears twice. For our purpose, we are interested in the elementary cycles that are distinct to each other, i. e., that are not permutations of each other [Joh75].

In general, a precedence can be part of multiple cycles in a precedence graph. The set of precedences that are part of a cycle is called Feedback Arc Set (FAS). To remove cycles with minimal effort, it is important to find the smallest set of precedences, i. e., the Minimum Feedback Arc Set (MFAS). Given the precedence graph PG, the MFAS is the set with the fewest number of edges, i. e., precedences which, if removed, make the graph acyclic.

In general, finding the MFAS is an NP-hard problem [GJ90]. To solve the MFAS problem, there is a heuristic from Eades et al. [ELS93]. The heuristic is inspired by the selection sort algorithm and time-bounded by the number of nodes (runnables) and edges (precedences) $\mathcal{O}(|\mathbf{V}| + |\mathbf{E}|)$.

Yet, if a precedence constraint is not part of MFAS, it can still be part of a cycle that has to be resolved. To find and resolve these cycles with minimal effort, we create the relation between the number of cycles a precedence is a member of to the total number of cycles in the precedence graph. To enumerate the distinct elementary cycles in a precedence graph we use the algorithm by K. A. Hawick and H. A. James [HJ08]. The algorithm by K. A. Hawick and H. A. James is a variation of Johnson's algorithm [Joh75] that can be applied to multigraphs, recognizes self-loops and can be implemented efficiently. The complexity of this algorithm is time bounded by $\mathcal{O}((|\mathbf{V}| + |\mathbf{E}|)(|C| + 1))$ where C is the number of cycles and space bounded by $\mathcal{O}((|\mathbf{V}| + |\mathbf{E}|))$.

Based on the set of distinct elementary cycles a precedence $c^{a,b}$ from the precedence graph PG is evaluated. Let $\mathbf{V}_{HJ}(PG)$ be the set of cycles of a precedence graph found by the algorithm of K. A. Hawick and H. A. James. Then $|\mathbf{V}_{HJ}(PG)|$ denotes the number of total cycles in the graph. Additionally, the function $\mathit{cycles} : (c^{a,b} \in \mathbf{C}, \mathbf{V}_{HJ}(PG)) \rightarrow \mathbb{N}$ counts the number of cycles the precedence $c^{a,b}$ is a part of. It counts by looking for the sequence of $[r_a, r_b]$ in the set of cycles. This number determines the number of cycles the precedence can resolve if the corresponding dependencies are removed. To rank the precedences according to their potential to remove cycles, we calculate the ratio of the number of cycles the precedence would resolve to the total number of cycles in the graph:

$$\mathit{rank}(c^{a,b}) = \frac{\mathit{cycles}(c^{a,b}, \mathbf{V}_{HJ}(PG))}{|\mathbf{V}_{HJ}(PG)|}, \quad |\mathbf{V}_{HJ}(PG)| \neq 0 \ . \tag{6.1}$$

The rank is zero if the precedence is not part of any cycle. The rank is 1 for precedences that are part of MFAS and thus highly beneficial to resolving cycles. Values inbetween are relative to the number of cycles in the graph.

Figure 6.4 shows an example precedence graph PG with five cycles: $[r_1, r_2]$, $[r_5, r_6]$, $[r_1, r_4, r_2]$, $[r_6, r_3, r_5]$ and $[r_1, r_4, r_5, r_6, r_3, r_2]$. All cycles can be resolved by removing the corresponding dependencies of $c^{2,1}$ and $c^{5,6}$. Hence, in this example the *rank* for these precedences is 1, as they are part of MFAS. The *rank* for all the others are annotated to the precedences.

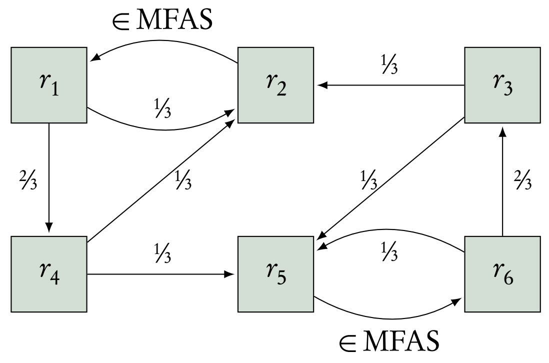


Figure 6.4: Example precedence graph that contains five cycles. Each precedence is annotated with the result of *rank*.

6.2 Unsynchronized Splitting

The goal of the *unsynchronized* splitting mode is to create task partitions that are completely decoupled, independent and unsynchronized. The idea of unsynchronized splitting is to find groups of runnables that can be distributed freely among the tasks partitions according to their precedences. These groups of runnables are part of the available concurrency of a task. However, any dependency limits the available concurrency as the runnables have to be executed in the given order and are thus not concurrent to each other. This is especially true for strictly undelayed dependencies and functional dependencies. Such dependencies requires that the resulting precedence is ensured and that the source and destination runnables are mapped to the same task partition. The requirements for strictly delayed dependencies are fortunately a bit more loose. The source and destination runnables can either be mapped to the same task partition or can be mapped to different task partitions. As there are no precedences generated for relaxed communication dependencies, there are also no requirements to the mapping.

The Splitting component in the unsynchronized mode works as follows. The input for the component is the dependency graph of a given task $DG(t)$ and a parallelization goal g. The parallelization goal g consists of the number of desired task partitions as well as the desired speedup. The number of task partitions to achieve is a fixed number $k > 1, n \in \mathbf{N}$ that is often set based on requirements from other parts within the development lifecycle. The speedup measures the achieved parallelism of the splitted task partitions in relation to the single legacy task. The speedup does not come naturally in mind when considering real-time systems, because processing the same problem faster, does not result in a better system. Real-time systems generally require the completion of computations in the correct order before deadline. However, for our approach, the speedup is an important indicator of how much additional functionality can be integrated after the task is splitted. Splitting a task into multiple task partitions frees up capacity for additional dependent functionalities within the task's LET in the corresponding period. The speedup is defined as the fraction of the makespan of the non-splitted legacy task $makespan_{SC}$ to the makespan of all splitted task partitions $makespan_{MC}$. The makespan is the overall completion time of a task or task partition. Thus, the speedup is defined as

$$speedup = \frac{makespan_{SC}}{makespan_{MC}}$$

where $makespan_{SC}$ is the sum of the Observed Worst-Case Execution Time (oWCET) of all runnables inside the legacy task. The $makespan_{MC}$ is the overall completion time, i. e., the finish time of the last runnable in any task partition.

Using this input, the steps to split unsynchronized are:

1. Generate the precedence graph from $DG(t)$.
2. Search for connected components.
3. Merge connected components to create task partitions.

The first step of generating the precedence graph was already discussed in Section 6.1.

Based on the resulting precedence graph we search for connected components in the second step. Connected components represent the groups of runnables that can be distributed freely among the task partitions. These components of runnables only have undelayed or functional precedences inside the component and only delayed or no precedences to other components. To find connected components we utilize an algorithm from the well-known Java Universal Network/Graph Framework (JUNG) [OFW+05]. The algorithm iterates over all nodes of the precedence graph and performs a breadth-first search for each unvisited node to find connected nodes in the corresponding neighborhood. It hereby ignores all delayed precedences. The runtime of the algorithm for our precedence graphs is upper bounded by $\mathcal{O}(|\mathbf{V}| + |\mathbf{E}'_P|), \mathbf{E}'_P = \forall c_k \in \mathbf{E}_P : o(c_k) \neq d$.

Figure 6.5 illustrates an example precedence graph of a task t_1 with five runnables and three precedences. The precedence edges are annotated with the origin o. Using our algorithm, there are three connected components found in this precedence graph. The first component containing runnable r_1 and r_2 are connected by an undelayed precedence but have no precedence relation to other components. The second component with runnable r_3 and r_4 are also connected by an undelayed precedence but r_4 also has a delayed precedence relation to r_5. Runnable r_5 forms the third component and has a delayed precedence relation to the second component.

In the last step, we merge the resulting connected components to find a solution for the given parallelization goal g. For this purpose, an adapted first-fit decreasing bin-packing algorithm is used. The algorithm packs the task partition bins $t_{i,1}, \ldots, t_{i,k}$ provided by the number of task partitions k in g with the connected components $m_1, \ldots, m_l$ from the graph search. The sum over all runnables' weights of a component

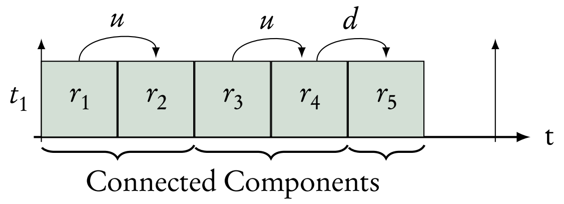


Figure 6.5: Weakly connected components of a task t_1 with 5 runnables and 3 communication dependencies.

m_l is used as the size w_l for the packing. The first fit decreasing algorithm is a greedy strategy that packs the components one after another in decreasing order of their size into the first task partition bin they fit in. The runtime for this algorithm is upper bounded by $\mathcal{O}(l \log l)$, with l being the number of connected components [JDU+74]. The components can be distributed among the task partition bins freely and also be rearranged inside the bins as long as the order inside the component is maintained. This order is specified by the precedence graph. However, when a component with a delayed precedence to another component is packed into the same task partition bin, the precedence has to be ensured. These precedence checks and the potential reordering inside a task partition bin is performed after the bin-packing.

Using the example from Figure 6.5, the task partitions are created as follows. We assume that the number of task partitions in the parallelization goal g is set to $k = 2$ and all runnables have the same weight. Hence, the three connected components are packed into two task partition bin. Bin $t_{i,1}$ gets the first component with runnables r_1 and r_2. The second bin $t_{i,2}$ gets the second component containing runnables r_3 and r_4. The last component with runnable r_5 is then packed to bin $t_{i,1}$. Here, no reordering inside the bins has to be done, as there are no delayed precedences among the components inside a bin. If the last component would have been packed to bin $t_{i,1}$, the reordering had to ensure that r_5 is always executed after the component with r_1 and r_2. In this example, the resulting speedup is 1.67.

Based on the resulting task partitions bins and the precedence graph, the partitioning information *Part* can be created. The mapping m of runnables to a task partition is derived from the contents of the bins. The order of execution $\mathbf{P}$ is adapted from the precedence graph for each connected component and the order of the components inside each bin. As the connected components are concurrent to each other, the order of the components in $\mathbf{P}$ can be arbitrary. And finally, the set of synchronization points $\mathcal{S}$ is empty because the task partitions are fully independent.

6.3 Synchronized Splitting

If the parallelization goal cannot be reached by the unsynchronized splitting our requirements to create fully independent task partitions has to be loosened up. The available concurrency given by the task's design is simply not enough to create fully independent task partitions and simultaneously meet the parallelization goal. In contrast to unsynchronized splitting, we allow strictly undelayed communication between the resulting task partitions in the synchronized splitting mode. By loosening the requirements for independent task partitions we can increase the solution space to find concurrent task partitions. Yet, allowing strictly undelayed communication raises the need for synchronization. Synchronization is needed to ensure the order of execution and communication among the parallel task partitions. With the splitting component using the mode *synchronized*, we can achieve a higher degree of parallelization with as little synchronization as possible. Similar to unsynchronized splitting, the goal is to create multiple task partitions from a single legacy task that are as independent as possible.

In this section, we propose a process to split complex legacy tasks with a high number of interdependent runnables. The main advantage of this novel task splitting process over similar approaches is that it can handle runnables with widely varying execution times and oWCETs. Hence, it can deal with the current state of practice in the automotive where software is typically not designed to allow tight WCET bounds and sporadic misses of task deadlines are often tolerable [ZH15].

Input for the splitting component is the dependency graph of a given task $DG(t)$ and a parallelization goal g. Using this input, synchronized splitting follows three steps which we discuss in the next subsections in detail:

1. Generate the precedence graph from $DG(t)$.
2. Split the task into multiple task partitions respecting the precedences, according to the parallelization goal and based on a static makespan-centric allocation heuristics (see Section 6.3.2).
3. Generate necessary synchronization points to ensure precedence constraints between task partitions (see Section 6.3.3).

6.3.1 Objectives

Similar to the unsynchronized mode, the parallelization goal that contains the objectives is input to the Splitting component. It contains the number of desired task partitions and the speedup that should be reached (discussed in Section 6.2). The number of task partitions is defined as $n > 1, n \in \mathbf{N}$. The speedup is the fraction of the makespan of the legacy task to the makespan of all splitted task partitions. It is important to note that the makespan includes possible idle or waiting times that may appear due to synchronization between the task partitions and is especially important in this splitting mode.

As mentioned before, introducing synchronization between task partitions comes at a cost. We therefore use the *brittleness* as an additional metric for synchronized splitting to compare solutions against the parallelization goal. The actual cost that comes with each synchronization point is that the statically calculated speedup may degrade when the task partitions are executed in parallel. These potential costs of a single synchronization point are measured by the brittleness while the *overall brittleness* measures the costs of a whole splitting solution.

The brittleness is based on the *gradient* metric. To motivate the gradient metric we illustrate in the following how and why the speedup can be degraded. Figure 6.6 shows a precedence graph example where a runnable r_a has to precede runnable r_b. This is specified by the precedence $c^{a,b}$. Due to other precedences not illustrated here for brevity reasons, the two runnables are mapped to different task partitions, r_a to $t_{1,0}$ and r_b to $t_{1,1}$. In such a case of inter-task partition precedences and thus communication, synchronization has to be introduced to enforce this behavior. Hence, the precedence $c^{a,b}$ has to be backed by a synchronization point $s^{a,b}$. Until this point, the task partitions and the synchronization is statically determined. In contrast, the top half of Figure 6.6 shows the execution of all runnables at their respective oWCET or better. As one can see, the calculated speedup of 2.0 for the two task partitions holds as long as the runnables execute at oWCET or better. Even the synchronization does not add any additional waiting time in this scenario. Note that we neglect the constant overhead that typically comes with synchronization in this example. On the other hand, the bottom half of Figure 6.6 shows a *oWCET overrun* of a runnable in the task partition $t_{1,0}$. Again, Observed Worst-Case Execution Times are the state of practice in the automotive domain. As there are no safe and exact WCET available, it is possible that runnables overrun their measured and observed WCET. In such a case, the synchronization $s^{a,b}$ takes action such that r_b has to wait for its execution until r_a finished. At the synchronization point $s^{a,b}$, the additional time that the destination task partition $t_{1,1}$ has to wait is directly added to the overall makespan. The additional waiting time is called a *makespan penalty* δ of the synchronization point $s^{a,b}$. In this scenario, the runnable before r_a overruns its oWCET by 2.5 times and the statically calculated speedup of 2.0 degrades to 1.78.

But there are also synchronization points that are, to some degree, more robust against oWCET overruns. For example, Figure 6.7 shows in the bottom half that despite the overrun, there is no makespan penalty. Still, when the first runnable in task partition $t_{1,0}$ overruns its oWCET and takes twice the time for its execution, the synchronization point $s^{a,b}$ does not lead to additional waiting times. The makespan of $t_{1,0}$ increases but the resulting overall makespan for both task partitions is the same as in both cases. Hence, also the statically calculated speedup of 2.0 is maintained.

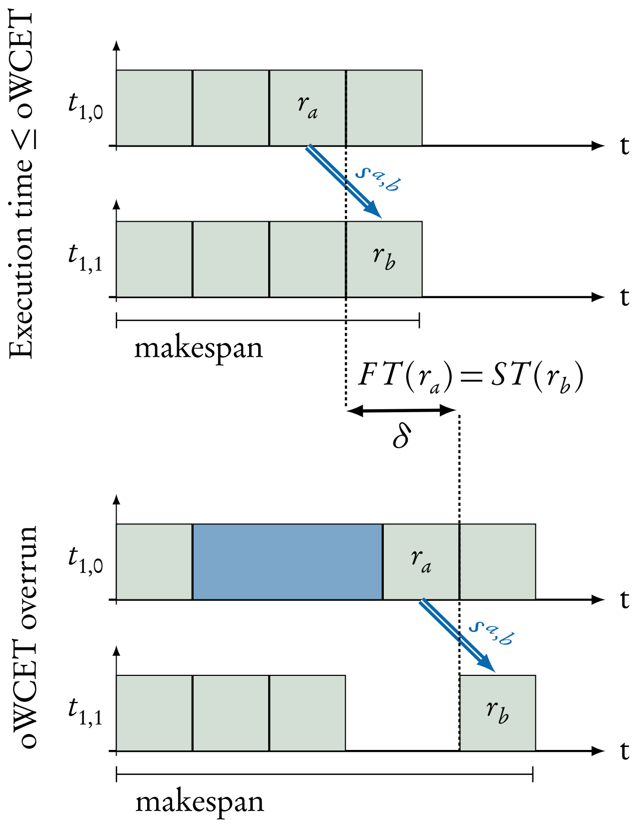


Figure 6.6: Scenario where the synchronization point $s^{a,b}$ with a gradient of zero immediately results in a makespan penalty δ in case of an oWCET overrun.

The difference between the examples in Figure 6.6 and Figure 6.7 is the *gradient* of a synchronization point. The gradient expresses an important factor to determine how likely it is that a synchronization point will add extra waiting time to the makespan and thus degrade the speedup. The gradient of a synchronization point is the time the runnables of a source task partition can overrun its oWCET without creating a makespan penalty. The gradient is calculated as

$$grad(s^{a,b}) = ST(dst(s^{a,b})) - FT(src(s^{a,b}))$$

where ST is the start time of the destination runnable of $s^{a,b}$ and FT is the finish time of the source runnable of $s^{a,b}$. As depicted in Figure 6.6, the gradient of synchronization point $s^{a,b}$ is zero such that any oWCET overrun results in a makespan penalty. On the other hand, the gradient of synchronization point $s^{a,b}$ in Figure 6.7 is greater than zero, i. e., the runnables of task partition $t_{1,0}$ can overrun the oWCET by up to the gradient's value without creating a makespan penalty. It is important to note that any oWCET overrun degrades the performance of a task, irrespective of being executed sequentially

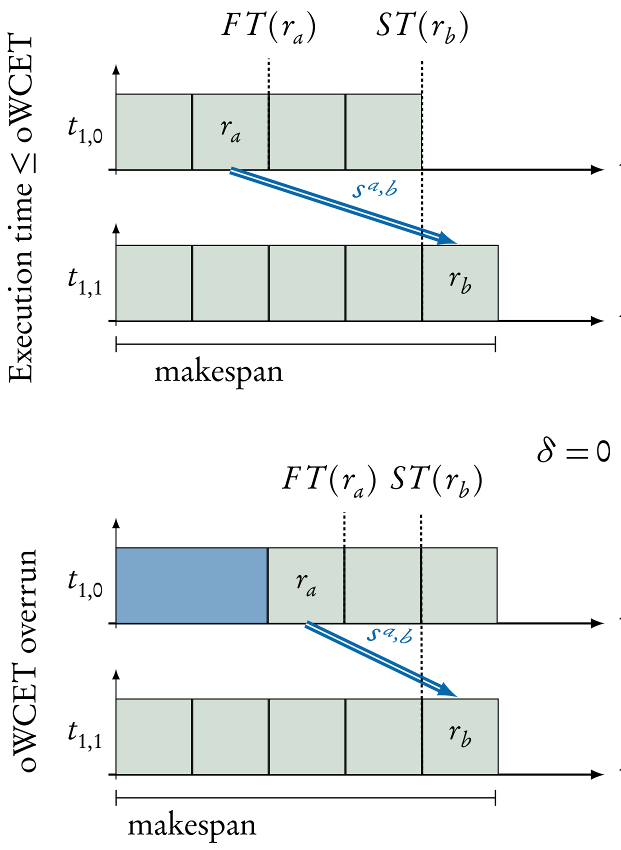


Figure 6.7: Scenario where the synchronization point $s^{a,b}$ with a gradient of > 0 creates no makespan penalty δ in case of an oWCET overrun.

or in parallel. However, when parallelizing a task using synchronized splitting, the introduced synchronization could potentially degrade the performance even further.

As the gradient is an absolute metric, it has always to be seen in relation to the task partitions. Therefore, the gradient is used to compute the *brittleness*. The brittleness is the normalization of the gradient in relation to the makespan of the task partitions and represents the costs of a synchronization point. By reducing the brittleness one can increase the robustness of the splitted tasks toward oWCET overruns. The brittleness of a synchronization point $s^{a,b}$ is defined as

$$brittleness(s^{a,b}) = \left(\frac{grad(s^{a,b})}{makespan_{MC}} - 1\right)^k \tag{6.2}$$

while $k \geq 1 \in \mathbb{N}$. The brittleness function with e. g., $k = 4$ is plotted in Figure 6.8. The y-axis measures the brittleness of a synchronization point $s^{a,b}$ while the x-axis represents the gradient in ratio to a fixed makespan. The brittleness is zero when the gradient of $s^{a,b}$ equals the makespan. This is the best possible case when synchronizing. The oWCET overrun of a single runnable would have to be greater than the overall makespan to result in a makespan penalty. However, a gradient that equals the makespan is only possible when both runnables have an oWCET of zero and are executed at the beginning and end of the respective task partition. A brittleness of zero is therefore only a theoretical value. The brittleness of a synchronization point in a splitting solution increases depending on k while the gradient decreases. In case the gradient is zero, the brittleness is defined to be at 1. This is the worst case for a synchronization point as any oWCET overrun directly results in a makespan penalty. Depending on the system and domain, the synchronization costs might not increase linearly with the gradient. Hence, the exponent k is used to adjust this correlation. Based on our experience with synchronization in an EMS, we will use $k = 4$ throughout this thesis.

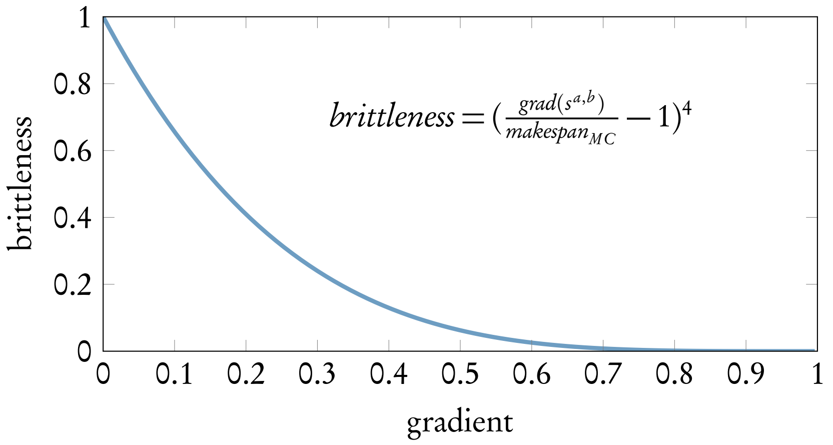


Figure 6.8: Trend of the brittleness in relation to the gradient. The y-axis measures the brittleness of a synchronization point $s^{a,b}$, the x-axis measures the gradient in ratio to a fixed makespan.

To compare different splitting solutions with each other and the parallelization goal, we introduce the *overall brittleness*. The lower the overall brittleness the more robust is the splitting solution against oWCET overruns. It is defined as the sum over the brittleness of all synchronization points $\mathcal{S}$ in a given splitting solution:

$$brittleness(\mathcal{S}) = \sum_{s_k \in \mathcal{S}} brittleness(s_k) \tag{6.3}$$

In future work we would also like to include the probabilities that a runnable exceeds its worst-case execution. The brittleness metric would benefit from current research on Probabilistic Worst-Case Execution Times as it becomes more accurate in determining when overruns might happen.

6.3.2 Mapping

In this section we explain how a task is split by mapping the runnables of a single legacy task to multiple parallel task partitions. The splitting is based on the precedence graph extracted in Section 5.3 and uses an algorithm based on list scheduling to reach the parallelization goal. The algorithm has two major phases, the *prioritizing phase* and the *selection phase*.

Starting with the *prioritizing phase*, our algorithm assigns a priority to each runnable. These priorities create a topological order that ensures all precedences of the PG. We use Highest Level First with Estimated Times (HLFET) [Hu61] for that purpose which is a widely-used static makespan-centric list scheduling heuristic. Hereby, the HLFET algorithm uses the longest path from a runnable r_i to the exit runnable that is called $b-level$ as shown in equation 6.4. In our case, the weight w_i is the observed Worst-Case Execution Time (WCET) of r_i. The $b-level$ is computed recursively by traversing the graph starting from the exit runnable with $b-level(r_{exit}) = w_{exit}$. The resulting list that is sorted by its $b-level$ ensures all precedences of the PG.

$$b-level(r_i) = w_i + \max_{r_j \in succ(r_i)} \{b-level(r_j)\} \tag{6.4}$$

In the following *selection phase*, a task partition for each runnable is selected to which the runnable is mapped. Hence, the first runnable of the ordered list produced by HLFET is chosen and mapped to a task partition based on a mapping heuristic. In the following we present three simple yet powerful heuristics to map runnables that can be combined effectively.

We distinguish between dependent and independent runnables. A runnable $r_i \in \mathbf{R}$ is dependent if it produces (or consumes) data that is consumed (or produced) via a dependency $q^{i,j}$ by another runnable $r_j \neq r_i \in \mathbf{R}$. Hence, as soon as a runnable has to precede or is preceded by another runnable, it is a dependent runnable. A runnable $r_i \in \mathbf{R}$ is independent if for all runnables $r_j \in \mathbf{R}$ there is no dependency between r_i and r_j. In terms of the precedence graph, if there is no precedence connecting the runnable to another runnable, it is an independent runnable. Note that these runnables are only independent in the context of their respective task. They can certainly consume or produce data for runnables outside of this task or for themselves. However, these dependencies are always of the kind strictly delayed and thus do not have to be considered.

The presented heuristics can map both dependent and independent runnables. Separating the independent runnables from the dependent ones can improve the solution because the mapping of independent runnables can be optimized further.

Dependent Runnables

In the following we present three greedy heuristics to map runnables to task partitions as part of the selection phase. Each heuristic implements the function *select* : $\mathbf{R}, \mathbf{T} \rightarrow \mathbf{T}$ that selects a task partition for the given runnable from the set of provided task partitions. All three heuristics are optimized toward one of the presented objectives and can be combined.

The first mapping heuristic is *earliest possible execution*. Figure 6.9 illustrates the idea of the heuristic. The selected runnable is hereby mapped to the task partition with the smallest makespan at the time. The task partition is selected by

$$select(r_k, \mathbf{T}_i) = \min_{\forall t_{i,j} \in \mathbf{T}_i} (makespan(t_{i,j})) \tag{6.5}$$

while *makespan* calculates the oWCET sum over all runnables of a task partition. From all task partitions, the one with the minimum makespan is returned by min. When there is a tie, the smallest index of the task partition decides. In the given example, runnable r_6 is taken from the priority list and should be mapped to a task partition. The heuristic chooses to map the runnable to task partition $t_{i,1}$ because its makespan is smaller than $t_{i,0}$. The earliest possible execution mapping heuristic is optimized for the speedup. This heuristic is also used as a tie-breaker for the following more specialized heuristics.

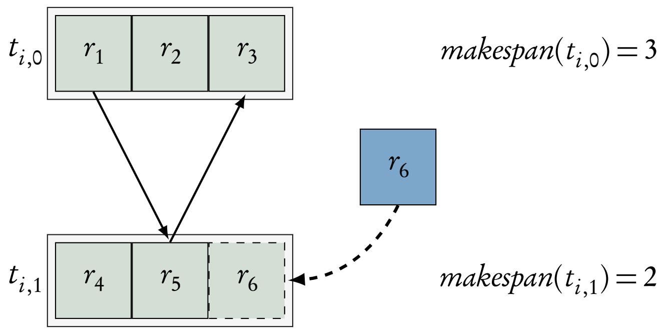


Figure 6.9: Mapping a runnable r_6 to a set of task partitions based on the earliest possible execution heuristic.

With the *max-parents* mapping heuristic, shown in Figure 6.10, the selected runnable is mapped to the task partition with the most preceding parents. The preceding parents for a given runnable r_k is the set of precedences from runnables mapped to the given task partition $t_{i,j}$ to the destination runnable r_k:

$$parents(r_k, t_{i,j}) = \{\mathbf{C}^{l,k} : m(r_l) = t_{i,j}\} \tag{6.6}$$

Using this, the task partition is selected by the maximum number of preceding parents over all task partitions:

$$select(r_k, \mathbf{T}_i) = \max_{\forall t_{i,j} \in \mathbf{T}_i} (|parents(r_k, t_{i,j})|) \tag{6.7}$$

In the example from Figure 6.10, the runnable r_5 is taken from the priority list and mapped to a task partition. The heuristic selects task partition $t_{i,1}$ for r_5 as there are two parents on this task partition ($c_{3,5}, c_{4,5}$) in contrast to $t_{i,0}$ that has only one parent ($c_{2,5}$).

The max-parents heuristic reduces the cross-task partition precedences and thus aims at reducing the number of synchronization points. In addition, a seeding phase enforces an initial seeding on the task partitions by mapping one runnable on each partition before all other runnables are mapped. To break ties, the earliest possible execution heuristic is used.

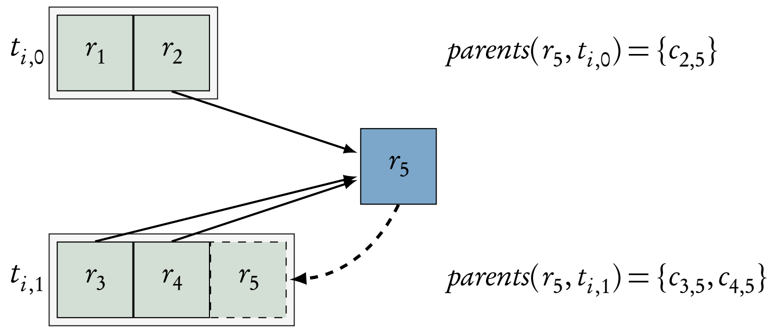

$t_{i,0}$
r_1
r_2
$parents(r_5, t_{i,0}) = \{c_{2,5}\}$
r_5
$t_{i,1}$
r_3
r_4
r_5
$parents(r_5, t_{i,1}) = \{c_{3,5}, c_{4,5}\}$

Figure 6.10: Mapping a runnable r_5 to a set of task partitions based on the max-parents heuristic.

With the *min-distance* heuristic, shown in Figure 6.11, the selected runnable is mapped to the task partition with the smallest *distance* to its parents. The distance is the sum of the oWCET of the runnables that are executed after the last parent runnable. Using the distance, the min-distance mapping heuristic tries to increase the gradient and hence reduces the brittleness of the cross-partition precedences.

Independent Runnables

As independent runnables have no precedences, they can be executed at any time within the task's LET duration. Therefore, we can use remaining idle time slots for independent runnables. This is shown in Figure 6.12. Our method maps the independent runnables by an adapted worst-fit decreasing heuristic [Joh74] after all dependent runnables were mapped. Based on the task partition bins $t_{i,1}, \ldots, t_{i,k}$ that already contain the dependent runnables, the algorithm maps the remaining independent

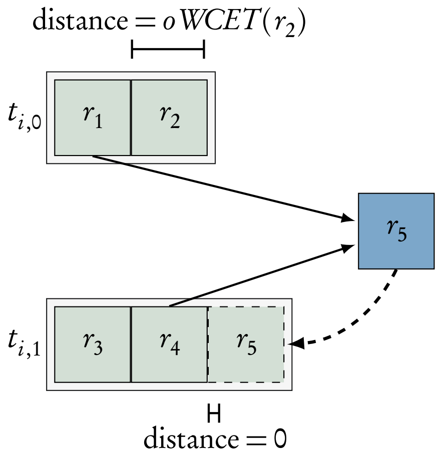


Figure 6.11: Mapping a runnable r_5 to task partitions based on the min-distance heuristic.

runnables $r_1, \ldots, r_l$. The oWCET of a runnable is used as the size w_l for the packing. In the first step, the runnables are sorted in decreasing order based on their size. Then, the greedy strategy of the worst fit decreasing heuristic maps the biggest runnable into the biggest idle slot of all task partition bins. An idle slot is the remaining space between two runnables in the schedule of a task partition bin. The idle slot always has to be greater than the size of the runnable to fit. If there is no fitting idle slot available in all task partition bins, the runnable is appended to the task partition with the smallest makespan. The runtime for this algorithm is upper bounded by $\mathcal{O}(l \log l)$ whereas l is the number of independent runnables.

6.3.3 Synchronization

In this last step of synchronized splitting, the component generates the synchronization that ensures the inter-task precedences of the legacy task. When splitting a task, our method maintains the precedences and thus the dependencies among the runnables. A precedence $c^{i,j}$ ensures that the runnable r_j can only start its execution after r_i has finished. As shown in the previous section, a runnable may execute longer than the specified oWCET bounds. While the precedences within a task partition are guaranteed by the order of execution at design time, the inter-task precedences have also to be ensured at run-time. To ensure these inter-task precedences, we employ synchronization and specifically synchronization points. As explicit synchronization at run-time is costly in terms of resources, we have to analyze for all precedences whether they have to be enforced at run-time by an explicit synchronization point.

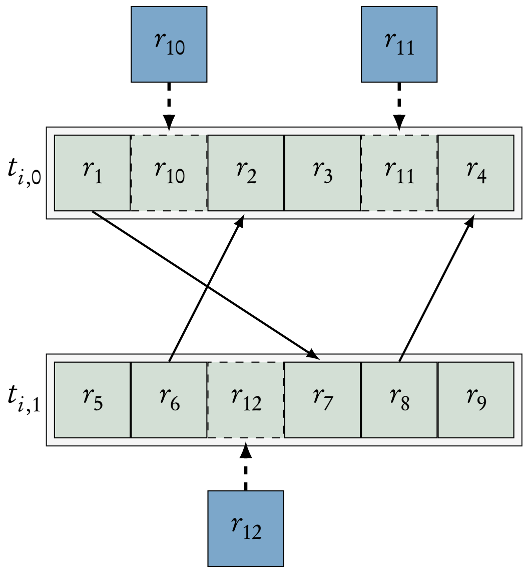


Figure 6.12: Mapping of the independent runnables r_{10}, r_{11}, r_{12} into idle slots using a worst-fit decreasing heuristic.

Our approach benefits from the fact that one synchronization point can guarantee multiple other precedences. In Figure 6.13 we give an example of two task partitions with four runnables each and three precedences. In this example, creating a synchronization point for $c^{2,7}$ would be sufficient to ensure also $c^{1,8}$ such that r_8 runs after r_1. In this case precedence $c^{2,7}$ *dominates* $c^{1,8}$. Graphically speaking, the precedence with the minimum gradient dominates all other constraints that cross it. Again, the gradient of a precedence $c^{i,j}$ is the difference from the start time of r_j to the finish time of runnable r_i. We use this dominating characteristic of the precedences to reduce the number of explicit synchronization points. This can reduce the overall synchronization overhead when executing the task partitions in parallel.

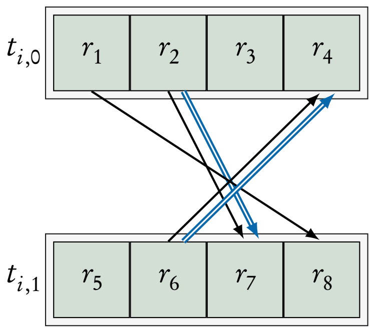


Figure 6.13: Precedence $c^{2,7}$ dominates precedence $c^{1,8}$ such that only $s^{2,7}$ needs to be synchronized which reduces the overhead.

The crossing precedences are determined by Algorithm 4 and works as follows. A precedence crosses another if the source runnable starts at the same time or later (*succ*) and the destination runnable starts earlier or at the same time as the runnables of the given precedence (*pred*). A precedence can only cross another if the direction (i. e., the source and destination task partition) is the same. In Figure 6.13, the precedence $c^{1,8}$ crosses $c^{2,7}$, but not $c^{6,4}$. The result of this algorithm is a sorted list of the crossing precedences sorted based on their gradient.

Algorithm 4 Determine crossing precedences.

```
1: procedure CROSSING($c^{k,l}$, $\mathbf{C}$)
2:     for all precedence $c'^{i,j} \in \mathbf{C}$ do
3:         if $c'^{i,j} \neq c^{k,l}$ and
             $src(c'^{i,j}) \in succ(src(c^{k,l}))$ and
             $dst(c'^{i,j}) \in pred(dst(c^{k,l}))$ and
             $m(src(c'^{i,j})) = m(src(c^{k,l}))$ and
             $m(dst(c'^{i,j})) = m(dst(c^{k,l}))$ then
4:             cc.add($c^{k,l}$)
5:         end if
6:     end for
7:     return cc
8: end procedure
```

Using the lists of crossing precedences, we determine the dominating precedence of the set of all precedences **C** by following Algorithm 5. For each precedence $c^{i,j}$, we find the crossing precedences (line 3). If there are no crossing precedences, the precedence $c^{i,j}$ needs to be synchronized (line 11). If the set of crossing precedences already contains a synchronization point, there is no need to synchronize another precedence with a more loose gradient (line 5–7). In case that all crossing precedences are not synchronized, pick the dominating one, i. e., the one with the minimum gradient and create a synchronization point (line 8–9). The resulting set of synchronization points $\mathcal{S}$ is free of deadlocks because the underlying precedence graph is directed and acyclic.

6.3.4 Integration

Due to the extra synchronization, the resulting task partitions are not fully independent and have to be coordinated. This coordination as part of our splitting approach is performed by one master task partition that manages itself and n other slave task partitions. The coordination and integration of the master and slave task partitions is shown in Figure 6.14. The figure shows a splitting solution with three task partitions. The task partition with the longest makespan becomes the master. This way, the cores

Algorithm 5 Create synchronization for dominating precedences.

```
1:  procedure DOMINATE(C)
2:      for all precedence $c^{i,j} \in \mathbf{C}$ do
3:          $\mathbf{C}_x \leftarrow$ crossing($c^{i,j}$, C)
4:          if $|\mathbf{C}_x| > 0$ then
5:              if $\exists s^{i,j} \in \mathscr{S}_x$ then
6:                  continue;
7:              end if
8:              $c'^{k,l} \leftarrow$ pick-min-gradient($\mathbf{C}_x$);
9:              $\mathscr{S} \leftarrow$ new-sync-point($s^{k,l}$)
10:         else
11:             $\mathscr{S} \leftarrow$ new-sync-point($s^{i,j}$)
12:         end if
13:     end for
14:     return $\mathscr{S}$
15: end procedure
```

executing the slave task partitions are freed as soon as the respective task partition finished its execution and has notified the master. The master task also contains code that has to be executed before (Start) and after (End) all task partitions. The start-code contains the barrier that has to be initialized before the slave task partitions can be activated. Once all start-code was executed, the master activates the slave task partitions via an inter-process activation (IPA). Typically, the slave task partitions start execution of their runnables after a small delay. The reason for the delay is the operating system overhead when activating tasks across cores. The master also provides a barrier at the end of its execution for all slave task partitions to synchronize with it. As soon as a slave task partition finishes execution, it notifies the master's barrier and frees up the core. Hence, the core that executes the master is blocked until it receives all notifications from the slave task partitions. In practice however, the master only has to wait for notifications when runnables of slave task partitions overrun its oWCET or are preempted by higher priority tasks. After the master received all notifications, optional end-code is executed to e. g., reset variables or free up memory space.

From the implementation point of view, the runtime environment has to enforce the synchronization points. In the automotive domain, this is typically achieved by the OSEK operating system [OSE05]. OSEK provides extended tasks (as introduced in Section 2.1.2) that allows a task to go into a waiting state and requires an event to continue. Due to the significant resource overhead of extended tasks, we implemented synchronization points using basic tasks. Our lean implementation uses events as synchronization primitives that act as barriers. Hereby, the source runnable of a synchro-

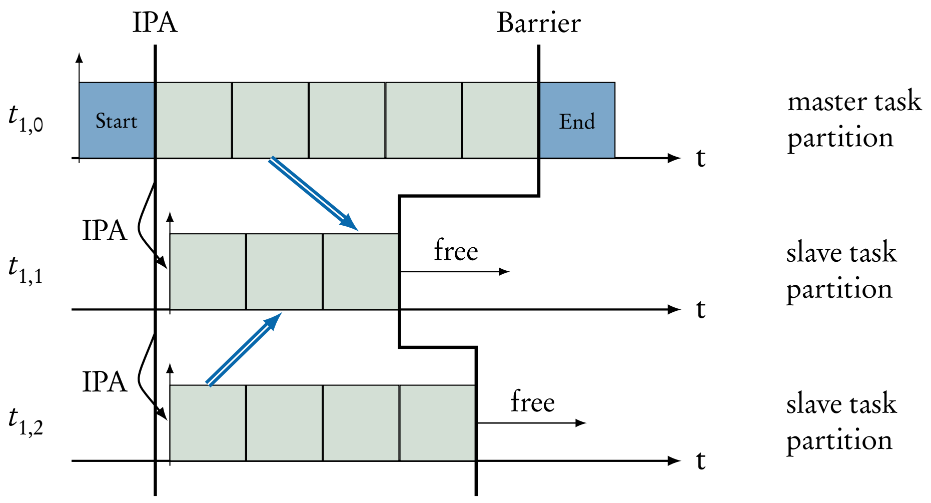


Figure 6.14: Integration of the task partitions by using a coordinating master task that manages the slave tasks.

nization point sets a specific event after it finished execution, while the destination runnable performs a busy wait for the event and resets it afterwards.

Obviously, the performance benefit depends on the overhead of the synchronization implementation. Our busy waiting implementation has a very low overhead for task splittings with a high speedup and low brittleness which is also the main goal of our approach. For splittings with a high brittleness, other implementations such as extended tasks may incur less overhead and are better suited. An overview of related synchronization techniques was published by Gerdes et al. [GKU$^+$12].

6.4 Summary

In this chapter we have presented our Splitting component to efficiently split tasks into multiple task partitions and to evaluate their level of concurrency. In contrast to other approaches, the splitting component can handle Observed Worst-Case Execution Time such that the splitting solution can be optimized for robustness against oWCET overruns. Based on the precedence graph that is generated from the dependency graph, a task can be split with and without synchronization. While ensuring all precedences, both splitting modes maintain the behavior of the legacy task. Unsynchronized splitting creates fully independent task partitions by analyzing the task's design for connected components. Splitting with synchronization allows to exploit even more concurrency from the task's design but introduces synchronization. We have shown an algorithm that reduces this synchronization overhead dramatically and a lean implementation for synchronization with OSEK basic tasks.

7 Relaxation

In this chapter we present the components to advance from the limited available concurrency of a legacy task toward its inherent concurrency. While the available concurrency stems from single-core design principles, the inherent concurrency reflects the actual physical requirements of the environment. Legacy tasks have strict requirements regarding the communication's timing to maintain its functional behavior. This strict communication timing limits the concurrency of a task by a high degree but is in fact not always necessary. Some communication dependencies may not require such a strict timing and can be *relaxed*. Relaxing the timing of a communication dependency increases the task's concurrency but can also potentially change the behavior. Hence, a domain expert is needed to evaluate that the functionality is still within its specifications and correct. However, due to the huge evaluation effort, not all dependencies can be evaluated. Therefore, our *Concurrency Analysis* component rates all dependencies to find the most relevant and promising dependency candidates for evaluation. The analysis is looking for potential evaluation candidates for which the timing can be relaxed without compromising the correctness of the application. The most promising evaluation candidate is then presented to a domain expert as part of our *Dependency Analysis* component. This component provides the interface to the expert and retrieves the expert's decision.

7.1 Relaxing Dependencies

Relaxing a dependency is a change in the design of a task as well as in its behavior and thus is a crucial modification. In our approach, a task's design is represented by its dependency graph containing all communication and all known functional requirements in the form of dependencies. In the following, the communication dependencies are especially important to the relaxation process. Functional dependencies are typically already specified and verified by a domain expert. When starting the relaxation process, we have to assume that all communication dependencies are either strictly delayed or strictly undelayed. To relax a dependency in this graph, we change the timing of the communication which may alter the functional behavior. We define the term *relaxation* as follows:

Definition 7.1. *A relaxation is the verified transition of a single communication dependency q_k between two runnables r_a and r_b from the criticality strict delayed $\gamma(q_k) = 1$ or strictly undelayed $\gamma(q_k) = 0$ to relaxed $\gamma(q_k) = 0..1$.*

Performing a relaxation always has to be verified by a domain expert or other means such that the functional behavior remains correct. Hence, relaxing a communication dependency is a two-step process:

1. Verify that the change in timing of the communication dependency maintains the functional behavior.
2. Change the timing of the communication dependency to relaxed.

The benefit of this relaxation transition is the potentially improved concurrency in the task's design. In case all communication dependencies between two runnables are relaxed and there is no functional dependency, then there is also no need for a precedence requirement among the two runnables. A removed precedence improves the task's concurrency that in turn can be exploited for parallelism by our Splitting component. It is important to note however, that if the verification step fails, the result is still beneficial to the overall approach. When a domain expert has verified the fact that a communication dependency needs to have a strict timing, then it is safe to assume that the physical environment requires it. In this case our approach can safely skip this dependency in future iterations.

For our approach, single dependencies are not relaxed for practical reasons. Instead, we relax the dependencies that are related to a pair of runnables. Only when all dependencies between a pair of runnables are relaxed and therefore do not require a precedence, the concurrency of a task's design is improved. Such a pair is denoted as $\langle r_a, r_b \rangle$. A relaxation is applied to the dependency graph for a given pair of runnables $relax : DG, \langle r_a, r_b \rangle \rightarrow DG$ that is defined by Algorithm 6. The algorithm first retrieves all dependencies that are related to the given pair of runnables (line 2). This resulting set of dependencies is then checked for dependencies that cannot be relaxed (line 3). If all dependencies in this set can potentially be relaxed, they are given to the domain expert for evaluation (line 4–7). Subsequently, the expert's decisions modify the dependency graph.

The naive approach to find the related dependencies of a runnable pair would be to collect all dependencies between such pair, independent of their direction. However, the application may also contain dependencies for multi-writers. In case of a multi-writer, there are at least two dependencies communicating the same label to the same runnable. Figure 7.1 illustrates this by using the mode-switching example from before. It shows a dependency graph for a task with five runnables and two multi-writers. Namely, r_2 and r_3 write to the same label d_x that is read from r_4. We assume that we want to relax the pair of runnables $\langle r_3, r_4 \rangle$ to improve the task's concurrency. Using the naive approach as described before, the related dependencies are all dependencies between the

Algorithm 6 Relaxation of a pair of runnables.

```
1: procedure RELAXPAIR($DG, \langle r_a, r_b \rangle$)
2:     $\mathbf{Q}_x \leftarrow getRelatedDependencies(DG, \langle r_a, r_b \rangle)$
3:     if $isRelaxable(\mathbf{Q}_x)$ then
4:         for all $q_k \in \mathbf{Q}_x$ do
5:             $DG \leftarrow relax(q_k)$
6:         end for
7:     end if
8:     return $DG$
9: end procedure
```

runnables r_3 and r_4. In this example, this is only the dependency $q_x^{3,4}$. If this dependency is relaxed, the precedence between r_3 and r_4 is removed. But without this precedence, we can construct the following execution sequence: $r_1 \prec r_2 \prec r_4 \prec r_3$. The problem here is that the value written by r_3 can never be read by r_4, because it is overwritten every time by r_2. Despite that the timing of $q_x^{3,4}$ is correctly relaxed, the communication behavior of the multi-writers has changed. This is a general problem with multi-writers: When relaxing an incoming communication dependency of a runnable that receives data from different runnables via the same label, the change in timing also affects the other writing dependencies. Despite that the writing order is maintained by the functional dependency, the timing of the reading runnables influence each other.

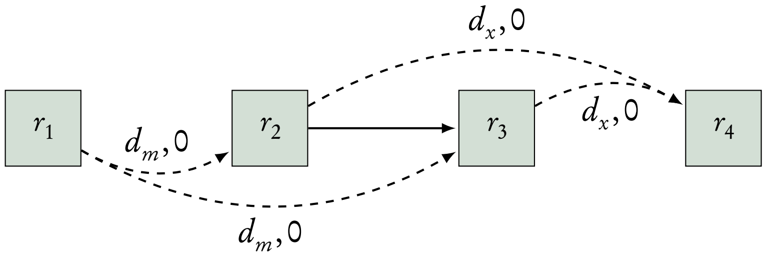


Figure 7.1: Relaxation of dependencies in a multi-writers scenario using the example of a dependency graph for mode-switching.

Thus, when evaluating a communication dependency that is part of multi-writing, all other dependencies that write to the same label have to be evaluated as well. In our example, we need to evaluate $q_x^{2,4}$ additionally to $q_x^{3,4}$. The dependency $q_x^{2,4}$ is an incoming communication dependency to r_4 and writes to the same label d_x. When both dependencies $q_x^{2,4}$ and $q_x^{3,4}$ are presented to the domain expert for evaluation, the expert has the complete multi-writer picture and can evaluate it accordingly. This way, it is ensured that the timing in a multi-writer scenario is evaluated as a closed group.

Theoretically, it is also possible to completely eliminate the multi-writing by removing the functional dependency that ensures the order of writing. But to do that, we need to know the functionality of the writing runnables and under which conditions they write to the label. Unfortunately, this is not possible from our level of abstraction.

Algorithm 7 specifies how all dependencies that relate to a pair of runnables in a dependency graph are determined. The algorithm includes the following related dependencies:

(1) All dependencies, communication and functional, between the two given runnables in either direction (line 2).
(2) All incoming dependencies of a target runnable that have the same label as one of the dependencies between the given pair of runnables. This ensures that communication dependencies of multi-writers are relaxed as a group (line 3–7).

Algorithm 7 Get all dependencies that relate to a pair of runnables for relaxation.

```
1: procedure GETRELATEDDEPENDENCIES($DG, \langle r_a, r_b \rangle$)
2:     $\mathbf{Q}_x \leftarrow \mathbf{Q}^{a,b} \cup \mathbf{Q}^{b,a}$
3:     for all $q_k \in \mathbf{Q}_C : dst(q_k) = r_a$ or $dst(q_k) = r_b$ do
4:         if $\exists q_i \in \mathbf{Q}_x : label(q_k) = label(q_i)$ and $dst(q_k) = dst(q_i)$ then
5:             add $q_k$ to $\mathbf{Q}_x$
6:         end if
7:     end for
8:     return $\mathbf{Q}_x$
9: end procedure
```

Based on the related set of dependencies for a pair of runnables, Algorithm 8 determines if the dependencies meet the requirements for relaxation. The requirements are:

(1) The set does not contain a functional dependency.
(2) The set does not contain any dependency that is already evaluated and still requires a strict communication timing.

If one of these requirements are not met, the whole set of dependencies is not allowed to be relaxed.

The actual relaxation sets a communication dependency to relaxed and evaluated. This is described by Algorithm 9. The algorithm provides the interface to the domain expert and is part of the Dependency Analysis component described in Section 7.6. The expert is asked a boolean question if the given dependency can be relaxed, described by the *eval* function (line 2). If the expert approves the relaxation, the timing of the dependency in the dependency graph is set accordingly (line 3–5). In either case, the status of the dependency is set to evaluated (line 6).

Algorithm 8 Determine if a set of dependencies can be relaxed.

```
1: procedure ISRELAXABLE($\mathbf{Q}_C$)
2:     if $\exists q_k \in \mathbf{Q}_C \Rightarrow q_k \in \mathbf{Q}_F$ or
           $\exists q_k \in \mathbf{Q}_C \Rightarrow \gamma(q_k) \neq 0..1 \wedge \sigma(q_k) \neq t$ then
3:         return false
4:     else
5:         return true
6:     end if
7: end procedure
```

Algorithm 9 Relaxation of a communication dependency.

```
1: procedure RELAX($DG, q_k$)
2:     $relax \leftarrow eval(q_k)$
3:     if relax then
4:         $\gamma(q_k) := 0..1$
5:     end if
6:     $\sigma(q_k) := t$
7:     return $DG$
8: end procedure
```

7.2 Workflow

Due to the huge amount of dependencies in legacy automotive applications, the biggest challenge is to find the most relevant ones for relaxation. In this section we present the overall workflow to find relevant dependencies, depicted in Figure 7.2. This workflow is implemented into the Concurrency Analysis component. Input for the component is the dependency graph, the precedence graph and the partitioning information of a task.

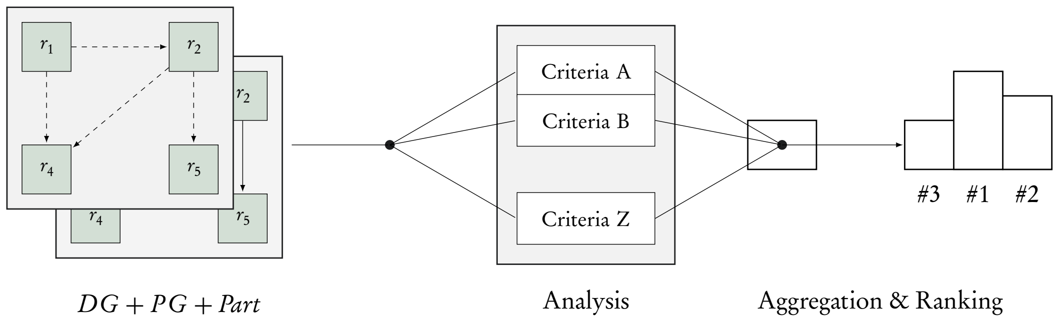


Figure 7.2: Overview of workflow inside the Concurrency Analysis component.

Based on the input, the analysis checks every pair of runnables with its related dependencies. Every pair is a candidate for evaluation and every candidate is analyzed according to a number of independent criteria. The analysis consists of criteria with each criterion having its own objective. The criteria are grouped in the following way:

- The *impact* group indicates the potential benefit of a dependency, if it is relaxed, toward the given parallelization goals. Using graph algorithms, this group analyzes properties of the communications between the runnables. The input of a criterion of this group is a pair of runnables and the corresponding precedences from the precedence graph.
- The *success* group rates how likely it is that the criticality of a dependency can be relaxed without significantly degrading the functional behavior. This can be quantified by the analysis of the physical dynamics a dependency represents. The input of a criterion of this group is a pair of runnables with all dependencies inbetween from the dependency graph.

The result of a criterion is a value between 0–1 and indicates either the success or impact of the analyzed objective on the given dependency or precedence (higher is better). Table 7.1 gives an overview of all criteria in our analysis, their respective use case and aggregation group. The criteria are described in detail in the following sections of this chapter.

Table 7.1: Overview of criteria and their respective use case and aggregation group.

	Strategy		Group	
Criteria	Early	Late	Impact	Success
Synchronization Points[1]	—	✓	✓	—
Forward Evaluation	✓	✓	✓	—
Dependency Classification	✓	✓	—	✓
Reaction Constraints	✓	✓	—	✓

[1] Can only be used if the parallelization allows synchronization.

The results of the analysis are weighted and aggregated for each pair of runnables to a score that represents the relevance for the evaluation by an expert. Based on the score, a ranking is created. The candidate with the best score becomes the result of the concurrency analysis and is marked for evaluation. In the next step the marked candidate is proposed to the domain expert.

7.3 Impact Criteria

An *impact* criterion indicates the benefit toward the parallelization goal if the corresponding dependencies of a precedence are relaxed. Impact criteria are only applied to precedences and specialized toward a single objective. An impact criterion is a normalized function $impact : \mathbf{C} \rightarrow [0, 1] \in \mathbb{R}$. An impact result of 0 means that relaxing all dependencies that correspond to the given precedence has no positive effect on the objective for the parallelization goal. When the impact is 1, relaxing the corresponding dependencies results in the highest possible benefit in respect to the criterion's objective. Inbetween, the benefit is relative to the impact. There are two impact criteria which we present in the following:

Synchronization Points: Synchronization due to precedences is expensive in terms of robustness and overhead. The relaxation of dependencies that correspond to a precedence can improve the robustness and reduce the synchronization overhead.

Forward Evaluation: The relaxation of dependencies that correspond to a precedence can improve the concurrency among the runnables in a task. This criterion analyzes the potentially gained concurrency.

7.3.1 Synchronization Points

Any synchronization within a task is costly in terms of robustness and overhead. A synchronization point always adds an overhead to the makespan, if it is used or not during execution. As shown in Section 6.3.3, our overall goal is to reduce the number of synchronization points because a synchronization can also decrease the robustness. Additional synchronization influences the robustness since Observed Worst-Case Execution Time (oWCET) overruns may result in makespan penalties. Here, the relaxation of dependencies has the potential that a corresponding synchronization point is improved in terms of its gradient or even removed from the task completely. This is an impact criterion as it indicates the benefit toward the parallelization goal in terms of synchronization. Obviously, this criterion is only enabled when the parallelization goal allows synchronization.

This criterion exploits the effect, that other precedences can potentially replace an existing synchronization point. When the dependencies of a pair of runnables are successfully relaxed, the precedence between the two runnables is removed. If this precedence previously resulted in a dominating synchronization point for other precedences, then another synchronization point has to be created. In such case, the replacing precedence and synchronization point has a better gradient than the previous one. Selecting the next best synchronization point is described in the optimization process in Section 6.3.3 and reused here. In general, evaluating the benefit toward synchronization when relaxing a dependency is twofold:

- The relaxation reduces the brittleness as synchronization points can safely be removed or another precedence creates a synchronization points that has a better gradient.
- The relaxation decreases the overall makespan when there is idle time in the schedule due to a synchronization point.

The input for the synchronization point criterion is one precedence from the PG of a task t together with the *Part* information. Based on this input, the synchronization point criterion analyzes the precedence in the context of the precedence graph. We found three different cases that influence the impact toward robustness/brittleness of the given precedence $c^{a,b}$ which we elaborate in the following.

The first and most trivial case is when the given precedence $c^{a,b}$ does not result in a synchronization point with the given partitioning information. In this case, the relaxation of the dependencies that relate to the given precedence have no effect on the synchronization and thus will not improve the task toward its parallelization goal. Clearly, the impact is 0.

In the second case the precedence $c^{a,b}$ results in a synchronization point as specified in the given partitioning information. The precedence $c^{a,b}$ can only be removed if all communication dependencies are relaxed and there is no functional dependencies inbetween them. In other words, when there is only one unrelaxed communication dependency or a functional dependency between r_a and r_b, the precedence will remain and so will the synchronization point. In this case, the criterion evaluates the benefit when the given precedence is removed by relaxing all related dependencies between the source and destination runnable. Due to the optimization of the synchronization points (see Section 6.3.3), another precedence may replace the given one as a synchronization point when relaxed. Still, replacing the synchronization point with another one can be beneficial. The benefit is, that the replacing precedence c_r leads to a synchronization point with a better gradient than the old precedence $c^{a,b}$, i. e., $grad(c_r) > grad(c^{a,b})$.

Algorithm 10 describes how to determine the precedence replacement. The algorithm works similar to the algorithm that reduces the number of synchronization points. Based on the reduction algorithm from Section 6.3.3, we can safely assume that the given precedence is the one with the smallest gradient in the set of crossing precedences. The algorithm follows this assumption. In the set of all precedences, it looks for precedences that cross the given one (line 3–7). In contrast to the reduction algorithm, the algorithm looks for predecessors of the source runnable and successors of the destination runnable. The resulting set is again sorted based on the gradient (line 9) and the precedence with the smallest gradient is picked as potential replacement c_r (line 10).

If a replacing precedence is found, the calculation of the impact is described in Equation 7.2. It assumes that $c^{a,b}$ is the original precedence and c_r is the replacement. The impact is calculated as

Algorithm 10 Determine the replacing precedence.

```
 1: procedure FINDREPLACINGPRECEDENCE($c^{a,b}$, $\mathbf{C}$)
 2:     $L_x \leftarrow \{\}$
 3:     for all $c_j \in \mathbf{C}$ do
 4:         if $c_j \neq c^{a,b}$ and
              $src(c_j) \in pred(src(c^{a,b}))$ and
              $dst(c_j) \in succ(dst(c^{a,b}))$ and
              $m(src(c_j)) = m(src(c^{a,b}))$ and
              $m(dst(c_j)) = m(dst(c^{a,b}))$ then
 5:             add $c_j$ to list $L_x$
 6:         end if
 7:     end for
 8:     if $L_x \neq \emptyset$ then
 9:         sort($L_x$, $grad$)
10:         return $c_r \leftarrow \min(L_x)$
11:     else
12:         return null
13:     end if
14: end procedure
```

$$c_r = findReplacingPrecedence(c^{a,b}, \mathbf{C}) \tag{7.1}$$

$$\begin{aligned} impact_{SP}(c^{a,b}) &= \overbrace{\frac{makespan}{grad(c^{a,b}) + makespan}}^{\text{weighting factor}} \times \overbrace{\frac{grad(c_r) - grad(c^{a,b})}{makespan}}^{\text{normalized difference}} \\ &= \frac{grad(c_r) - grad(c^{a,b})}{grad(c^{a,b}) + makespan} \end{aligned} \tag{7.2}$$

where the normalized difference is the difference in the gradient of the replacing precedence to the original precedence. As $grad(c_r) > grad(c^{a,b})$ holds, the difference can be normalized by the overall makespan. Additionally to the difference in the gradient, also the gradient of the replaced precedence has an impact on the benefit. As shown in Figure 7.3, when the difference in the gradient is the same, replacing a precedence with a gradient of zero is better than replacing a precedence with a gradient > 0. The reason is that the improvement in robustness decreases exponentially with the gradient of the replaced precedence. This is represented by the weighting factor.

A special case is when there is a precedence replacement which has a better gradient and there is also idle time before the receiving runnable. When the resulting synchronization point has a gradient of zero, the relaxation additionally improves the makespan. Figure 7.4 illustrates this additional benefit. Here, the precedence c_k is removed due to relaxation and replaced by precedence c_r. Before the removal of c_k, the resulting synchronization point created idle time between r_3 and r_4. This idle time disappears with the replacing precedence c_r and also improves the makespan.

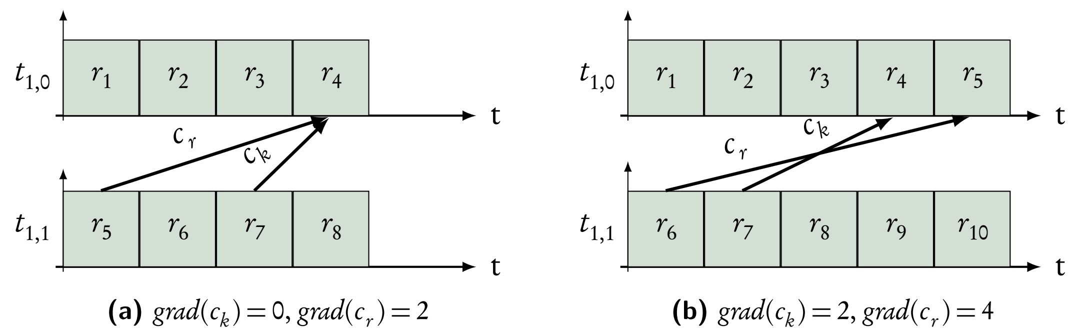


Figure 7.3: Two scenarios where the difference in gradient is the same but the original precedences differ. We rate the left scenario higher because robustness increases exponentially with the gradient.

In the third case, the given precedence $c^{a,b}$ results in a synchronization point. When relaxing the corresponding dependencies, the precedence is removed and there is no other precedence that replaces the given one. In this case, the Algorithm 10 returns *null*. The synchronization is completely removed which is the best possible outcome when relaxing in terms of synchronization. Hence, the impact criterion results in the value 1.

Combining all cases, the impact is calculated as follows assuming the replacing precedence $c_r = \mathit{findReplacingPrecedence}(c^{a,b}, \mathbf{C})$:

$$impact_{SP}(c^{a,b}) = \begin{cases} 0, & s^{a,b} \notin \mathcal{S} \ . \\ \frac{grad(c_r) - grad(c^{a,b})}{grad(c^{a,b}) + makespan}, & c_r \neq null \ . \\ 1, & c_r = null \ . \end{cases} \tag{7.3}$$

In future work we consider probabilistic worst-case execution times to improve the ranking of the synchronization points. With probabilistic Worst-Case Execution Times (WCETs) we know the chances for a oWCET overrun and at what costs. This could improve the calculation of the impact when relaxing related dependencies of a precedence that leads to a synchronization point.

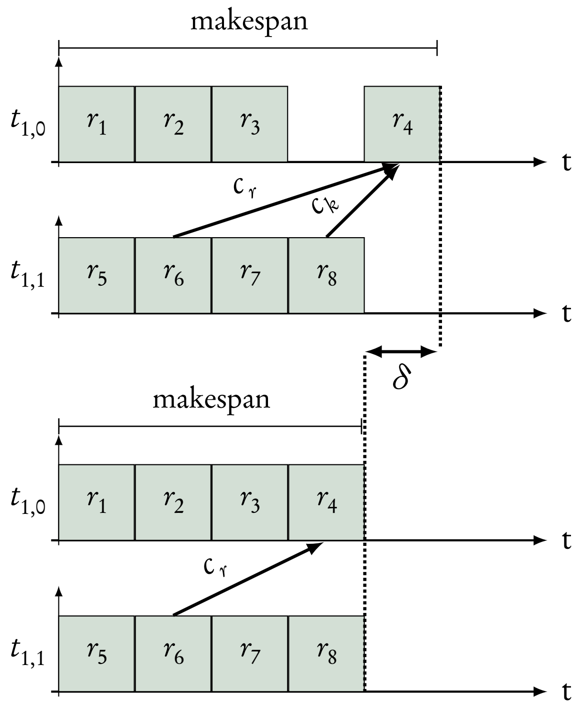
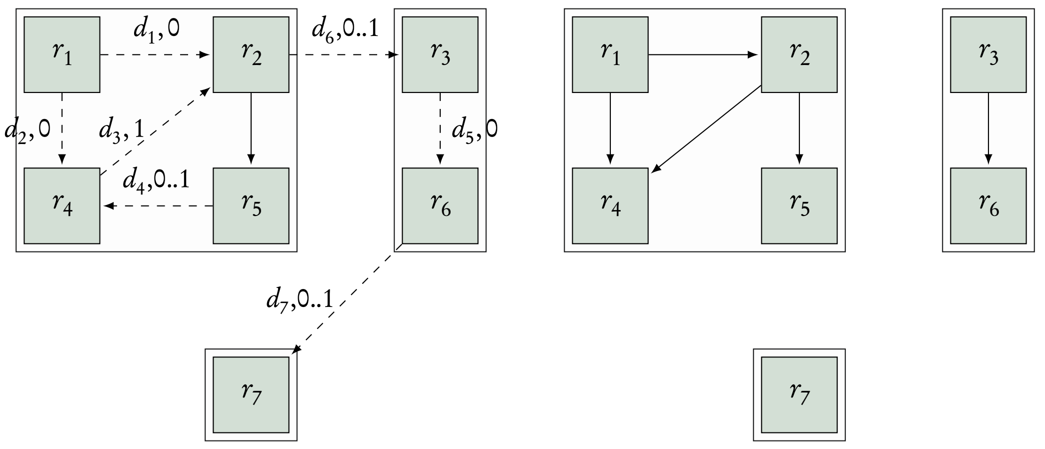


Figure 7.4: Improvement of the makespan when the corresponding precedence to a synchronization point with idle time is relaxed.

7.3.2 Forward Evaluation

One of the high level objectives of our approach is to improve concurrency and exploit it for parallelization. Thus we need to evaluate the benefit of relaxing dependencies toward the parallelization goal in terms of concurrency. For this purpose, the forward evaluation criterion simulates the influence of a relaxation and measures the resulting concurrency. Clearly, the objective of this criterion is concurrency.

With this criterion we measure concurrency by finding (weakly) connected components in the precedence graph, similar to the unsynchronized splitting from Section 6.2. The criterion is however independent of the splitting mode. Given a precedence graph PG, a connected component is a maximal subgraph such that all nodes of the subgraph are reachable from every other in the underlying undirected graph. I. e., there is an undirected path from r_a to r_b and a directed path from r_b to r_a. Each component in a precedence graph represents a set of runnables that has unrelaxed dependencies inbetween. Due to these dependencies, the runnables communicate with a strict timing and thus have to follow a specific order of execution as specified in the precedence graph. Between components there are no precedences which implies that there are no or only relaxed dependencies. Figure 7.5 gives an example precedence and dependency graph with three weakly connected components.



(a) Dependency Graph **(b)** Precedence graph

Figure 7.5: Weakly connected components in a precedence graph and its implied dependency graph used in the forward evaluation criterion.

A connected component in a precedence graph has the property that all communication to the outside of the component is either relaxed or non-existent. Therefore, a component can be executed, as a group of runnables and precedences, at any point in time within the boundaries of the Logical Execution Time (LET) of the task. Each component inside a task is concurrent to another. Furthermore it is even possible to extract a component as an exclusive task with the same recurrence as the original task. The number and size of components in a precedence graph hence indicates the level of concurrency inside a task.

To determine the components in a precedence graph we use the algorithm by Hopcroft et al. [HT71]. Let $CC_i \in CC(PG)$ be the i-th weakly connected component in the set of components of the precedence graph PG. The component CC_i is the maximal subgraph such that all nodes of the subgraph are reachable from every other in the underlying undirected graph. We denote $|CC_i|$ as the number of nodes in the subgraph CC_i. The algorithm $Hopcroft : PG \rightarrow CC$ finds all components of the precedence graph PG in linear time of $max(|\mathbf{V}|, |\mathbf{E}|)$. As relaxing a precedence can create new components, we can evaluate the potential benefit by simulating the relaxation of the given precedence. We measure the impact by means of the change in the components structure with and without this assumption. First of all, one relaxed precedence can create one additional component at most. This is denoted by the boolean function

$$createCC(c^{i,j}) = |Hopcroft(relax(DG, \langle r_i, r_j \rangle))| - |Hopcroft(PG)| \quad . \tag{7.4}$$

If a precedence $c^{i,j}$ creates an additional component when the corresponding dependencies are relaxed, *createCC* results in 1, otherwise the result is 0.

If an additional component is created, we differentiate it by its consisting runnables and relation to other components. The objective hereby is to create well-balanced components. We denote the oWCET of a component $oWCET(CC_i)$ as the sum over all runnables' oWCET of CC_i and use the share per component in the set of components to determine the balance

$$I_{CC_i} = oWCET(CC_i) / \sum_{CC_k \in CC} oWCET(CC_k) \ . \tag{7.5}$$

The components are equally balanced if the standard deviation σ of the component's share is zero. To favor additional components that add a benefit to the balance, we use $1-\sigma$. All combined, the aggregated criterion $impact_{\mathrm{FE}}$ is

$$impact_{FE}(c_k^{i,j}) = \begin{cases} 1-\sigma, & \text{if } createCC(c^{i,j}) = 1 \ , \\ 0, & \text{otherwise} \ . \end{cases} \tag{7.6}$$

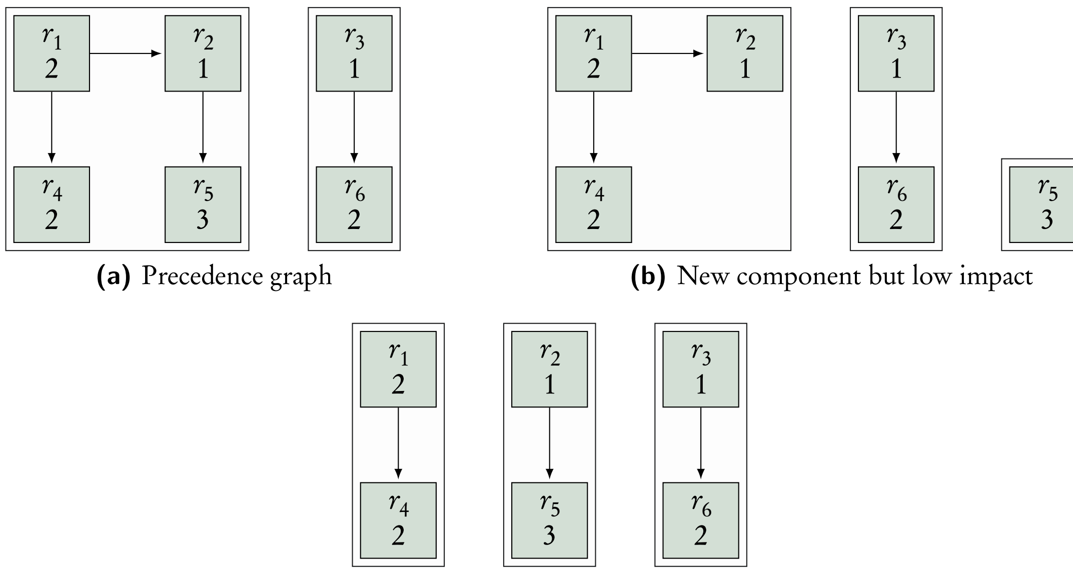
The function $impact_{FE}$ covers the two cases: (1) When the precedence does not create a new component when relaxed, $impact_{FE}$ returns zero (2) When relaxing the precedence creates a new component, $impact_{FE}$ returns the benefit toward the components' balance.

Figure 7.6 shows an example wherein the $oWCET$ of the runnables is annotated below the node name. The original precedence graph is given in Figure 7.6a with six different runnables and four precedences. Based on this graph, the impact is evaluated for the precedence $c^{2,5}$ and $c^{1,2}$. Figure 7.6b shows the components when the dependencies for precedence $c^{2,5}$ would be relaxed. In this case, the relaxation creates a new component with the single runnable r_5. The $impact_{FE}(c^{2,5})$ for this scenario is 0.895. Figure 7.6c shows the components when the dependencies for precedence $c^{1,2}$ would be relaxed. In this case, the relaxation creates a new component with the two runnables r_2 and r_5. The $impact_{FE}(c^{1,2})$ for this scenario is 0.948.

In future work we also want to support the evaluation of more than one precedence. Currently, the forward evaluation criterion is local to one precedence for which the dependencies should be relaxed. However, a new component may only be created if the precedences of more than one pair of runnables are relaxed.

7.4 Success Criteria

A *success* criterion indicates the probability of success in the terms that a communication dependency is granted to be relaxed by an expert. There are a multitude of ways for



(a) Precedence graph

(b) New component but low impact

(c) New component and high impact

Figure 7.6: Resulting precedence graphs for the forward evaluation criterion when the dependencies for precedence $c^{2,5}$ and $c^{1,2}$ are relaxed.

a domain expert to evaluate if a dependency can be relaxed. This kind of criterion anticipates parts of the evaluation that a domain expert would do. From the set of tools for evaluating a dependency, some can be done automatically. Hence, a success criterion supports the manual evaluation process of the expert. It can perform checks prior to the dependency analysis by the expert.

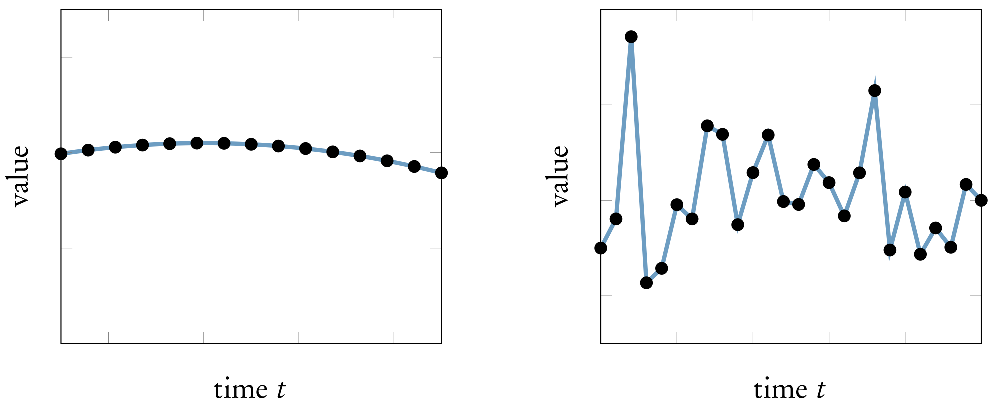
A success criterion uses local properties of a dependency as well as the context of the whole dependency graph. The criterion is only applied to communication dependencies. Functional dependencies are left out, because they are either user-defined or derived from other communication dependencies. A success criterion is the normalized function $\mathit{success} : \mathbf{Q}_C \rightarrow [0,1] \in \mathbb{R}$. If the result of *success* is 0, chances are extremely low that an expert will allow a relaxation after a manual evaluation. If the result is 1, the objective of the criterion is fulfilled and it is likely that an expert grants the relaxation. Inbetween, the benefit is relative to the success. In the following we present two success criteria:

Dependency Classification: The criterion analyzes the dynamics of the data that is communicated via a dependency and how a change in the timing may have an impact on the values.

Reaction Constraints: When event chains with timing constraints are defined for the application, then this criterion analyzes if a change in the timing may violate such a constraint.

7.4.1 Dependency Classification

The dependency classification criterion targets the following main question: How does the value of a label change over time? Based on the answer to this question we can evaluate if and how a delay in the communication of a label has an effect on the behavior. For example, when the outside temperature of a car is used for control purposes. Figure 7.7a shows an example plot of the development of the outdoor temperature label over time. While the blue line represents the actual value, the black dots represent the sampled values that are stored in the label. Adding a delay to the communication of such a temperature label may have only a neglecting effect on the behavior due to the physics of the environment. However, this is also dependent on the sampling interval. On the other hand, the value of a crank-angle based label follows a different trend as shown in Figure 7.7b. The runnables that read such a crank-angle based label might corrupt the behavior with a delayed communication. Hence, labels that represent a physical value are sensitive to changes in the environment. Some labels are highly sensitive such that the trend of the corresponding value is unsteady and uncertain to predict. For other labels the trend of the value is predictable to some degree and follows a steady development. Another motivation for this criterion is that runnables might perform an oversampling of a sensor due to architectural reasons. Such an oversampling might not be necessary as the runnable would emit the same behavior at a slower frequency. This criterion elaborates if the behavior of the system is maintained even when a runnable reads an older value from a label that was updated in a previous job instance.



(a) Label with low time-sensitivity **(b)** Label with high time-sensitivity

Figure 7.7: Sensitivity of a label in terms of the change in its value over time.

The motivation for this criterion is to examine the change of a label's value over time and its resulting effect on the behavior. In general, every communication dependency in the dependency graph relates to a communicated label from one runnable to another. In control-engineered systems, these runnables are typically the implementation of time-discrete controllers, observers, and estimators. Thus, the labels represent mostly data from continuous parts of the system and discrete state machines e. g., for modes. While a label typically represents a value from the continuous environment, they become discrete values by the sampling of the system. Yet, many properties of the value from the continuous environment still apply to the discrete value of the label. These labels are the in- and output of runnables and form a relation together with the runnable as illustrated in Figure 7.8. In this relation, the change of a label's value over time from the continuous environment can be used for a sensitivity analysis in the discrete system.

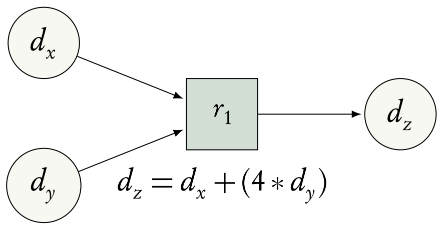


Figure 7.8: Relation of two labels d_x and d_y that are input to runnable r_1 and the output label d_z.

As each runnable represents complex control-engineering algorithms, each input label potentially has an influence on the output label. Given the mathematical representation of the runnables, we can create a system of equations between the in- and output labels. Obviously, this works only for the continuous parts of the system. For hybrid systems, i. e., systems with switches, the control-flow has to be analyzed such that each mode with only continuous parts can be evaluated.

Based on this system of equations, one can analyze and quantify changes of a label's value. In the given example, we can quantify the change in the label's value d_x of 1 leads to a change of the label's value d_y by 4. This method is called sensitivity analysis. The sensitivity analysis studies the relation of uncertainty between the output of a model to the input. In our case, the model is a runnable, in- and output are labels and the sensitivity analysis is local, i. e., it is specific to a change of a single label. The local sensitivity analysis is based on taking the derivative of the output d_y with respect to an input factor d_x at some fixed point [Cac07, SRA+08]. Of a function

$$f : R^n \to R, y = f(x_1, \ldots, x_n) \tag{7.7}$$

the derivative is

$$dy = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(x_1, \ldots, x_n) dx_i \quad . \tag{7.8}$$

For this criterion we specifically want the evaluate the sensitivity over time. Hence the question is: What is the effect of a change in a label's value at time t_0 to a time $t > t_0$? By transforming the time-discrete dynamic system into its state representation, the system of equations can be simulated and the result shows the sensitivity over time for a given label.

However, such a sensitivity analysis needs the equations and in case of a hybrid system also all modes and the corresponding equations to work properly. For a hybrid system at the scale of an Engine Management System (EMS), creating and simulating these equation systems is very time-consuming and a huge effort. While a sensitivity analysis is a powerful method to determine the effect of changes in a label's value, we focus here on a more scalable solution that follows this idea.

Instead of representing a label's value in the context of equations, our solution is based on the technical representation of a label. From the technical perspective, a label represents an Electronic Control Unit (ECU) variable. ECU variables are described using A2L [ASA15a], the standard description format for measurement and calibration data in the automotive industry and introduced in Section 2.3.1. It describes e. g., the data type, format, and computation method used for measurement and calibration purposes. Thus, A2L creates a context for a label's value. As with the sensitivity analysis, the labels that represent a continuous value are of interest for our solution. Especially for these labels, one can extract unit, range, precision and other contextual information from A2L. For example, a label that represents the oil temperature of an engine is a measured value, it is measured in Celsius and is stored with a 3-decimal digits precision. A2L also stores the task with the runnable that samples the label's value which provides the sampling rate. A control engineer can interpret this A2L information and decide how sensitive the corresponding value of the label is. The sensitivity hereby is solely based on the A2L information that is annotated to a label and thus more generic than the sensitivity analysis of systems of equations. Based on this contextual information and the control engineer's knowledge on how the label represents the dynamics of the value from the continuous environment, the sensitivity can be estimated. In our example of the engine's oil temperature, the control engineer likely realizes that the label has a low sensitivity over time. The next step is then to decide how the sensitivity is affecting the behavior. This is in the hands of a control engineer as well. In our case, the control engineer might decide that the effect on the behavior is neglecting. Any temperature in the given embedded system changes slower than any sampler could detect a change. Obviously, this decision is generalized, depending on the domain and specific to the application.

To prevent the interpretation of A2L of each and every label, control engineers can create classes with similar time-sensitivity prior to the analysis. As the A2L information is standardized, all attributes and value ranges are known beforehand. Using the A2L specification, a control engineer can create classes. A class features the labels that have a similar sensitivity over time. The key to each class is a set of attributes and values from the A2L specification. For example, there can be a class of labels that has a low sensitivity. The keys to this class are the A2L properties unit = temp_cels or vehicle_speed. We measure the time-sensitivity on a scale from 0–1, with 0 meaning the value is developing steadily and 1 meaning the value is oscillating fast. This scale directly relates to the success when doing a manual evaluation of a dependency by a domain expert. Hence, it is also the result of the dependency classification criterion.

Let B be the set of classes created by control engineers prior to the analysis. Each class is characterized by properties from the A2L specification and has an associated time-sensitivity for the containing labels. The key to each class is the A2L information such that the function $class : \mathbf{D} \rightarrow B$ can identify the corresponding class of a label as part of a dependency during the analysis. The success rate is then given by the time-sensitivity of the class, i. e., by the function $dyn : B \rightarrow [0,1] \in \mathbb{R}$. Typically the classes create a staircase function in relation to the success, as shown in Figure 7.9.

The complete criterion $success_{\mathrm{DC}}$ of a communication dependency is denoted as:

$$success_{\mathrm{DC}}(q_k) = dyn(class(label(q_k))) \tag{7.9}$$

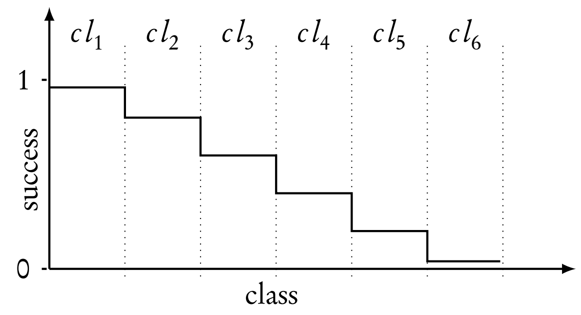


Figure 7.9: Staircase function of the relation of A2L classes to the success. The example plot shows six different classes with a success value from 0.9 to 0.1.

In future work we want to consider the classification of dependencies based on its sensitivity toward other dependencies. As mentioned in the motivation for this criterion, there is an interconnection between the inputs of a runnable and its outputs. We would investigate this interconnection to determine how the sensitivity classes are propagated through runnables and ultimately the whole graph.

7.4.2 Reaction Constraints

When relaxing a dependency, the end-to-end timing inside the system may change due to a higher degree of flexibility in the mapping of runnables. In fact, the end-to-end timing between the runnables of a dependency may increase by the period of the runnables' task. Assuming that there are event chains modeled with end-to-end timing constraints, then the process of relaxing a dependency may violate a timing constraint. Such timing constraints are typically modeled as reaction constraints. Reaction constraints are domain knowledge and specified by experts. Thus, if a domain expert evaluates the possibility to relax the timing of a dependency, the expert would check against any timing constraints of event chains. The idea of the reaction constraints criterion is to provide automatic checks against reaction constraints. The criterion is a success criterion because it can potentially eliminate a dependency from being proposed to a domain expert. If relaxing a dependency violates a reaction constraint, then this dependency should not be proposed to an expert.

Due to the determinism in the LET model of computation, the reaction latency of an event chain can be calculated statically. Note that we calculate the maximum reaction latency of an event chain. There are typically more upper bounds of reactions constraints specified in our domain than lower bounds. Hence, we check for violations of the upper bound of a reaction constraint in this criterion, but the criterion can be easily extend to support lower bounds as well. Algorithm 11 calculates the reaction latency of an event chain *ec* from the application model based on the dependency graph DG and the partitioning information *Part*. It assumes that the communication dependency q_k that has to be analyzed by the criterion is part of the event chain. If it is not, the criterion returns 1.

Algorithm 11 Calculate the reaction latency of an event chain.

```
1:  procedure CALCREACTLAT(EventChain ec, $q_k^{src,dst}$)
2:      $latency := 0$
3:      for all subeventchain tuple $(r_{src}, r_{dst})$ in ec do
4:          if $m(r_{src}) = m(r_{dst})$ then
5:              if $\exists q_k^{src,dst} : \gamma(q_k^{src,dst}) = 0..1 \vee \gamma(q_k^{src,dst}) = 1$ then
6:                  $latency \mathrel{+}= P_{m(r_{src})}$
7:              end if
8:          else
9:              $latency \mathrel{+}= P_{m(r_{src})}$
10:         end if
11:     end for
12:     $latency \mathrel{+}= P_{m(r_{dst})}$
13:     return latency
14: end procedure
```

The algorithm iterates over all tuples in the event chain and checks every tuple of runnables for dependencies and mapping (line 3). If both runnables are mapped to the same task and only one dependency between them is either strictly undelayed or relaxed, the latency is increased by the period of the source runnable's task (line 4–6). The period of the source runnable's task is also added to the latency when the runnables are mapped to different tasks (line 8–9). Lastly, the period of the task that the last runnable of the event chain is mapped to is added to the latency (line 12). It can only be ensured that the last runnable has finished execution at the end of the LET.

To evaluate $\mathit{success}_{\mathrm{RC}}$ of a dependency q_k, the criticality is tentatively set to relaxed and *calcReactLat* is calculated for the event chain that contains the runnables. In the subsequent step the resulting latency from *calcReactLat* is checked against the reaction constraint of the specified event chain. If the resulting latency is greater than the constraint, then the relaxation of the dependency q_k would violate the reaction constraint. Hence, $\mathit{success}_{\mathrm{RC}}$ would evaluate to zero as shown in the following equation.

$$\mathit{success}_{\mathrm{RC}}(q_k) = \begin{cases} 0, & \text{if } q_k \text{ violates any reaction constraint if relaxed} \\ 1, & \text{otherwise} \end{cases} \tag{7.10}$$

Unfortunately, event chains have often not been specified for legacy designs. Due to this fact, only a small amount of event chains with reaction constraints were available to us such that we have not implemented this criterion for our case studies.

7.5 Candidate Aggregation & Ranking

In the candidate aggregation & ranking phase of the Concurrency Analysis component, all pairs of runnables are evaluated by the presented criteria. For each pair of runnables that is connected by a precedence, all dependencies inbetween are evaluated by the success criteria and the precedence inbetween is evaluated by the impact criteria. After evaluating the criteria of each dependency and precedence, the results are aggregated. The overall success score of a dependency $q_k^{a,b}$ is the product of the n success criterion results.

$$\mathit{success}(q_k \in \mathbf{Q}_C^{a,b}) = \prod_{i=0}^{n} \mathit{success}_i(q_k) \tag{7.11}$$

The overall impact score of a precedence $c^{a,b}$ is the weighted sum of the m impact criterion results.

$$\mathit{impact}(c^{a,b} \in \mathbf{C}) = \sum_{j=0}^{m} w_j \times \mathit{impact}_j(c^{a,b}) \tag{7.12}$$

The impact criteria are weighted and the weights are specified beforehand but can also be changed during the workflow to control the analysis. For example, when the main objective besides the parallelization goal is to remove or improve the synchronization, then the weight of the SP criterion can be increased.

The $score(\langle r_a, r_b \rangle))$ of a pair of runnables is then the sum of the overall success score over all dependencies times the overall impact score.

$$score(\langle r_a, r_b \rangle)) = \sum_{q_k \in \mathbf{Q}_C^{a,b}} success(q_k) \times impact(c^{a,b}) \tag{7.13}$$

The calculated score and the dependencies of the pair of runnables are used to create an evaluation candidate. These evaluation candidates are inserted into a prioritized queue. The evaluation candidate $\langle r_a, r_b \rangle$ with the highest score is marked for evaluation $\sigma(\mathbf{Q}_C^{a,b}) = r$ and proposed to the domain expert as part of the Dependency Analysis component. For the current iteration, the candidate with the highest score represents the most suitable and probable candidate for an evaluation in respect to the parallelization goal.

7.6 Dependency Analysis

The Dependency Analysis component is the interface to the domain expert. During the execution of this component the domain expert evaluates if the proposed evaluation candidate allows a relaxed timing and responds with a boolean decision.

Any evaluation candidate is part of a controller with a certain functionality. When relaxing the timing of the corresponding dependencies, the communicated data can be delayed or accelerated. This change in timing may alter the timing of the controller which in turn potentially alters the functionality. Depending on the controller design, the change may violate latency or stability requirements, and other performance criteria. As a complete specification of these requirements for each single controller and their interactions is not available, a manual evaluation by an expert in this domain is mandatory.

For the evaluation, the Dependency Analysis component presents the evaluation candidate together with the criteria values to the expert. The expert receives the set of dependencies of the candidate together with the scoring. The scores and criteria values give hints why a candidate's dependency is the most relevant one among the others. This can influence how the expert evaluates a dependency. There are various techniques to evaluate the execution timing impact on the functionality of the controller such as formal verification or simulation [ZH15]. It is up to the expert which technique is the most suitable one. The presented evaluation process within this component thus follows our objective to provide a simple interface to the domain expert.

After the expert evaluated all dependencies of the evaluation candidate, a decision is made. If the decision is positive, the dependencies are set to relaxed and refined in the application model. Independent of the decision, all dependencies are marked as evaluated. This finishes a workflow iteration.

7.7 Summary

The contribution of this chapter are the two components, Concurrency Analysis and Dependency Analysis. The two components are part of the workflow to find the most beneficial and probable dependencies for evaluation by a domain expert. Dependencies and precedences are hereby analyzed using multiple criteria within the Concurrency Analysis. The impact criteria analyze a precedence between two runnables and evaluates how beneficial the relaxation of the corresponding dependencies would be toward the parallelization goal. The success criteria yields indicators if a dependency is likely to be relaxed by a domain expert. Both groups of criteria are applied to precedences and dependencies alike and aggregated to a final score. This score allows to select the most promising candidate for evaluation. The best candidate is then presented to a domain expert as part of the Concurrency Analysis component. Based on the score and the criteria details, the domain expert evaluates the corresponding dependencies and makes a decision. This decision refines the application model and advances it toward its inherent concurrency.

In future work we want to consider multiple partitioning informations. The splitting component already provides a range of splitting heuristics, each following its own objective. The resulting partitioning information from the different heuristics provides additional data that can be a benefit to the criteria.

8 Case Studies

In this chapter, we evaluate the proposed parallelization concept from the previous chapters. The evaluation was conducted in the form of case studies and applied to different real-word Engine Management System (EMS). The EMS is used in these case studies as it is one of the most complex examples of embedded real-time control software from the automotive industry. A premium-class car contains 70 to 100 Electronic Control Units (ECUs) executing approximately 100 Mio lines of code [Cha09] while an EMS executes the biggest portion. In the future the most code might be executed by the ECUs responsible for Highly Automated Driving (HAD) and autonomous driving. The case studies use different EMS from leading automotive manufacturers. Due to IP restrictions, we cannot disclose further details but the benchmark published by Kramer et al. [KZH15] characterizes these EMS to some degree.

8.1 Standalone Synchronized Splitting

In this case study we evaluate the parallelization concept specifically toward its task splitting capabilities. The concept is used as part of the late strategy in the integration phase and on the architecture level of the product development. At this point in time, all functionality is designed and implemented as well as almost all architectural decisions were made. The architecture is ready to be integrated into the system. For this case study we assume the issue that the implementation of a task's set of runnables exceeds its capacity for the specified multi-core platform. The implemented task exceeds the computational power of a single core on this platform and hence cannot be integrated. Our parallelization concept is used to split the given task such that the resulting task partitions can be integrated into the system. We hereby assume that the multi-core platform provides enough computational power in terms of number of cores. Due to being at the end of the product development lifecycle of the EMS, the costs associated to changes are high. Hence, we allow synchronization between the splitted task partitions as it enlarges the solution space to find a suitable solution at low costs.

In this case study, the path through our parallelization concept uses three components, all of which are automatic components. The components for relaxation are disabled to prevent any changes in the behavior of the application. Input for our approach is the application model of the EMS as well as the parallelization goals. We assume that the oversized heavy task t_i should be splitted into two task partitions $t_{i,0}$ and $t_{i,1}$ which is a typical request in our domain and that kind of scenario. The speedup goal for the two task partitions is set to 1.7 which is a realistic goal for real-world applications. As mentioned before, our concept is allowed to introduce synchronization points. The amount of synchronization should hereby not exceed an overall brittleness of 12 to prevent major speedup degration for Observed Worst-Case Execution Time (oWCET) overruns.

For this case study, we selected a heavy task t_i from the EMS that has 144 runnables and communicates via 3171 labels. The task t_i encapsulates the most runnables in the application. In the first step of our concept, the dependency graph of the task t_i is extracted from the legacy application model. The extracted dependency graph contains 496 dependencies in total. There are 477 communication dependencies with 296 forward, 110 backward and 71 loop dependencies. The Dependency Graph Extraction component also found multi-writers that lead to 19 functional dependencies. Besides the functional dependencies due to multi-writers there were no other functional dependencies extracted from the application model for this task. Based on the dependency graph, the Splitting component generates the precedence graph needed to perform task splitting. The component generated 425 precedences for the precedence graph of task t_i.

According to the given goals, the Splitting component performs the task splitting in the synchronized mode. For evaluation purposes, we applied all available heuristics of this mode to task t_i. Table 8.1 show the results of the heuristics earliest possible execution (EPE), min-distance (MinD), max-parents (MaxP) and independent runnable allocation (I). When independent runnables are handled separately, this is denoted as "+ I". When a tie-breaker is used that is different from Section 6.3.2, this is indicated in brackets. E. g., "MaxP(MinD)" is using the tie-breaker MinD instead of EPE. We have also implemented and evaluated RunPar [PKQn+14] to compare our approach with related work. The table contains the number of synchronization points, the speedup and the brittleness of the splitting solutions.

The best speedup of 1.75 is achieved by the MaxP+I heuristic with 11 synchronization points and with the third best brittleness of 8.267. It is closely followed by RunPar [PKQn+14] with a speedup of 1.71 and 13 synchronization points, but with a significantly worse overall brittleness of 10.796. This is expected as RunPar assumes safe and tight bounds for Worst-Case Execution Times (WCETs). It does not focus on possible overruns which could result in a degraded speedup. In terms of robustness against oWCET overruns, the MaxP+MinD+I heuristic provides the best solution with the lowest brittleness of 6.628 and the lowest number of synchronization points at a very competitive speedup of 1.66.

Table 8.1: Results of the splitted task from the EMS.

Heuristic	SyncPoints	Speedup	Brittleness
EPE	22	1.65	18.859
MinD	9	1.64	12.056
MaxP	9	1.39	7.020
MaxP+I	11	1.75	8.267
MaxP(MinD)+I	8	1.66	6.628
RunPar [PKQn+14]	13	1.71	10.796

The developer can now do focused tradeoffs based on the speedup, number of synchronization points and brittleness. To assist the developers decision, the results are illustrated in a point map as shown in Figure 8.1. In our case study, the pareto-optimal solutions considering speedup and brittleness are MaxP+I and MaxP+MinD+I. Experiments with other tasks confirm the general trend that the proposed heuristics achieve similar speedups but with a significantly improved robustness against oWCET overruns compared to related work.

We chose the pareto-optimal solution MaxP+I for evaluation in a vehicle. Based on this solution an OS configuration was created containing the two parallel task partitions $t_{i,0}$, $t_{i,1}$ and also synchronization code using our lean implementation. Further following the toolchain described in Section 5.1, a build is created using the artifacts generated by our concept. The resulting build of the ECU software was deployed in a prototype vehicle and put into field test.

We observed the following from the evaluation of the prototype vehicle. First of all, we could validate through functional tests that there was no observable change in functionality compared to the original legacy application with the unsplitted task t_i. This is a strong hint that all dependencies are indeed maintained and mandatory functional dependencies were not left out. Next, we measured the execution time and hence the waiting time at the synchronization points. By measuring the actual waiting times at the synchronization points we can evaluate and compare the statically determined values with the runtime values. Figure 8.2 shows the measured average waiting times in comparison with the statically determined waiting times and the brittleness for each of the 11 synchronization points. The figure shows, that for all synchronization points with a brittleness strictly less than one (points $1, 2, 7, 9$), the waiting time at the synchronization point is on average 0.8 μs. This time interval represents the operating system and scheduling overhead indicated by the dashed line. At these synchronization points there is no busy waiting as the barrier event is already set. This shows that the brittleness of a synchronization point is a suitable indicator for its robustness. Furthermore, it can be seen in Figure 8.2, that the statically determined waiting times at the synchroniza-

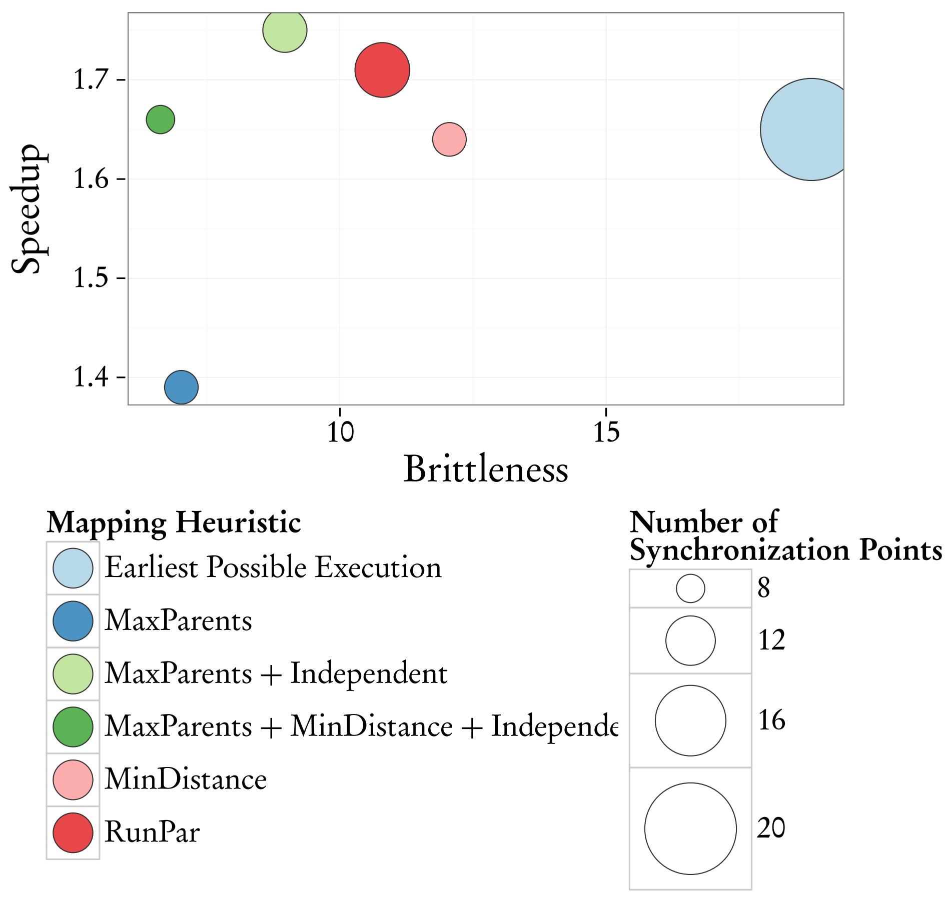


Figure 8.1: Results of the splitted task t_i from the EMS.

tion points match the actual average waiting times quite well. The statically determined waiting time is the time between the start of execution of the receiving runnable of a synchronization point to the finish time of its predecessor. Exceptions are the synchronization points 3, 4, and 5 where the actual waiting times are longer due to preemptions of the source task by higher priority tasks. The opposite effect with a shorter waiting time of these preemptions are seen for the synchronization point 11. When accounting for the preemptions observed in the execution traces, the adjusted waiting times show the same deviation from the computed waiting times as for the other synchronization points. Taking these preemptions into account, the splitting algorithm could be used to minimize waiting times. However, this remains future work.

Overall, we could observe that the actual makespan observed in the worst-case is only 0.5 % below the calculated oWCET makespan. The MaxP+I solution is thus very robust against varying execution times such as oWCET overruns.

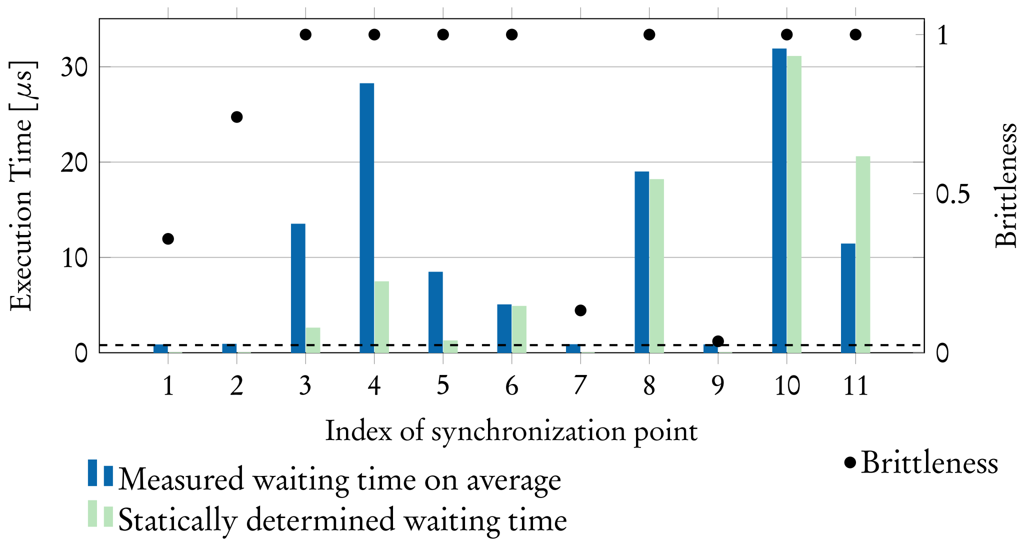


Figure 8.2: Measured and statically determined waiting time compared to the brittleness of each synchronization point.

8.2 Relaxing & Unsynchronized Splitting

In this case study we evaluate the parallelization concept toward its relaxation and unsynchronized task splitting capabilities. It is used as part of the early strategy where the product development lifecycle is still in its design phase. On the architecture level of the design, functionality will be changed while the application is further refined. The motivation for this case study is that architecture decisions regarding the concurrency can be evaluated early in the development lifecycle. Our parallelization concept facilitates the process of evaluating communication timing decisions for domain experts while designing the application. It is used to evaluate the impact of a change in a communication's timing quickly. The domain expert can iteratively simulate timing decisions on the application model without leaving the lifecycle phase. In this case study we want to specifically evaluate the unsynchronized splitting mode to create fully decoupled task partitions.

The workflow path through our parallelization concept uses all components in this case study. Input for our approach are the parallelization goals as well as the application model of the EMS. As the application model is still under development, there is no implementation of the current design and also no runtime information. However, previous development lifecycles provide such information in the form of a reference architecture. This reference architecture completes the application model with needed oWCET information. We assume that we want to evaluate the concurrency of a task t_e. Ideally, it should be provide two fully decoupled task partitions $t_{e,0}$ and $t_{e,1}$. The speedup goal for the two task partitions is set to 1.9 to provide a variability of 10 % when balancing the tasks.

For this case study we have selected a heavy task t_e from a different customer project than in the case study in Section 8.1. The dependency graph of the selected task t_e was extracted from the application model. The resulting dependency graph contains 234 runnables, 248 communication dependencies and no functional dependencies. Based on the dependency graph, the Splitting component generates the precedence graph. The generated precedence graph with 234 runnables has 113 precedences. The graph is shown in Figure 8.3a. It shows the runnables as nodes, excluding the runnables that are not connected via a precedence for illustration reasons. It shows the initial distribution of connected components in the graph. The largest component contains 65 % of the task's oWCET and is thus the main focus for this case study. All other components are smaller than 4.9 %.

Due to the complexity of this EMS, we would need to involve many different experts to evaluate the functional impact on each controller. To efficiently assess our approach in this case study, we therefore simulate the experts based on the Dependency Classification success criterion. For each proposed dependency, the simulated expert decides randomly with the probability of $success_{DC}$ if the dependency can be relaxed. This follows our argument that the criterion $success_{DC}$ indicates how the expert would decide. In the future we plan to test our approach with real-world domain experts and compare the results.

To assess the benefit of our approach, the key metric is the number of expert-in-the-loop workflow iterations. As our objective is to reduce interactions with domain experts at all times because an expert is a scarce resource, the number of iterations reflects the efficiency of our concept. Less iterations mean less involvement of the experts and thus a higher efficiency. We assume that it takes a fixed amount of work for the expert to evaluate a proposed dependency. Hence, the number of iterations times the fixed amount of work becomes the metric for the evaluation effort. For comparison, we executed the same workflow but selected dependencies for evaluation at random. Both approaches were repeated 100 times.

The results in Figure 8.3 show one execution of how our parallelization concept splitted the task t_i. In Figure 8.3b the parallelization goal was reached after 9 iterations using our simulated expert. The largest component with initially 65 % of the task's oWCET could be reduced to below 50 % which creates the first task partition $t_{e,0}$. Hence, the Concurrency Analysis could find dependencies that were successfully relaxed such that additional components were created. These additional components that were split of the component with 65 % form the second task partition $t_{e,1}$. Note that 126 components with a size of one runnable are left out for illustration reasons.

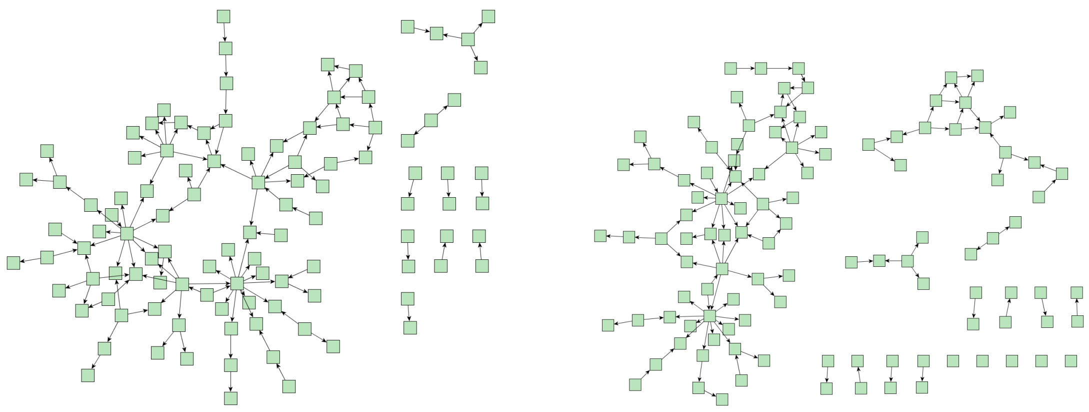

(a) Iteration #0 **(b)** Iteration #9

Figure 8.3: Task t_e in comparison from iteration #0 to iteration #9.

In Figure 8.4 the results of our case study are visualized in a boxplot. It shows the number of iterations needed to reach the given parallelization goal for our Concurrency Analysis in comparison to a random selection. On average, after 9.4 iterations, the Concurrency Analysis has found suitable dependencies for relaxation that satisfies the parallelization goal. Even though the simulated expert decides against the relaxation of a proposed dependency in some iterations, the error is very small. Except for four outliers that need 11, 13, 20 and 24 iterations, all workflows finished after 8–9 iterations. In comparison, when dependencies were selected randomly, approximately 74.5 iterations were needed on average to reach the parallelization goal. The upper quartile and lower quartile are 99.0 respectively 55.25 iterations and their were no outliers. Comparing the average of both approaches, our Concurrency Analysis needs only 12.6 % of the iterations compared to selecting dependencies randomly for evaluation. This shows that our approach successfully guides the selection of dependencies for evaluation and significantly reduces the number of iterations and thus the evaluation effort for reaching the parallelization goal.

8.3 Relaxing & Synchronized Splitting

This case study serves the purpose to evaluate how the parallelization concept can decrease the brittleness when task splitting. A task that would contain synchronization points when splitting can potentially be splitted into fully decoupled task partitions by relaxing specific dependencies. In this case study we specifically target dependencies that lead to synchronization points. Synchronization points increase the brittleness for

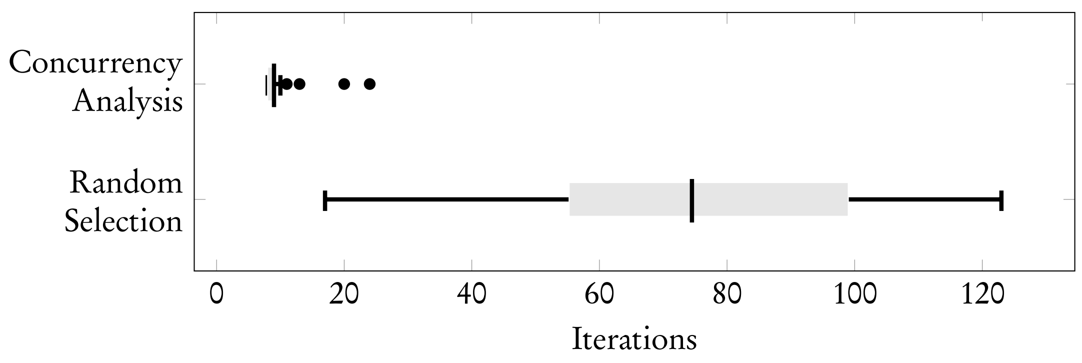


Figure 8.4: Number of iterations needed to reach the parallelization goal with our Concurrency Analysis in comparison to a random selection. The decision to relax the proposed dependency is made by a simulated domain expert.

makespan penalties at oWCET overruns. Any synchronization between task partitions prohibits them of being truly decoupled Logical Execution Time (LET) tasks. However, the dependencies that lead to synchronization can be relaxed using our Concurrency Analysis and the domain expert. The parallelization concept is used as part of the early strategy. In this strategy, the product development lifecycle is still in its design phase. The design phase allows and promotes changes in the functionality. Changes such as the relaxation of dependencies can be made at low costs in this phase.

All components of our parallelization concept are in use during this case study. Input for our approach are the parallelization goals as well as the application model of the EMS. As the application model is still under development, the needed oWCET information for the application model is retrieved from previous development lifecycles. We require that the task t_i should be splitted into two task partitions $t_{i,0}$ and $t_{i,1}$. Ideally, the splitting should provide two fully decoupled task partitions $t_{i,0}$ and $t_{i,1}$ without any synchronization. The amount of synchronization should be zero as well as the overall brittleness. To enforce this goal, the speedup to achieve for the task is set to 2.0. This demands perfectly balanced task partitions as it allows no variability. Such a goal can typically never be reached and thus the workflow will never exit. We stop the workflow manually as soon as the solutions converge and remain unchanged over more than two iterations.

For this case study we chose a relatively small task with 21 runnables from the EMS. It is a task that is executed with a high frequency and is mainly responsible for parts of the exhaust management. The precedence graph of this task is shown in Figure 8.5. We chose this task because of two reasons. First, the task's architecture contains a low level of available concurrency. Hence, splitting it always leads to synchronization independent of the used splitting heuristic. Second, the task is less complex than the tasks of the other case studies. This reduces the overall runtime of the task splitting due to the additional exhaustive search in conjunction with the task splitting to illustrate the benefit of our approach.

The exhaustive search is needed because the heuristics presented in Section 6.3 to split a task may find a sub-optimal solution. We use these heuristics because our objective is scalability such that we cannot provide optimal solutions due to the complexity of real-world automotive applications. Due to providing sub-optimal splitting solutions, the solutions potentially differ in mapping, order and synchronization on every iteration of our workflow. Ideally, one or more dependencies are relaxed in every iteration. As soon as a dependency is relaxed, the concurrency of the application model is changed. While it is a minor change in the precedence graph of the application, this change has a dramatic effect on the resulting splitting solutions. Thus, to demonstrate the benefit of the concurrency analysis for synchronization points, we perform an exhaustive search to find the optimal solution for the splitting. Still, the exhaustive search is only used for demonstration purposes in this case study. Due to the two metrics used for evaluation, speedup and brittleness, there is more than one optimal solution. In fact, there is a pareto-optimal set of solutions.

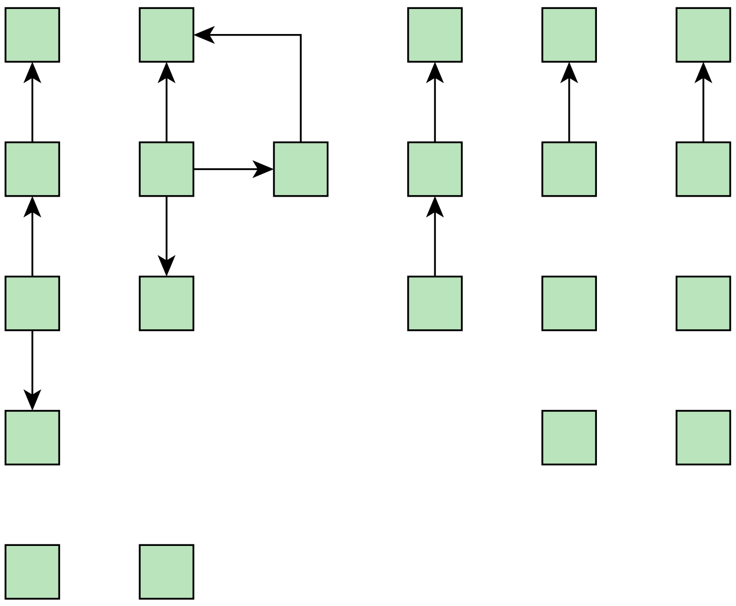


Figure 8.5: Precedence graph of the task used in this case study.

To perform the exhaustive search we use an approach similar to Genetic Algorithms (GEs), but without the population and operators. The encoding of the problem in genes of an individual should represent all possible solutions of the search space and also be unique. The obvious encoding method for our mapping problem is to encode the order of execution for each task partition into an individual. This idea was already published by [HAR94] and [CFR96]. Although [CFR96] is using a precedence graph to ensure that only valid individuals are created, the genetic operators become complex as they have to ensure the correctness when generating new individuals.

In our approach we encode the mapping of a runnable to a task partition, independent of the order of execution and precedence constraints. Given the set of runnables of a task $\mathbf{R}_{t_i} = \{r_1, r_2, \ldots, r_n\}$, the gene of an individual encodes the mapping of $\mathbf{R}_{t_i}$ to the task partitions $t_{i,1}, t_{i,2}, \ldots, t_{i,k}$. The gene is composed of the task partition that a runnable is mapped to, i. e., the individual $s = \{m(r_1), m(r_2), \ldots, m(r_n)\}$. For example, to encode that r_1 is assigned to $t_{i,2}$ and r_2 is assigned to $t_{i,1}$, the gene of the individual is specified as $s = \{t_{i,2}, t_{i,1}\}$. The encoding only specifies the mapping and not the order of execution inside a task partition. The order is later determined by a scheduler. The scheduler hereby ensures that all precedences are met and the runnables are mapped to the task partition specified by the individual. The benefit of this gene encoding is that there are no invalid individuals and the mapping of an individual to a solution is straightforward. The only drawback we know of is that not all solutions in the search space can be represented by an individual. As the order of execution is determined by the scheduler, solutions that e. g., delay the activation of a runnable are left out. However, we haven't seen any delayed activation in our application models.

To find all pareto-optimal solutions we evaluate every possible solution that can be represented using this encoding scheme. For this case study we use HLFET as scheduler for the encoded solutions. Note that other schedulers for precedence graphs could be used as well. For a task with 21 runnables that has to be split into two task partitions, the encoded solution space consists of $n^k = 2^{21}$ possible solutions. Note that there are duplicates within these 2 Mio solutions. Two solutions in the encoded solution space represent one solution to the original problem. For example, let r_1, r_2 be two runnables that are mapped to two task partitions. This can be encoded in two different ways: $s = \{m(r_1) = t_{i,0}, m(r_2) = t_{i,1}\}$ and $s = \{m(r_1) = t_{i,1}, m(r_2) = t_{i,0}\}$. The duplicates are of use to an GE but for our exhaustive search it is of no benefit. Therefore, we removed the duplicates.

Furthermore, we configured the weights for the criteria in the Concurrency Analysis such that the synchronization point criterion dominates. All success criteria are disabled and the weight for the synchronization point criterion is set to one thousand times the weight of the forward evaluation criterion. By setting the weights as described we can break potential ties using the FE criterion. For simplicity reasons, we assume that all dependencies that are proposed to the domain expert are relaxed.

Figure 8.6 shows the solutions on the pareto-front between the speedup and the brittleness. Each solution represents a splitting solution using the exhaustive search and on each iteration a relaxation took place. The single-core solutions with a speedup of 1.0, no synchronization and hence a brittleness of zero are left out. Our focus in this figure is on the solutions with a high speedup and low brittleness. In this range, the pareto-front is changing due to the relaxation of dependencies. We can see that the pareto-optimal solutions do not or only marginally improve in speedup due to removal of idle-time in the overall makespan. The brittleness is however drastically reduced due to the relaxation of dependencies related to synchronization points. The pareto-fronts show a clear trend: The fronts are developing toward a solutions where there is no synchronization without degrading the speedup. After 10 iterations, the two pareto-optimal solutions are: 1.999957 with no synchronization and 1.999935 with a very low brittleness of 0.11. After another iteration, there is no synchronization point left with a speedup of 1.999935. This solution cannot be optimized any further.

Hence, in this case study we have shown that our parallelization concept can decrease the brittleness when task splitting with little effort. By proposing the dependencies that relate to synchronization points for relaxation, our concept can efficiently guide an expert to fully decoupled task partitions.

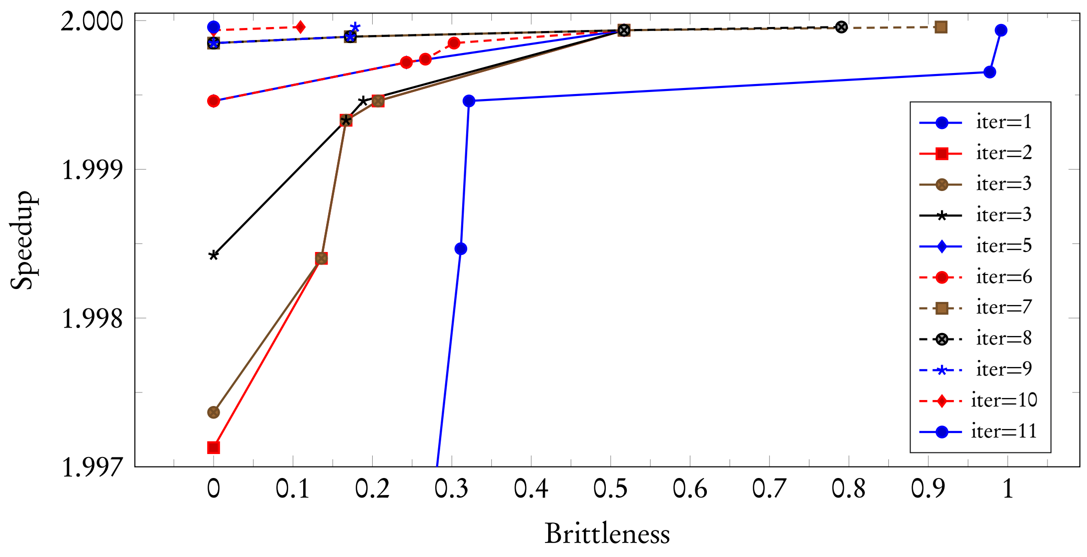


Figure 8.6: Development of the pareto-front of the splitting solutions when relaxing dependencies that lead to synchronization points.

9 Conclusion & Future Work

In this final chapter, we summarize and review the central results of this thesis according to the problem statement and objectives defined in the beginning. We finish this chapter by outlining and discussing future research directions.

9.1 Summary

In this thesis, we have presented a concept to parallelize legacy real-time tasks from automotive control applications on multi-core platforms. The proposed model-based concept offers automatic workflows to parallelize a single task from legacy Electronic Control Unit (ECU) applications that can optionally be supported by a domain expert. It can be integrated seamlessly into existing automotive toolchains using the well-established AMALTHEA platform. The provided parallelization concept can be used in typical phases of the automotive development lifecycle. The concept can evaluate early design decisions toward parallelism prior to the implementation as well as parallelizing already implemented tasks with reduced effort during the late integration phase.

When parallelizing, the concept can either completely maintain the behavior of the legacy task or maintain the functional correctness while altering the behavior. The legacy real-time design of the tasks is hereby the greatest limiting factor to the parallelization. Established construction principles for the design of tasks on single-core platforms have introduced a huge amount of dependencies. These dependencies limit the task's concurrency that can be exploited for parallelism.

Our parallelization concept can efficiently extract the concurrency available inside a task. Based on the extracted concurrency, a task is split into multiple parallel task partitions. The resulting task partitions can either be fully decoupled in terms of Logical Execution Time (LET) or may allow synchronization with other task partitions. The presented splitting algorithms for this challenge are highly scalable and can cope with tasks from real-world applications. Splitting a task is performed by a combination of a list-scheduling algorithm and multiple mapping heuristics that are optimized for speedup and robustness. The robustness is important because overruns of observed Worst-Case Execution Times (WCETs) in combination with synchronization can result in a speedup degradation of the splitted task at runtime. It is the current state of practice and lack of predictable hardware in the automotive domain such that only observed boundaries on the WCET are available to us. Our approach can cope with these domain specific aspects by providing a robustness metric that indicates the likelihood that Observed Worst-Case Execution Time (oWCET) overruns lead to a speedup degradation. To minimize the overhead of potential synchronization, we also provide an algorithm to reduce the amount of synchronization together with a lean implementation for OSEK basic tasks.

The limited available concurrency inside a legacy task originates from the single-core design process but is not always mandatory. To further improve the concurrency for parallelization, the proposed concept can increase the concurrency by altering the behavior but still maintain correctness. The behavior is altered through the relaxation of suitable dependencies inside the task. A relaxation is the verified removal of timing constraints of a dependency. To find these suitable dependencies, our analysis performs an efficient ranking of all dependencies of a task. It evaluates the impact of a relaxation toward the parallelization goal as well as the likelihood of success that the relaxation of a dependency is approved by the domain expert. The evaluation is performed using an extensible set of criteria that are applied to each dependency and precedence inside a task. We have presented two impact criteria: (1) The forward evaluation determines the benefit in terms of concurrency. (2) The synchronization point criterion measures the change in robustness against oWCET overruns. Furthermore we have proposed two success criteria: (1) The dependency classification determines the dynamics of the communicated data to classify how a relaxation may impact the behavior. (2) The reaction constraints criterion checks if a relaxation may violate specified reaction constraints of event chains. The results from these criteria are aggregated and create a ranking to find the most suitable dependencies for relaxation. The most suitable dependencies are then presented to a domain expert for validation of the altered behavior. To validate that the behavior is still within its specifications and correct, we provide an interface with a simple yes-no question to reduce the effort for the domain expert. After the validation, the application model is refined accordingly and reflects the improved concurrency of a task. Following the iterative nature of our concept, this improved concurrency can again support task splitting to achieve the specified parallelization goal.

In this thesis, we have evaluated the presented parallelization concept with real-world case studies from different automotive engine management systems. We have shown the task splitting capabilities of our concept for a heavy task that encapsulates the most runnables in the application with 144 runnables and 496 dependencies. In this case study, our concept was used during late development lifecycle phases to solve the issue that the implemented task exceeds the computational power of a single core. We could show that the task splitting successfully exploits the concurrency of the given legacy task and provides results comparable to other approaches but with improved robustness against oWCET overruns. In order to validate the solution, the resulting splitted task was integrated into a prototype vehicle that successfully drove on a proving ground. In another case study we have shown that our workflow to find and relax suitable dependencies facilitates the evaluation of the concurrency during early development lifecycle phases with little effort. For this purpose we also selected one of the biggest tasks with 234 runnables and 248 dependencies from an Engine Management System (EMS). Our concurrency analysis was able to reach the parallelization goal of two balanced and unsynchronized task partitions at an average of 12.6 % of the iterations compared to selecting dependencies randomly for evaluation. The target of the third case study have been tasks with a low level of concurrency and a high degree of synchronization as a result of the splitting. We have shown that these tasks can be freed from synchronization by relaxation with little effort.

9.2 Future Work

During our research, we have identified several interesting open questions and opportunities that are worth further investigation. In the following, we outline research directions that extends our presented parallelization concept.

Preemption and Synchronization

Preemptive schedulers are commonly used in the automotive industry for both single- and multi-core platforms. Many automotive multi-core platforms currently have less cores than tasks such that many tasks have to share a core. Hence, when tasks are sharing a core, preemptions are very likely. While the preemption do not interfere with a splitted and fully decoupled task, it has an effect on a splitted and synchronized task. Preemptions do not invalidate the splitting solution but it has an impact on the overall task performance. We have seen this impact during the case study in Section 8.1 that one of the two splitted tasks was interrupted by another task with a higher priority. The splitting solution contained synchronization such that the non-preempted task had to wait additionally for the preempted task at the next synchronization point. Therefore, the overall task performance decreased due to the additional waiting time. To detect these scenarios, an extension to the synchronized splitting could be to integrate the list of tasks that can potentially preempt the splitted and synchronized task. This extension could improve the splitting toward robustness.

Probabilistic Worst-Case Execution Times

Due to the current state of practice in the automotive industry, there are typically no exact worst-case execution times available. However, besides the observed WCET, also the deviation and the number of times a runnable over- or underruns this bound can be measured. Based on this information, the distribution of the observed execution times could be calculated. In future work we would consider including this distribution into our metric to measure the brittleness of synchronization points. The brittleness metric would improve its accuracy in determining when overruns might happen.

Splitting with Genetic Algorithms

The splitting heuristics we have presented are scalable and can cope with tasks from real-world applications. The drawback to this scalability is that the splitting solutions are not optimal. On the other hand, algorithms to find optimal solutions for the task splitting problem cannot complete in reasonable time due to the complexity of real-world automotive applications such as an EMS. One possibility to improve the task splitting without losing the scalability is to incorporate Genetic Algorithms (GEs). In particular, we would use our heuristics to create the initial population for the GE and further optimize the task splitting solutions toward robustness or speedup.

Bibliography

[AEF$^+$14] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4):82:1–82:37, 2014. doi:10.1145/2560033. Cited on page 1.

[Aoy12] M. Aoyama. Computing for the next-generation automobile. *Computer*, 45(6):32–37, June 2012. doi:10.1109/MC.2012.153. Cited on page 1.

[ASA] ASAM e. V. Association for Standardisation of Automation and Measuring Systems. Retrieved May 10, 2018. URL: https://www.asam.net/. Cited on page 29.

[ASA15a] ASAM e. V. ASAM MCD-2 MC V1.7.0, February 2015. Retrieved May 10, 2018. URL: http://www.asam.net/nc/home/standards/standard-detail.html?tx_rbwbmasamstandards_pi1[showUid]=3078. Cited on pages 31 and 109.

[ASA15b] ASAM e. V. ASAM MDX V1.2.0, June 2015. Retrieved May 10, 2018. URL: https://wiki.asam.net/display/STANDARDS/ASAM+MDX. Cited on pages 29 and 146.

[AUT] AUTOSAR. AUTomotive Open System ARchitecture. Retrieved May 10, 2018. URL: http://www.autosar.org. Cited on pages 28, 29, 31, 32, and 146.

[Bar09] Michael Barr. Real men program in C. Embedded Systems Design. TechInsights, 2009. Retrieved May 10, 2018. URL: http://www.embedded.com/electronics-blogs/barr-code/4027479/Real-men-program-in-C. Cited on page 7.

[BDN$^+$15] M. Becker, D. Dasari, V. Nélis, M. Behnam, L. M. Pinho, and T. Nolte. Investigation on AUTOSAR-Compliant Solutions for

Many-Core Architectures. In *2015 Euromicro Conference on Digital System Design*, pages 95–103, 2015. doi:10.1109/DSD.2015.63. Cited on pages 18 and 34.

[BEGL05] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static WCET analysis to automotive communication software. In *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 249–258, 2005. doi:10.1109/ECRTS.2005.7. Cited on page 12.

[BKK$^+$13] Manfred Broy, Sascha Kirstan, Helmut Krcmar, Bernhard Schätz, and Jens Zimmermann. What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310, 2013. Cited on page 26.

[BKPS07] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95(2):356–373, Feb 2007. doi:10.1109/JPROC.2006.888386. Cited on page 1.

[BP03] H. Bauer and W. Post. *Conventional and Electronic Braking Systems*. Technical instruction. Robert Bosch GmbH, 2003. Cited on page 16.

[Bro05] M. Broy. Automotive software and systems engineering. In *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, pages 143–149, 2005. doi:10.1109/MEMCOD.2005.1487905. Cited on page 7.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*, volume 24 of *Real-Time Systems Series*. Springer, 2011. doi:10.1007/978-1-4614-0676-1. Cited on pages 13 and 14.

[But12] D. Buttle. Real-time in the prime-time. Keynote on ECRTS, 2012. URL: https://www.ecrts.org/fileadmin/user_media/ecrts12/ECRTS12-Keynote-Buttle.pdf. Cited on pages 1 and 9.

[C$^+$09] TIMMO Consortium et al. TADL: Timing Augmented Description Language version 2. *TIMMO (TIMing MOdel), Deliverable*, 2009. Cited on page 17.

[Cac07] Dan G. Cacuci. *Sensitivity and Uncertainty Analysis: Theory, Volume I: 1*. CRC Press, 2007. Cited on page 108.

[CCS+08] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An integrated framework for MPSoC application parallelization. In *Proc. of the 45th ACM/IEEE Conf. Design Automation Conference (DAC)*, pages 754–759, 2008. Cited on pages 38 and 40.

[CENM13] D. Cordes, M. Engel, O. Neugebauer, and P. Marwedel. Automatic extraction of multi-objective aware parallelism for heterogeneous MPSoCs. In *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pages 1–10, Sept 2013. `doi:10.1109/MuCoCoS.2013.6633599`. Cited on pages 34 and 40.

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. *ERTS*, 2010. Cited on page 19.

[CFR96] R. C. Correa, A. Ferreira, and P. Rebreyend. Integrating list heuristics into genetic algorithms for multiprocessor scheduling. In *Parallel and Distributed Processing, 1996., Eighth IEEE Symposium on*, pages 462–469, 1996. `doi:10.1109/SPDP.1996.570369`. Cited on page 124.

[Cha09] Robert N Charette. This car runs on code. *IEEE Spectrum*, 46(3):3, 2009. Retrieved May 10, 2018. URL: `http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code`. Cited on pages 8, 9, 26, and 115.

[CM12] Daniel Cordes and Peter Marwedel. Multi-objective aware extraction of task-level parallelism using genetic algorithms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 394–399, San Jose, CA, USA, 2012. EDA Consortium. `doi:10.1109/DATE.2012.6176503`. Cited on pages 34 and 40.

[cmm] CMMI Capability Maturity Model Integration. Retrieved May 10, 2018. URL: `www.sei.cmu.edu/cmmi`. Cited on page 28.

[CMM10] D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *Proc. IEEE/ACM/IFIP Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 267–276, 2010. doi:10.1145/1878961.1879009. Cited on pages 33, 34, 37, and 40.

[CNEM13] Daniel Cordes, Olaf Neugebauer, Michael Engel, and Peter Marwedel. Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 950–959. IEEE Computer Society, 2013. doi:10.1109/ICPP.2013.113. Cited on pages 34 and 40.

[CSG99] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999. Cited on pages 33, 37, and 46.

[Don14] John Donovan. What Engineers Need to Know When Selecting an Automotive-Qualified MCU for Vehicle Applications, 2014. Retrieved May 10, 2018. URL: https://www.digikey.com/en/articles/techzone/2014/jul/what-engineers-need-to-know-when-selecting-an-automotive-qualified-mcu-for-vehicle-applications. Cited on pages 8 and 145.

[ELS93] Peter Eades, Xuemin Lin, and W.F. Smyth. A Fast and Effective Heuristic for the Feedback Arc Set Problem. *Information Processing Letters*, 47(6):319–323, 1993. doi:10.1016/0020-0190(93)90079-O. Cited on page 76.

[FLN13] H. R. Faragardi, B. Lisper, and T. Nolte. Towards a communication-efficient mapping of AUTOSAR runnables on multi-cores. In *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–5, 2013. doi:10.1109/ETFA.2013.6648168. Cited on pages 35 and 40.

[FLSN14a] Hamid Reza Faragardi, Björn Lisper, Kristian Sandström, and Thomas Nolte. A communication-aware solution framework for mapping AUTOSAR runnables on multi-core systems. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–9, 2014. doi:10.1109/ETFA.2014.7005244. Cited on pages 35 and 40.

[FLSN14b] Hamid Reza Faragardi, Björn Lisper, Kristian Sandström, and Thomas Nolte. An Efficient Scheduling of AUTOSAR Runnables to Minimize Communication Cost in Multi-core Systems. *International Symposium on Telecommunications (IST)*, 7, 2014. Cited on pages 35 and 40.

[Fos95] Ian T. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995. Cited on page 37.

[FRNJ08] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 2008. Cited on page 17.

[Fü10] Simon Fürst. Challenges in the Design of Automotive Software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 256–258, 3001 Leuven, Belgium, 2010. European Design and Automation Association. `doi:10.1109/DATE.2010.5457201`. Cited on page 26.

[GE07] J. Gustafsson and A. Ermedahl. Experiences from Applying WCET Analysis in Industrial Settings. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, pages 382–392, 2007. `doi:10.1109/ISORC.2007.36`. Cited on page 12.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. Cited on pages 33 and 76.

[GKU$^+$12] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat. Time analysable synchronisation techniques for parallelised hard real-time applications. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 671–676, 2012. `doi:10.1109/DATE.2012.6176555`. Cited on page 92.

[GP94] Milind Girkar and ConstantineD. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, 1994. `doi:10.1007/BF02577777`. Cited on page 34.

[GSVK+06] Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A Hierarchical Coordination Language for Interacting Real-time Tasks. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 132–141, New York, NY, USA, 2006. ACM. doi:10.1145/1176887.1176907. Cited on page 21.

[HAR94] E. S. H. Hou, N. Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, 1994. doi:10.1109/71.265940. Cited on page 124.

[HHK01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software*, pages 166–184, London, UK, 2001. Springer-Verlag. doi:10.1007/3-540-45449-7_12. Cited on pages 1 and 20.

[HJ08] K. A. Hawick and H. A. James. Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In *Proc. 2008 Int. Conf. on Foundations of Computer Science (FCS'08)*, pages 14–20, Las Vegas, USA, 2008. CSREA. Cited on page 76.

[HKI15] Robert Höttger, Lukas Krawczyk, and Burkhard Igel. Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems. *International Journal of Computer, Control, Quantum and Information Engineering*, 9(1):268–274, 2015. Cited on pages 34 and 40.

[HT71] John E. Hopcroft and Robert E Tarjan. Efficient algorithms for graph manipulation. Technical report, Stanford, CA, USA, 1971. Cited on page 104.

[Hu61] T. C. Hu. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9(6):841–848, 1961. arXiv:http://dx.doi.org/10.1287/opre.9.6.841, doi:10.1287/opre.9.6.841. Cited on page 85.

[HvHM+16] Julien Hennig, Hermann von Hasseln, Hassan Mohammad, Stefan Resmerita, Stefan Lukesch, and Andreas Naderlinger. Poster abstract: Towards parallelizing legacy embedded control software using the LET programming paradigm. In *2016 IEEE Real-Time and*

Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016, page 51. IEEE Computer Society, 2016. `doi:10.1109/RTAS.2016.7461355`. Cited on pages 36 and 40.

[Int11] International Standard Organization (ISO). Road vehicles – functional safety – Part 5: Product development at the hardware level, 2011. Cited on page 28.

[ITE] ITEA2 Project: AMALTHEA (ITEA2-09013). Amalthea - model-based, open source development environment for automotive multi-core systems. Retrieved May 10, 2018. URL: `http://www.itea2.org/project/index/view/?project=10015`. Cited on pages 10, 26, 27, 34, 56, and 146.

[JDU+74] David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974. `doi:10.1137/0203025`. Cited on page 79.

[JFG+14] Ralf Jahr, Martin Frieb, Mike Gerdes, Theo Ungerer, Andreas Hugl, and Hans Regler. Paving the Way for Multi-cores in Industrial Hard Real-time Control Applications. *IEEE International Symposium on Industrial Embedded Systems (SIES)*, 9, 2014. Cited on pages 37 and 40.

[JFGU14] Ralf Jahr, Martin Frieb, Mike Gerdes, and Theo Ungerer. Model-based Parallelization and Optimization of an Industrial Control Code. *Tagungsband des Dagstuhl-Workshops*, page 63, 2014. Cited on pages 37 and 40.

[JGU13a] Ralf Jahr, Mike Gerdes, and Theo Ungerer. A Pattern-supported Parallelization Approach. In *Proceedings of the Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM, pages 53–62, New York, NY, USA, 2013. ACM. `doi:10.1145/2442992.2442998`. Cited on pages 37 and 40.

[JGU13b] Ralf Jahr, Mike Gerdes, and Theo Ungerer. On Efficient and Effective Model-based Parallelization of Hard Real-Time Applications. In *Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*, pages 50–59, 2013. Cited on pages 37 and 40.

[Joh74] David S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272 – 314, 1974. doi:10.1016/S0022-0000(74)80026-7. Cited on page 87.

[Joh75] D. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975. arXiv:http://dx.doi.org/10.1137/0204007, doi:10.1137/0204007. Cited on pages 75 and 76.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999. doi:10.1145/344588.344618. Cited on page 33.

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. doi:10.1007/978-1-4684-2001-2_9. Cited on page 33.

[KKuBdBfIidB98] Bundesministerium des Innern KBSt Koordinierungs-und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung. *Entwicklungsstandard für IT-Systeme des Bundes. Vorgehensmodell (V-Modell).: Teil 2: Behördenspezifische Ergänzungen. Teil 3: Handbuchsammlung. Stand: Juni 1997.* Beilage zum Bundesanzeiger. Bundesanzeiger Verlagsges., 1998. Cited on page 24.

[KQnBS15] Sebastian Kehr, Eduardo Quiñones, Bert Böddeker, and Günter Schäfer. Parallel execution of autosar legacy applications on multicore ecus with timed implicit communication. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 42:1–42:6, New York, NY, USA, 2015. ACM. doi:10.1145/2744769.2744889. Cited on page 21.

[KZH15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmark for free. *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015. Cited on page 115.

[LLP$^+$09] R. Long, H. Li, W. Peng, Y. Zhang, and M. Zhao. An Approach to Optimize Intra-ECU Communication Based on Mapping of AUTOSAR Runnable Entities. In *2009 International Conference on Embedded Software and Systems*, pages 138–143, 2009. doi:10.1109/ICESS.2009.63. Cited on page 34.

[Lun16] Jan Lunze. *Regelungstechnik 1: Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen (Springer-Lehrbuch) (German Edition)*. Springer Vieweg, 11 edition, 2016. This is a german book. Cited on pages 9, 11, and 145.

[Mar10] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2nd ed. 2011 edition, 11 2010. URL: `http://amazon.com/o/ASIN/9400702566/`. Cited on page 7.

[MGL06] P. Montag, S. Görzig, and P. Levi. Challenges of timing verification tools in the automotive domain. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 227–232, 2006. `doi:10.1109/ISoLA.2006.52`. Cited on page 12.

[MHAK15] Georg Macher, Andrea Holler, Eric Armengaud, and Christian Kreiner. Automotive embedded software: Migration challenges to multi-core computing platforms. In *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*, pages 1386–1393, 2015. `doi:10.1109/INDIN.2015.7281937`. Cited on pages 1 and 45.

[MKTM94] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling DAGs on multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, pages 446–451, 1994. `doi:10.1109/IPPS.1994.288264`. Cited on page 33.

[MMS01] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Parallel programming with a pattern language. *STTT*, 3(2):217–234, 2001. `doi:10.1007/s100090100045`. Cited on page 37.

[MNBSL12] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion. Multi-source Software on Multicore Automotive ECUs - Combining Runnable Sequencing With Task Scheduling. *IEEE Transactions on Industrial Electronics*, 59(10):3934–3942, 2012. `doi:10.1109/TIE.2012.2185913`. Cited on page 35.

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and data structures: The basic toolbox*. Springer Science & Business Media, 2008. Cited on page 75.

[MSR] Manufacturer Supplier Relationship (joint project). Retrieved May 10, 2018. URL: `http://www.msr-wg.de/`. Cited on pages 29 and 146.

[OFW$^+$05] Joshua O'Madadhain, Danyel Fisher, Scott White, Padhraic Smyth, and Yan biao Boey. Analysis and visualization of network data using jung. Journal of Statistical Software, 2005. Retrieved May 10, 2018 and source code available at `https://github.com/jrtom/jung`. Cited on page 78.

[OSE05] OSEK Group. OSEK/VDX Operating System Version 2.2.3, February 2005. Retrieved May 10, 2018. URL: `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`. Cited on pages 16, 17, 91, and 145.

[PKQn$^+$14] Miloš Panić, Sebastian Kehr, Eduardo Quiñones, Bert Boddecker, Jaume Abella, and Francisco J. Cazorla. RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. In *Proceedings of the Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, CODES '14, pages 29:1–29:10, New York, NY, USA, 2014. ACM. `doi:10.1145/2656075.2656096`. Cited on pages 35, 40, 116, and 117.

[Rei14] Konrad Reif. *Diesel Engine Management*. Springer, 2014. Cited on page 10.

[Rob] Robert Bosch GmbH. Bosch Mediaspace. Retrieved May 10, 2018. URL: `http://www.bosch-mediaspace.de`. Cited on pages 11 and 145.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessing (Research Monographs in Parallel and Distributed Computing)*. The MIT Press, 1989. Cited on page 33.

[SCCM15] Salah Eddine Saidi, Sylvain Cotard, Khaled Chaaban, and Kevin Marteil. An ILP approach for mapping AUTOSAR runnables on multi-core architectures. In *Proceedings of the Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '15, pages 6:1–6:8, New York, NY, USA, 2015. ACM. `doi:10.1145/2693433.2693439`. Cited on pages 36 and 40.

[SRA$^+$08] A. Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global Sensitivity Analysis: The Primer*. Wiley-Interscience, 1 edition, 2 2008. Cited on page 108.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005. URL: `http://mondrian.die.udec.cl/~mmedina/Clases/ProgPar/Sutter-TheFreeLunchisOver.pdf`. Cited on page 18.

[SZ16] Joerg Schaeuffele and Thomas Zurawka. *Automotive Software Engineering: Principles, Processes, Methods, and Tools*. SAE International, 2nd revised edition, 2016. Cited on pages 1, 24, 25, 26, and 146.

[TEHZ16] S. Tobuschat, R. Ernst, A. Hamann, and D. Ziegenbein. System-level timing feasibility test for cyber-physical automotive systems. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016. `doi:10.1109/SIES.2016.7509419`. Cited on page 15.

[UBG+13] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. G. Zaykov, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In *2013 Euromicro Conference on Digital System Design*, pages 363–370, 2013. `doi:10.1109/DSD.2013.46`. Cited on pages 8, 37, and 40.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008. `doi:10.1145/1347375.1347389`. Cited on pages 12 and 145.

[WKH+15] C. Wolff, L. Krawczyk, R. Höttger, C. Brink, U. Lauschner, D. Fruhner, E. Kamsties, and B. Igel. AMALTHEA - Tailoring tools to projects in automotive software development. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applica-*

tions (IDAACS), volume 2, pages 515–520, 2015. doi:10.1109/IDAACS.2015.7341359. Cited on page 26.

[ZG11] M. Zhang and Z. Gu. Optimization issues in mapping AUTOSAR components to distributed multithreaded implementations. In *2011 22nd IEEE International Symposium on Rapid System Prototyping*, pages 23–29, May 2011. doi:10.1109/RSP.2011.5929971. Cited on page 35.

[ZH15] Dirk Ziegenbein and Arne Hamann. Timing-aware control software design for automotive systems. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6, 2015. doi:10.1145/2744769.2747947. Cited on pages 14, 16, 20, 51, 80, and 113.

Publications by the Author

[LZG15] Martin Lowinski, Dirk Ziegenbein, and Sabine Glesner. Partitioning Embedded Real-Time Control Software based on Communication Dependencies. In Michal Antkiewicz, Joanne M. Atlee, Juergen Dingel, and Ramesh S, editors, *Proceedings of the International Workshop on Modelling in Automotive Software Engineering co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada. September 27, 2015*, volume 1487 of *CEUR Workshop Proceedings*, pages 3–12. CEUR-WS.org, 2015.

[LZG16] Martin Lowinski, Dirk Ziegenbein, and Sabine Glesner. Splitting tasks for migrating real-time automotive applications to multi-core ECUs. In *11th IEEE Symposium on Industrial Embedded Systems, SIES 2016, Krakow, Poland, May 23-25, 2016*, pages 113–120. IEEE, 2016. doi:10.1109/SIES.2016.7509418.

List of Figures

List of Tables

List of Algorithms

List of Symbols

DG	Dependency Graph
PG	Precedence Graph
$\mathbf{V}$	Set of graph nodes
v	Graph node
w	Weight of a graph node
$\mathbf{E}$	Set of graph edges
e	Graph edge
$\mathbf{R}$	Set of runnables
r	Runnable
ST	Start time of a runnable
FT	Finish time of a runnable
$\mathbf{D}$	Set of labels
d	Label
$\mathbf{T}$	Set of tasks
t	Task
P	Period of a task
D	Deadline of a task
J	Job instance of a task
m	Mapping function of a runnable to a task
c	Processor core
$oWCET$	Observed worst-case execution time of a runnable or task
$\mathbf{Q}$	Set of dependencies
q	Dependency
γ	Criticality of a dependency
σ	Evaluation flag/status
$label$	Associated label of a dependency
$\mathbf{C}$	Set of precedence constraints

$\mathbf{P}$	Precedence relation
c	Precedence constraint
o	Origin of a precedence constraint
src	Source runnable of a dependency or precedence
dst	Destination runnable of a dependency or precedence
$pred$	Set of predecessor runnables
$succ$	Set of successor runnables
$\mathscr{S}$	Set of synchronization points
s	Synchronization point
$grad$	Gradient of a precedence or synchronization point
g	Parallelization goal
$\mathsf{0}$	Strictly undelayed criticality of a dependency
$\mathsf{1}$	Strictly delayed criticality of a dependency
$\mathsf{01}$	Relaxed criticality of a dependency
$relax$	Relaxation function

List of Abbreviations

ABS	Anti-lock Braking System
AICC	Automatic Interval Control System
APD	Activity and Pattern Diagram
API	Application Programming Interface
ASAM	Association for Standardization of Automation and Measuring Systems
ASW	Application Software
BSW	Basic Software
CMMI	Capability-Maturity-Model-Integration
DAG	Directed Acyclic Graph
DC	Dependency Classification
DySched	Dynamic Scheduling
ECU	Electronic Control Unit
EMS	Engine Management System
ESC	Electronic Stability Control
FAS	Feedback Arc Set
FE	Forward Evaluation
GE	Genetic Algorithm
HAD	Highly Automated Driving
HLFET	Highest Level First with Estimated Times
ILP	Instruction-Level Parallelism
ILP	Integer Linear Programming
LET	Logical Execution Time
LIB	Last Is Best
MAPS	MPSoC Application Programming Studio
MCD	Measurement, Calibration and Diagnostic
MDX	Model Data Exchange Format
MFAS	Minimum Feedback Arc Set
MoC	Model of Computation
MSR	Manufacturer Supplier Relationship

NoC	Network-on-Chip
OEM	Original Equipment Manufacturer
OS	Operating System
oWCET	Observed Worst-Case Execution Time
PaVaSt	Parameter Variable Structure
RPM	Rotations Per Minute
RTE	Runtime Environment
SWC-T	Software Component Template
TADL	Timing Augmented Description Language
TDL	Timing Description Language
TRC	Traction Control System
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language
WCET	Worst-Case Execution Time
WDAG	Weighted Directed Acyclic Graph