

parMERASA

Multi-Core Execution of Parallelised Hard Real-time Applications Supporting Analysability

D 2.4 – Sequential to Parallel Program Development Path and Parallel Execution Patterns

Nature:	Report
Dissemination Level:	Public
Due date of deliverable:	2014-03-31
Actual submission:	2014-03-28
Responsible beneficiary:	UAU
Responsible person:	Theo Ungerer
Grant Agreement number:	FP7-287519
Project acronym:	parMERASA
Project title:	Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability
Project website address:	http://www.parmerasa.eu
Funding Scheme:	STREP SEVENTH FRAMEWORK PROGRAMME THEME ICT – 2011.3.4 Computing Systems
Date of latest version of Annex I against which the assessment will be made:	June 20, 2012
Project start:	October 1, 2011
Duration:	36 month
Period covered:	2013-10-01 – 2014-03-31

Release Approval

name	role	date
Pavel Zaykov	WP2 Leader	2014-03-28
Hans Regler	WP2 Partner	2014-03-28
Bert Böddeker	WP2 Partner	2014-03-28
Theo Ungerer	Coordinator	2014-03-28

DELIVERABLE SUMMARY

This deliverable incorporates the results from two prototyping deliverables: D 2.2 – Parallelised Applications of Avionics, Automotive, and Construction Machinery Domains, D 2.5 – Optimized Parallel Applications of Avionics, Automotive, and Construction Machinery Domains. This document covers the following tasks from the DoW:

Task 2.4: Detailed model of parallelisation and implementation of the parallel applications targeting maximum parallelism on the parMERASA multi-core simulator (m10 :: 15m):

- Parallel applications will be implemented targeting a high level of parallelism.
- Parallel applications will be run on as many cores as possible.
- Communication and synchronisation patterns as well as communication and synchronisation overhead will be investigated for the selected industrial applications.
- Optimisation and agglomeration suggestions will be investigated.
- UAU will contribute by refining the parallel software engineering approach, implementation of parallel execution patterns, and supporting parallelisation of applications.
- HON will continue with the second phase of the inter- and intra-partition communication and synchronisation behaviour analysis and modelling with respect to shared resources of a selected COTS HW and/or parMERASA platforms. This effort will be finalised by completing the statistical modelling of inter- and intra- partition communication and synchronisation and performing appropriate goodness-of-fit tests. Furthermore, parallel execution patterns should be applicable to both COTS and parMERASA multi-core platforms. These patterns should also handle both legacy and new applications. The effort will be naturally finalised by adapting the parallel execution patterns for a better WCET analysability and legacy code design artefacts reuse.
- DNDE will implement the parallel version of the selected automotive application targeting maximum parallelism and analyse communication and synchronisation overhead.
- BMA will implement the parallel version of the control code of a large drilling machine targeting maximum parallelism and analyse communication and synchronisation overhead.
- Feedback will be provided to WP3 about usage of tools and WP4 about system software.
- Check objective of an at least eightfold WCET speedup on applications in collaboration with WP3.
- Patterns chosen in task 2.2 will be assessed for suitability.
- Fulfilment of requirements defined in task 2.2 will be checked.
- Target after month 24 are parallel applications running on as many cores as possible, but potentially with a low efficiency due to a bad communication vs. computation ratio. Synchronisation and communication overhead has been assessed. Clear ideas arise regarding how the efficiency can be increased by techniques such as agglomeration of short threads or of threads with high synchronisation/communication demands.

Conclusions of task 2.4:

As described in D2.2 all applications were parallelised targeting a high level of parallelism, which allows further optimizations. All applications were ported to the parMERASA platform and make use of the common infrastructure defined by the kernel library.

For the P4080 platform an analysis of communication and synchronisation overheads was conducted.

DNDE's application does not require active synchronization between cores. Instead a delay mechanism is used to synchronize the execution on all cores with a global clock. The overhead

incurred by this mechanism (albeit small) has been assessed. An evaluation of the communication overhead is subject to the final phase of the project, because no additional communication mechanisms have been used until month 30 of the project.

In BMA's application, the communication and synchronization overhead was measured by comparing the time of get and set functions with and without synchronization primitives.

The at least eight-fold WCET speedup has not been reached for any application; however, this justifies the optimisation work in Task 2.5.

Besides these deviations all intended subtasks have been carried out successfully.

Task 2.5: Optimised parallel application implementation (m25 :: 6m):

- Parallel applications will be optimised and implemented targeting a high efficiency with a potentially lower level of parallelism. In particular the efficiency (speedup divided by number of cores used) should be higher than for the maximum parallel applications of the previous stage.
- Parallel applications will run on a sufficient number of cores yielding WCET and average speedups and efficiencies.
- Communication and synchronisation overhead will be assessed once more to validate the efficiency of the proposed optimisations.
- UAU will support parallelisation of applications and refine the parallel software engineering approach and implementations of parallel execution patterns.
- HON will contribute by adapting avionics applications to run on top of commercial RTOS and COTS HW, as well as on the tiny avionic RTE and parMERASA multi-core processor. Furthermore HON will perform initial test on both COTS and parMERASA HW.
- DNDE will implement an optimised version of the selected automotive application targeting maximum efficiency.
- BMA will implement an optimised version of the control code of a large drilling machine targeting maximum efficiency.
- Target after month 30 is running parallel applications efficiently on an adequate number of cores, i.e. with a good cost-performance ratio

Conclusions of task 2.5:

Progress in measurement of communication and synchronisation overheads was made; its finalization is postponed for the evaluation phase, i.e., Task 2.6. Optimizations towards higher efficiency are also in progress.

Besides this, all intended subtasks have been carried out successfully.

Milestone 3: Optimisation and Refinement

- parallelised application programs went through the agglomeration and mapping phases yielding optimal parallel program configurations by full system simulations (application programs, system software, parMERASA multi-core)
- parallel applications run efficiently on an adequate number of cores, i.e. with a good cost-performance ratio

- final debug feedback provided to WP3 tools and WP4 system software

With the restriction that efficiency will finally be provided in the concluding deliverable all tasks have been completed as defined.

TABLE OF CONTENTS

Introduction.....	8
1 Parallelization Approach with Parallel Design Patterns for Hard Real-Time Systems	9
1.1 Parallelization Approaches from HPC Domain	9
1.1.1 The PCAM-Approach by Foster	9
1.1.2 Approach by Mattson et al.: Parallelization with a Parallel Pattern Language	10
1.1.3 Discussion	10
1.2 Structured Parallelism: PDPs and Catalogue with PDPs.....	10
1.2.1 Description of Concepts	10
1.2.2 The parMERASA Pattern Catalogue.....	11
1.2.3 Requirements for the Analysability of PDPs.....	12
1.2.4 Examples for PDPs in Industrial Applications	13
1.3 Modelling with the Activity and Pattern Diagram (APD).....	13
1.4 The Parallelization Approach: Two-Phase Process to Reveal and Optimize Parallelism in a Model 15	
1.4.1 Phase I: Exposing a High Degree of Parallelism (Platform Independent).....	15
1.4.2 Phase II: Agglomeration and Mapping to Respect Trade-offs (Platform Dependent) ..	17
1.5 Fulfilling the Requirements for Static WCET Analysis	19
1.6 Tool-support for the Parallelization Process	20
1.6.1 Dependency Analysis.....	20
1.6.2 Speedup Approximation and Automatic Selection of Agglomeration Steps	21
1.6.3 Algorithmic Skeletons.....	21
1.7 Towards a Parallel Implementation	21
1.7.1 Implementation of PDPs.....	21
1.7.2 Synchronization of Access to Shared Resources	22
1.8 Summary, Future Work, and Outlook on Application by Application Partners	23
1.8.1 Avionics.....	24
1.8.2 Construction Machinery	24
1.8.3 Automotive	24
2 Overview of Industrial Applications	25
2.1 Integration of industrial applications to the parMERASA tools and platform	25
2.2 Communication and synchronization overheads – the evaluation strategy.....	26
2.3 Application WCET in the different application domains	27
3 Avionic Applications	28
3.1 Parallel applications	28

3.1.1	3DPP – initial parallelization	28
3.1.2	StereoNav – initial parallelization.....	29
3.1.3	Evaluation and outlook for optimization.....	29
3.2	Optimized applications.....	33
3.2.1	3DPP – optimized parallelization.....	33
3.2.2	StereoNav – optimized parallelization	34
3.3	Conclusions and outlook	37
4	Automotive applications	38
4.1	Parallel application	38
4.1.1	AUTOSAR reconstruction.....	39
4.1.2	Detection of Constraints.....	40
4.1.3	Timing Analysis	41
4.1.4	Scheduling	41
4.1.1	Implementation.....	42
4.1.2	Potential for Optimization.....	43
4.2	Optimized applications.....	44
4.2.1	Supertasks	44
4.2.2	Pattern-supported Parallelization	45
4.2.3	Classification of Constraints	45
4.2.4	Optimized Memory Layout.....	46
4.3	Conclusions and outlook	46
4.4	Future Work	47
5	Construction Machinery.....	48
5.1	Parallel application	48
5.2	Optimized application	52
5.3	Conclusions and outlook	54
6	Conclusions.....	55
7	References.....	57

INTRODUCTION

Future embedded systems will feature more often processors with multiple cores. To gain benefit from these additional resources requires multi-threaded applications. However, these applications must sustain timing analysability, hence have defined worst-case execution times (WCETs).

This deliverable presents both the parallelization paths for the initial parallelization (Task 2.4) and the optimized parallelization (Task 2.5) as sketched out in the Description of Work of the parMERASA project.

First (Section 1) a general approach is presented for parallelizing sequential code parts of a legacy single-core application. It is model based and, in difference to existing approaches, sustains timing analysability by strongly relying on structured parallelism based timing-analysable elements. This parallelization is applied or adapted by the three industrial applications (Sections 2, 3, 4, and 5).

Section 6 concludes the report with a summarization of the results.

1 PARALLELIZATION APPROACH WITH PARALLEL DESIGN PATTERNS FOR HARD REAL-TIME SYSTEMS

So far, there are mainly two well-known development approaches for the transformation of sequential code parts of single-core software into parallel multi-threaded software; both were defined in the high-performance domain, i.e., the *PCAM-Approach by Foster* (see Section 1.1.1) and the *parallelization approach by Mattson, Sanders, and Massingill* (see Section 1.1.2).

In this chapter, we introduce a new approach with clear advantages in methodology and development effort (especially) for the embedded hard real-time systems domain with future multi-core processors (e.g., up to 64 cores). The construction cycle is based on Foster while the strong usage of patterns as building blocks is similar to the approach by Mattson et al.

The main goal of our *pattern-supported parallelization approach* is to depict a way to develop (a) parallel software for embedded systems, which can be (b) mapped onto different many-core target architectures (c) providing high (worst-case) performance with (d) low development effort and (e) supporting timing analysis. It is a methodical model-based design approach to be executed by an engineer or software developer.

Its central idea is to allow parallelism only by structured parallelism defined by *Parallel Design Patterns (PDPs)* and *Synchronization Idioms (SIs)* out of a *Pattern Catalogue*. Throughout the parallelization process an extended version of the UML2 Activity Diagram (AD) called *Activity and Pattern Diagram (APD)* is created and worked on. This allows rapid definition and refinement of situations exhibiting chances for parallelism. This optimized model of the application is then the blueprint for adaption of the single-core software.

The following sections are a synopsis of our work already published in the articles [20, 19, 17], the technical report [14], and the articles [21, 18], which are submitted and under review. The structure is as follows: Section 1.2 introduces PDPs and related concepts which are the basis for modelling in Activity and Pattern Diagrams (APDs, see Section 1.3) during the parallelization approach (Section 1.4); related work to it is shown in Section 1.1. The effects on timing-analysis are elaborated in Section 1.5. Possible tool support for the parallelization is presented in Section 1.6 and the way back from the model to parallel source code is sketched in Section 1.7. Concluding remarks are given and the applicability in the parMERASA application domains are discussed in Section 1.8.

1.1 Parallelization Approaches from HPC Domain

This section shortly introduces the two well-known parallelization approaches from the HPC domain and puts them in context with our parallelization approach for hard real-time embedded systems.

1.1.1 The PCAM-Approach by Foster

The *PCAM-Approach*, introduced in the book of Foster [10] for the high-performance domain, describes a way to design parallel algorithms. The starting point is a “problem”, then four steps are performed leading to a parallel algorithm to solve the given problem.

The main idea is to split the basic problem into as many parts as possible by functional and domain decomposition. As next step, communication links between these program parts, which are called tasks, are re-established. To respect trade-offs, agglomeration steps are performed and the process concludes with a mapping phase also to tribute to communication and synchronization costs.

1.1.2 Approach by Mattson et al.: Parallelization with a Parallel Pattern Language

The *parallel pattern language* [25, 27] is split into four design spaces. These spaces contain PDPs of different granularity and functionality. The parallelization process with the parallel pattern language is, according to Mattson et al., an iterative process. Decisions and uses of patterns on lower abstraction levels might propose to apply another pattern on a higher abstraction level. An example for this is [26] giving more an organizational solution for the way from a sequential program to a parallel one instead of a clearly described kind of algorithm to apply.

Parallelization is done directly in code and not in an abstract model making it much harder to imagine and document candidate situations for parallel execution.

1.1.3 Discussion

The aim of the pattern-supported parallelization approach presented in Section 1.4 is to combine the advantages of both approaches while reducing the disadvantages, respectively. In difference to the approaches of Foster and Mattson et al., we target embedded multicores with much smaller workloads and smaller parallelization overheads, e.g., because of synchronization primitives supported by hardware (see e.g. [15]). Hence parallelism is interesting also on a much smaller granularity. In addition, our approach is based on an extension of the prevalently used UML2 Activity Diagram instead of a novel notation. The known parallel structure realized by the PDPs avoid complex synchronization and communication structures leading to lower implementation effort and much better WCET analysability. Also the methodical approach eases application even for inexperienced engineers.

1.2 Structured Parallelism: PDPs and Catalogue with PDPs

This section describes our idea of structured parallelism by Parallel Design Patterns (PDPs) and Synchronization Idioms (SIs). We present meta-patterns to describe PDPs and SIs and highlight requirements which PDPs and SIs must fulfil so that they can be included in a *Pattern Catalogue for Timing Predictable Parallel Design Patterns*. Examples for such PDPs found in industrial applications are presented shortly, too.

1.2.1 Description of Concepts

We mainly apply two concepts for introducing structured parallelism into sequential code parts of single-core source code, see Figure 1. The main difference between PDPs on the left (in Figure 1) and the stack on the right (in Figure 1) is proximity to implementation: The three concepts on the right are provided as source code whereas PDPs are textual concepts.

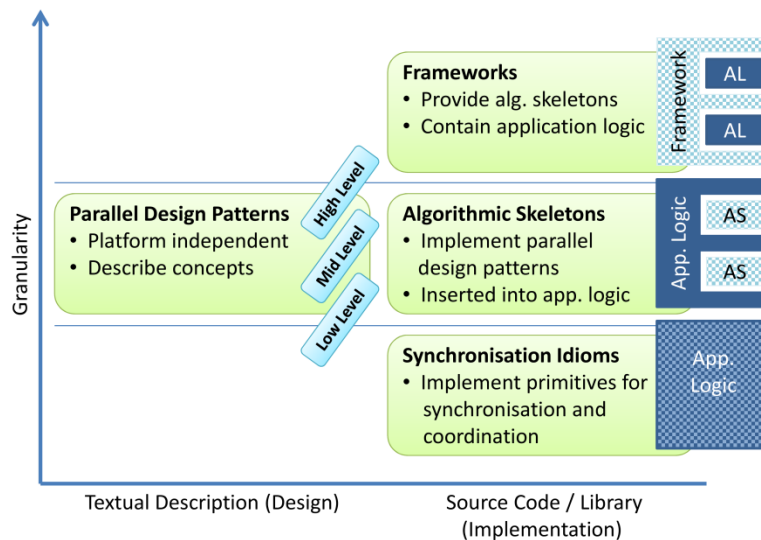


Figure 1: Overview of the concepts for parallelization with Parallel Design Patterns (PDPs). The illustrations on the right show the interplay with application logic (custom code) and the source-code parallelization concepts; checked elements contain synchronization elements.

Parallel Design Patterns (PDPs): Design patterns describe well-known and widely accepted solutions to recurring problems in a specific context. They were first introduced in 1977 by Christopher Alexander in the domain of architecture, and later in the domain of software engineering by Gamma et al. [11]. Because PDPs are described as text only, it is the burden of the software developer to implement them well, which can be a major challenge especially for inexperienced persons¹. However, to ease the implementation of PDPs, code examples are typically given in a Pattern Catalogue.

Synchronization Idioms (SIs) are the basic synchronization operations which are necessary for the implementation of PDPs. They are described in the Pattern Catalogue, too, because of their importance for and the strong links to the PDPs. Because not all possibly applicable primitives are timing analysable and hence useful for HRT applications it is necessary to select and document those which are allowed because they are timing analysable, e.g., ticket locks instead of spin locks [15, 13]. In contrast to PDPs, the SIs are provided for a target platform and (if applicable) also a real-time operating system (RTOS). Hence, the variance of implementation is near zero and, therefore, they are described together with source code. Often these are only a few lines of assembler code.

More general concepts on the source-code level are **Algorithmic Skeletons** [9] and **Frameworks** (similar to [24]). Algorithmic Skeletons are code fragments, typically made available as library, which implement PDPs. Algorithmic Skeletons can be embedded into an existing application and hence facilitate parallelism because a smaller fraction of source code must be modified for implementing parallelism (separation of concerns). Frameworks, in contrast, define a program structure (“semi-complete application”). Therefore, they are not useful for the parallelization of existing sequential programs but for developing parallel programs from scratch.

1.2.2 The parMERASA Pattern Catalogue

The parMERASA Pattern Catalogue [14] contains as a starting point 4 PDPs and 2 SIs, which are based on observations in the application introduced in Section 1.2.4. The PDPs and SIs are (as it is necessary

¹ See [29] for implementing the farm skeleton, which is strongly related to the PDP *Task Parallelism*.

for every Pattern Catalogue) specified in a fixed format, the meta-pattern (see Sections 2.1 and 3.1 in [14]). The focus of the parMERASA Pattern Catalogue is on timing predictability and timing analysis, hence only PDPs and SIs fulfilling these properties are contained.

Modifications to the meta-pattern structure, as defined by Mattson et al. [27], for our predictable PDPs are introduced from the real-time perspective. That is how to include the mandatory real-time requirements for programmers, and how to include the possible output for automatic timing analysis with WCET tools. Also, the forces/motivation part should include real-time aspects. Therefore, new items have been added—namely *Real-Time Prerequisites*, *Synchronization Idioms*, and *WCET Hints*—for transferring information needed for timing analysis from a programmer to a timing analysis tool (cf. Section 2.1 in [14]). These items include hints and requirements (defined by a timing analyser, i.e., the person performing the analysis) for a programmer to enforce timing analysability, too.

The data exchange and progress coordination between threads of a parallel program at synchronization points is an important factor concerning the estimation of WCET guarantees (e.g., worst-case waiting times that arise from threads interfering with and waiting on each other). On the one hand, for being able to compute upper bounds for execution time at all, synchronization techniques used at synchronization points in a parallel program need to be designed for timing analysability (see [15], for example), and mostly also need to be clearly understood by a static timing analyser or static timing analysis tool. On the other hand, the overestimation and pessimism introduced from synchronizations ought to be as low as possible to gain worst-case efficiency and performance, and timing predictable behaviour.

The *Synchronization Idioms* category of each PDP gives a list of stand-alone SIs on synchronization techniques. When using a specific PDP, for example, the programmer might have different requirements on how the data between concurrent HRT threads is exchanged, or how the progress is coordinated. That might be, for instance, a “last is best” strategy, or it might be required that the threads notify each other on each change of the shared data. Those different cases would then result in different SIs. Also, the use of the SIs, or in more detail the timing analysability of them, depends highly on the chosen architecture (ISA), the RTOS, and the programming model. So, each SI presents different timing analysable solutions, e.g., for securing a critical section, on different platforms.

1.2.3 Requirements for the Analysability of PDPs

Clearly not all (general) PDPs are appropriate for HRT embedded systems; some of them hinder a close estimation of time bounds with static code analysis by their dynamic nature or make it even impossible. The following requirements and recommendations, besides the typical guidelines for sequential applications (see, e.g., [32, 5, 12] turned out to be important:

- **Only Defined Synchronization Idioms and Points** For static WCET analysis the interference points are of big importance and must also be specified [28]. Hence, only the subset of synchronization primitives defined as SIs in the Pattern Catalogue is allowed and synchronization must be deterministic. (Counter-example: speculation)
- **Static Mapping of Tasks to Threads and Cores** It must be clear before execution which code fragment is executed on which thread and core. Else a “worst-case mapping” would have to be assumed leading to high over-estimation; the impact is especially high for architectures with non-uniform communication latencies. (Counter-example: work stealing)
- **Limited Usage of Shared Dynamic Data Structures** Dynamic data structures typically come hand in hand with dynamic memory allocation. Both have to be avoided in single-core HRT

programs for close WCET estimates and this persists for parallel programs. However, some PDPs are very hard to implement without queues and similar structures. These PDPs should not be included in the Pattern Catalogue and are not allowed for parallelization because of this.

- **No Indeterminism by Race Conditions** Like all other types of indeterminism also race conditions must be strictly avoided in HRT systems.

It is clear that only PDPs fulfilling the above requirements can be included in a Pattern Catalogue with time-predictable PDPs like the parMERASA Pattern Catalogue. Also it helps understanding the implementation pitfalls if there is at least one working and analysable implementation available.

1.2.4 Examples for PDPs in Industrial Applications

In the scope of the parMERASA project we discussed the parallelization of four industrial applications from the construction machinery, automotive and avionics domains. Starting with PDPs from the *algorithm strategy* and *implementation strategy* levels of OPL by Mattson et al. [27] we found the following linkage between our industrial applications and some PDPs, which are now the basis of the parMERASA Pattern Catalogue [14]:

- **Avionics Applications – 3D Path Planning and Stereo Navigation** These applications are data intensive, one of them processes images from two cameras, the other one calculates a path in an array of 3D pixels. Parallelization is here possible by (a) running the recurring processing steps as pipelines, which maps to the PDP *Parallel Pipeline*, and (b) by splitting input data and processing the parts with the same function on different cores—this is *Data Parallelism* (also known as SPMD).
- **Construction Machinery – Control Code for Foundation Crane** The structure of the application, which is a prototypical example for a complex industrial control code as it can be found in many different types of machines, defines many tasks which are executed periodically—this corresponds to *Periodic Task Parallelism*. Also there is a so-called main-loop, which is a comparatively big control loop, and many groups of the comprised tasks can be executed in parallel because they often control different parts of the machine. *Task Parallelism* is the PDP to model this.
- **Automotive Application – Diesel Engine Control Code** The dominant pattern is the repeated execution of tasks², either regular for a time interval or triggered by an external interrupt describing the crank angle in the engine. This can be modeled by an extended *Periodic Task Parallelism* PDP, taking into account event-driven tasks, too.

All four PDPs mentioned above fulfil the requirements defined in Section 1.2.3. Nevertheless, PDPs are textual concepts. Hence the developer or engineer responsible for the actual implementation must also know about the requirements and recommendation for time-predictable code. Algorithmic Skeletons can help here a lot to reduce coding effort and ensure analysability.

1.3 Modelling with the Activity and Pattern Diagram (APD)

The APD is derived from the Activity Diagram (AD) in the Unified Modeling Language 2 (UML2, [1]) and provides a notation for quick modelling and tuning of parallelism. This is because first the UML2 AD and our APD are easy to derive from existing code in comparison to notations like the UML2 sequence diagram, Petri nets, or finite-state machine. Second there is already strong tool support for modelling and drawing UML2 ADs and a big fraction of these tools can be used to model APDs, too.

² A task shall be a function call with self-contained competence, e.g., to control a defined electric motor.

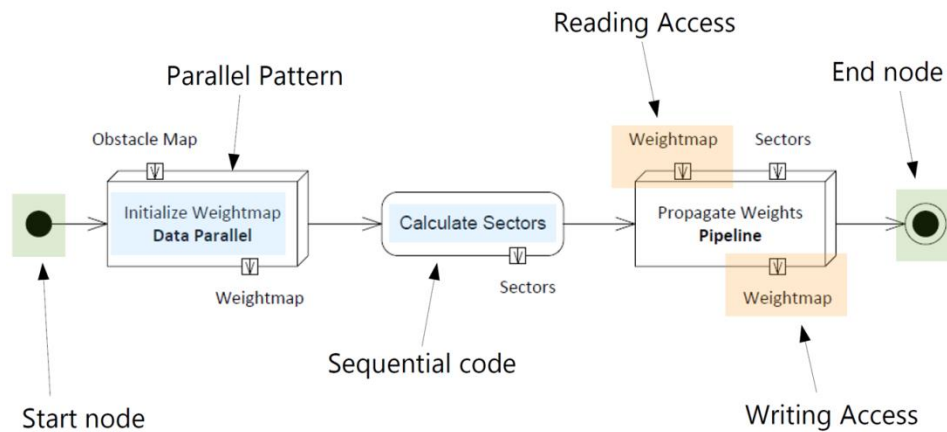


Figure 2: Example of an APD with PDP at the left and right and a sequential code block in the centre.

An example APD is shown in Figure 2. Compared to the UML2 AD, there are only two changes: (1) In addition to the elements of the basic AD, a second kind of activity nodes is added in the APD to model the PDPs. They can be employed exactly the same way and behave exactly the same way as the (sequential) activity nodes but can encapsulate not only a single but multiple functionalities modelled as separate APDs. (2) The usage of the bold “fork” and “join” bars existing in the basic UML2 AD to model creation and join of threads is not allowed any more.

Loops often provide lots of possibilities for parallelism. Hence they are modelled as pattern nodes in Phase I of our approach leaving the calculation of the degree of parallelism for Phase II, where it can be scheduled and the actual degree of parallelism is calculated and/or defined. It is very important to model dependencies in an early phase because they restrict parallelism and allow scheduling. Therefore, activities and patterns have to be annotated with *input*, *output*, and *resource pins* expressing the use of data structures and shared resources like special devices or interfaces. Also interrupts can be annotated in a similar way. The APD can be further extended to model more functional and non-functional aspects.

Some PDPs have to be annotated with further parameters. As an example, for patterns representing loop structures like the parallel pipeline pattern it is important to denote the number of iterations which it is executed. For later optimization of the APD it helps a lot to describe hotspots in the code, i.e., activities executed for a very large fraction of the overall execution time. This can be done by assigning weights to them, e.g., representing average execution time (ACET) or the worst case execution time (WCET) depending on later optimization goals.

From the analysis of the BMA application we had to learn that the modelling of data dependencies by pins and ports can reach its limits: For the about 300 global shared variables it was better to store them in a separate file where for each activity in the diagrams a section describing the dependencies is added. Because of its simple structure this file format is still easy to read and parse.

The APD can be represented without any problems in XML (either as XMI or custom format) or the more compact notation shown in Figure 3 for the main control loop of the BMA application. This allows a later automatic model-based optimization based on the APD and the file describing the data dependencies.

TP vBetrieb [4 seq. + 4 TP] 114882, 116515, 58947, 55298
TP vLogo_allg [7 seq.] 23272, 44005, 83385, 22211, 20147, 39429, 21253
TP vLogo_mc [1 seq. + 2 TP] 65992
 TP Funktionen_alle_MCs [12 seq. +1 TP] 19338, 19870, 50085, 37170, 42556, 39659,
 89637, 118031, 88613, 396529, 40114, 191066
 TP vFRG_Fahrwerk [2 seq. + 1 TP] 100460, 105366
 TP vfahrwerk [3 seq.] 65772, 191781, 195926
 TP Funktionen_Oberwagen [8 seq. + 1 TP] 68700, 107885, 114454, 68556, 25274, 45401,
 68975, 107919
 TP vFreifall [4 seq.] 41144, 326724, 238664, 71890
TP vWandlung_mc [13 seq.] 20846, 44082, 23862, 57897, 59539, 25696, 19458, 22159, 22284,
 40000, 22510, 22480, 25258
TP vVerfahren/vfunktion_bdc [3 seq.] 494761, 451631, 674207
TP Everything else [4 seq.] 23051, 23088, 19660, 21454

Figure 3: Nesting of Task Parallelism instances and sequential activities (here represented by their ACETs); as example, Task Parallelism vBetrieb consists of 8 subordinated activities from which 4 are sequential (represented by ACETs) and 4 are other Task Parallelism instances (vLogo_allg, vLogo_mc, ...).

1.4 The Parallelization Approach: Two-Phase Process to Reveal and Optimize Parallelism in a Model

Parallelism is extracted (Section 1.4.1) and optimized (Section 1.4.2). The approach describes a systematic way starting with a sequential and resulting in a parallel program. PDPs are the only allowed means for introducing parallelism, hence the resulting program features structured parallelism.

The applied parallelization approach is model-based. It requires manual work to isolate and describe situations suitable for parallelization. They must match PDPs as described in the parMERASA Catalogue (see Section 1.2.2 and [14]) with time-predictable design patterns. For modelling the *Activity and Pattern Diagram (APD)* as introduced in Section 1.3 is employed.

1.4.1 Phase I: Exposing a High Degree of Parallelism (Platform Independent)

The goal of the first phase of the parallelization approach is to construct a model of the software by APDs and to reveal a high degree of parallelism in this model by PDPs. In order to allow optimizations, this parallelism should be about a magnitude higher than the level which seems reasonable for the target platform. Both (a) situations for parallel execution fitting design patterns have to be spotted and, what proved to be the more work intensive task, (b) dependencies between the different code parts have to be identified and described. This is necessary to be able to decide if a parallel execution is indeed worthwhile. Phase I of the approach covers both the *Partitioning* and *Communication* phases of the Foster approach, which are executed concurrently.

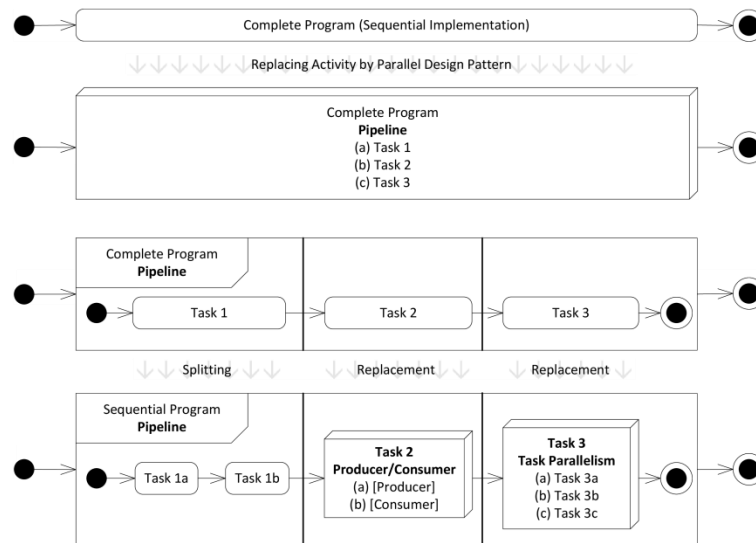


Figure 4: Intermediate APD (top) representing a sequential implementation in a single activity and its decomposition with patterns by repeated replacement and splitting of activity nodes

The starting point is either a problem description or, and this case is in the focus here³, a sequential software represented by an activity chart with a single activity. Two “operators” are to be applied to increase the level of details in the diagram; Figure 4 shows an example:

Replacement Replacing an activity by a **PDP** as specified in the Pattern Catalogue, which acts as a kind of container for sub-graphs. This operation should always be preferred over splitting because it increases parallelism.

Splitting Separate an activity into **several (sequential) activities** (hence more details are added for this sequential program part).

These operations should be repeated until it is clear that the level of parallelism cannot be increased any further, either technically or because the resulting activities are very small⁴. All subtasks after this last step are defined to be *non-parallelizable activities*. Throughout the process, the decomposition of an activity should have the same semantic meaning as the activity itself.

For identifying situations for PDPs, the source code has to be checked manually. Unfortunately, there is yet no supporting program available for this. Even if available, automatic tools should be used carefully: The main difference between automatic parallelization and our approach is that also situations fitting only roughly (and maybe requiring manual adaption of the source code) can be identified by the developer whereas automatic tools will only find exact matches.

³ We share the opinion that a parallelization of a single-core application cannot reach the same quality as developing software from scratch with keeping parallelism in mind throughout the whole development process. Nevertheless, the development effort for a parallelization of legacy software is supposed to be much lower.

⁴ The term “maximum parallelism” was used throughout the DoW to characterize the principal project steps based on the PCAM approach of Forster: parallelise for “maximum parallelism”, then optimize and agglomerate. Forster avoided in his approach the term “maximum parallelism” and used “a parallelism degree one order of magnitude larger than the available number of processors”. His approach target (numerical) high-performance algorithm design from scratch.

Starting the parallelization from source code, the intention to keep parallelization effort low, and also the need to sustain timing analysability lead to the decision to concentrate on functional decomposition instead of domain decomposition. Functional decomposition allows per definition of Foster less potential for parallelization because it “does not yield a large number of tasks” [10, p31].

The revealed degree of parallelism in the resulting APD is, as mentioned already, high enough, if it is far above the number of cores in the target system. Also the search for further possibilities can be stopped if the comparison of execution times with the overheads for parallelization (see O_n in Section 1.4.2) brings out that a speedup is not achievable with further parallelization. Some code structures can prevent a further increase of the level of parallelism, too. In the BMA application, for example, the examination is stopped if long if-then-else structures, state machines, or only calls to library functions are found. Adding parallelism is complicated and often not effective because of the typically very small code sections.

For each potentially parallel situation it has to be decided if the code fragments—in the model they map to activities—can actually be executed in parallel without difficulties. This is mainly dependent on the data and sometimes control dependencies. As first step, for each activity the accesses to shared variables have to be identified as well as the access types (read or write). The second step is to compare the access sets of activities to be executed in parallel: If they are disjoint or if they only match for reading accesses then a parallel execution is very probably possible. If the sets are not disjoint and one or more accesses of their intersections are writing accesses, then a more detailed examination is necessary. To avoid race conditions, the order in which read and write accesses are performed often needs to be preserved.

The result of the first phase of the parallelization approach is APDs with PDPs for the main loop.

1.4.2 Phase II: Agglomeration and Mapping to Respect Trade-offs (Platform Dependent)

The aim of the second phase is adjusting the parallelism discovered in the first phase (Section 1.4.1) to the target architecture. The main advantage of doing the optimization in a model, too, is that the labour-intensive implementation task can be delayed for an already strongly optimized model of the software, hence (hopefully) reducing tuning efforts afterwards.

The second phase can be interpreted as optimization problem. For symmetrical many-core processors like the parMERASA architecture (if clusters are ignored for the moment), the optimization of parallelism is interpreted as selecting a set of PDPs for parallel execution and assigning threads (and hence cores) to them. For this, a parameter is defined for each PDP found in the first phase. This parameter specifies the number of threads used for the execution of the pattern. Its value can vary between one and the minimum of the number of activities contained in the pattern and the number of threads available for the main loop in the system.⁵ Two objective values can be calculated in the model for each such configuration:

- The number of **threads** necessary for executing the configuration with the specified degree of parallelism.

⁵ If a pattern is assigned fewer threads than activities are contained in it, then these activities (and also sub-patterns) are grouped for the optimization by their approximated execution time with a greedy algorithm leading to more or less equal execution times for all threads.

- The **approximated worst-case execution time** of a PDP is based on the worst-case execution times of a sequential execution in clock cycles on the target platform. It is assumed that the execution of multiple program parts A, B, \dots (with at most n parts) in parallel by n threads takes as long as the longest execution time of a single program part plus a fixed overhead O_n for parallel execution with n threads:

$$t_n(A||B||\dots) = \max(t_1(A), t_1(B), \dots) + O_n$$

On the one hand, this approximation can only be very rough because synchronization effort for multiple activities (and their shared variables) executed in parallel is very hard to estimate.

On the other hand, if the implementation is done with algorithmic skeletons providing implementations of PDPs (cf. [8, 9]), an assessment of the overhead O_n is reasonable and feasible (see also [2]) probably also for the worst-case. For the applied skeleton implementation the following function was found to model the overhead for skeleton execution in the average-case, including thread acquisition from a thread pool and releasing them after execution of the skeleton, precisely on the target platform (an extension of the analysis to the worst-case is work in progress):

$$O_1 = 25,013; O_n = 79,766 + n * 39,883$$

These numbers were gained by running a synthetic benchmark, which represents an instance of Data Parallelism, and analysing the results

Because of the typically large design space (for the BMA application with 11 task parallelism instances it comprises $\approx 110 \cdot 10^6$ possible configurations) a genetic algorithm is applied for a multi-objective exploration minimizing both the approximated execution time and the number of necessary threads. The tool developed especially for this task reads the APD of the control loop from an XML file as well as the list of functions with their accesses to shared variables built during the code analysis.

Based on the Pareto-optimal configurations the execution times, the speedup, and efficiency can be estimated. This helps to tune parallelism by assigning best number of threads to each PDP in the model.

The implementation effort, i.e., the transition from the optimized model to source code, is mainly defined by the need for (a) implementing the PDPs and (b) securing accesses to global shared variables, data, and devices (resources, in general). Because of provided Algorithmic Skeletons the effort for implementing PDPs will be low for the programmer, see later Section 1.7.1. Adding synchronization for global shared variables, in contrast, is the main driver of implementation effort (see Section 1.7).

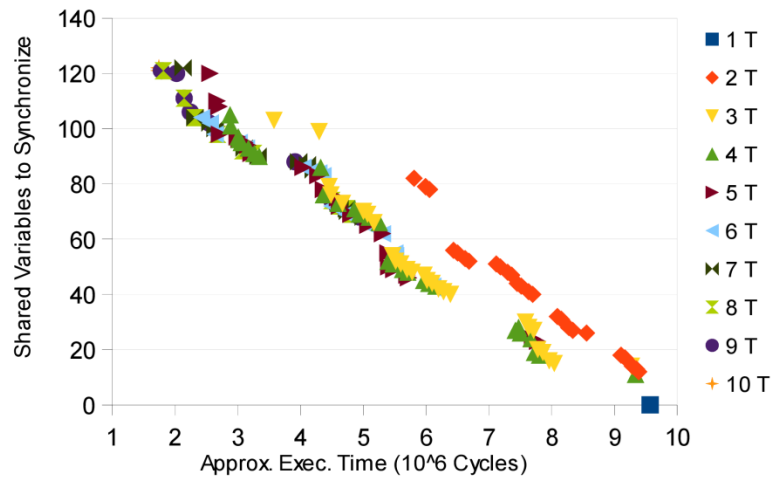


Figure 5: Approximated execution time with parallelization overheads and the number of shared variables (“worst-case”) necessary for the implementation with different numbers of threads (not respecting synchronization times of shared variables).

To assess this programming effort for synchronization, the number of variables, which have to be secured, is calculated based on the model of PDPs and the dependency list from Phase I (Section 1.4.1). The results show that with reduction of the estimated execution time the number of concurrently accessed shared variables grows nearly linearly (see Figure 5 for the main control loop of the BMA application).

1.5 Fulfilling the Requirements for Static WCET Analysis

One of the main challenges for static WCET analysis of parallel applications [30] is to determine the interdependencies between different threads [7]. Because not all such dependencies can be detected automatically and reliably in source code or the binary file, the static WCET analysis of industry applications with the static WCET analysis tool OTAWA⁶ [3] is so far a time consuming partially manual task. Annotations in source code can ease this.

For OTAWA, an annotation format to specify IDs pointing to lines in source code is being defined. These IDs are placed as comments in the source code and are then referenced in an XML file describing the interactions between different threads, e.g., different code parts requiring the same lock for continuation or for threads participating at a barrier.

It is clear that specifying these IDs and annotations requires knowledge of the specification format and lots of experience. To reduce this overhead the description of all PDPs is enriched in the Pattern Catalogue with (a) requirements for WCET analysis and (b) the necessary annotations for WCET analysis. More formally speaking, the meta-pattern describing the format of the description of a PDP is extended.

This eases the analysability of the whole parallel application (a) because only analysable PDPs out of a modified Pattern Catalogue (i.e., a subset of all PDPs) can be used for the parallelization and (b) because WCET analysis of sequential code blocks is supposed to be feasible. Also the WCET analysis is eased because the SIs are defined already for the platform and can be marked. Hence custom, unverified, and hard to analyse implementations of for example barriers should be eliminated.

⁶ Available as open-source software: <http://www.otawa.fr>

1.6 Tool-support for the Parallelization Process

Although the parallelization approach as presented in Section 1.4 is basically a work-flow for manual execution there are many different situations in the approach where tools could help to increase the degree of automation. In the following, tools are suggested to (a) reduce parallelization effort, (b) reduce implementation and testing effort, and (c) increase performance of the parallel implementation. Figure 6 gives an overview of the proposed tools for assistance during the parallelization process; they are described in details in the following sub-sections.

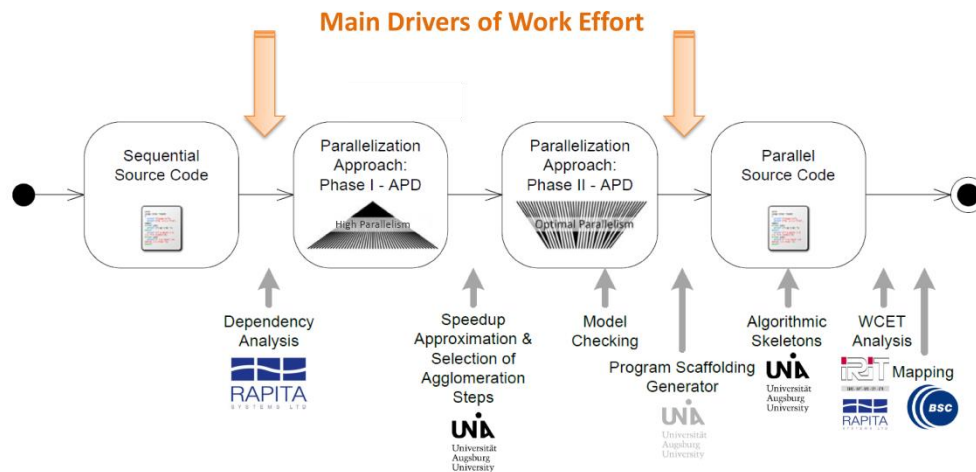


Figure 6: Suggested tools to ease the parallelization process

Rapita Systems is developing in the parMERASA project also further tools to check parallel implementations, e.g., a tool for code coverage, equality of a sequential to a parallel implementation, and a visualization tool. The WCET tools OTAWA and RapiTime are improved to support the analysis of parallel applications, too. Details on these supporting tools are provided in the deliverables by Work Package 3.

1.6.1 Dependency Analysis

As mentioned before the role of dependencies expressing accesses to shared data structures and resources has important impact on the parallelization. In the APD these dependencies are modelled as pins and ports; alternatively a list of activities and accessed shared resources can be built.

Without sophisticated static analysis this is a hard job. Several tools have been presented already; examples are [22, 23, 31]. Assistance could also be provided by the MAP file generated by the linker; however, this file does not contain the functions accessing the global variables. CScope⁷ can indeed provide assistance because it can list the accessing functions for a global variable; however, a list of variables accessed by a function is not available.

In the scope of the parMERASA project, a tool for the analysis of dependencies is under development by Rapita Systems (see parMERASA Work Package 3).

⁷ Homepage: <http://cscope.sourceforge.net/>

1.6.2 Speedup Approximation and Automatic Selection of Agglomeration Steps

The APD created in Phase I of the parallelization approach can also be understood as computationally accessible data structure. With worst-case timing information from a sequential implementation and the dependencies expressed in the APD the theoretically possible speedup for the different PDPs could be calculated automatically. This approximation would be more precise if algorithmic skeletons are applied. Also the approximation can be refined over time to be closer to speedups achieved in actual implementations.

Such a worst-case performance model is the basis for a model-based optimization as suggested in Section 1.4.2.

1.6.3 Algorithmic Skeletons

Because we propose to only use PDPs to introduce parallelism, it is possible to apply *algorithmic skeletons* for implementing the PDPs, so also the code for parallel code structures can be reused leading to a reduction of development and debugging efforts.

In the parMERASA project, we are working towards algorithmic skeletons for the most frequently used PDPs found in the evaluated industrial applications (see Section 1.2.4). The focus will be on timing analysability and to reduce implementation effort for our target applications. This is work in progress; more details are presented in Section 1.7.1.

1.7 Towards a Parallel Implementation

The parMERASA platform provides a programming API similar to POSIX Threads (PThreads). Two synchronization primitives as described by Gerdes et al. [15, 14] are available for process and progress coordination: *Fetch and increment barriers (F&I-barriers)* and *ticket locks*. Both are specifically designed for worst-case performance by providing fairness between hard real-time threads.

To realize parallelism as described and optimized in the model of the main loop it is necessary to implement the PDPs (Section 1.7.1) and to synchronize accesses to shared resources (Section 1.7.2).

1.7.1 Implementation of PDPs

PDPs are, as mentioned already, abstract concepts textually describing best-practice solutions for recurring situations of parallelism. *Algorithmic Skeletons*, first introduced by Cole [8], are a concept on source code level for the implementation of PDPs; they implement a parallel structure and contain placeholders for problem-specific procedures and declarations (which are inserted, e.g., as function parameters). In general, Algorithmic Skeletons reduce the development and maintenance effort for typically error-prone parallel code by reusability.

For the C programming language and with PThreads-like threading no Algorithmic Skeleton Library is available (cf. [16]). Hence, we implemented a Skeleton for the Task Parallelism PDP, i.e., a *Farm*-like skeleton [6]. A timing analysis is aimed for in near future⁸. Therefore, the Skeleton implementation was done according to known guidelines for timing analysable code [5, 12]. For example, the assignment of tasks is done statically as well as the mapping of tasks to threads and cores with

⁸ The developed Skeleton is the first of the *Timing-Analyzable Algorithmic Skeletons (TAS)*, which are under development.

running one thread on one core; there are no dynamic memory allocation, no work stealing [4] or other kinds of dynamicity.

The following is an example for a skeleton instance:

```
// Variables
SHARED_VARIABLE(membase_uncached0) volatile
    void * bdc_tp_args[3] = {NULL, NULL, NULL};
SHARED_VARIABLE(membase_uncached0) volatile
    tas_runnable_t bdc_tp_runnables[] = {
        (tas_runnable_t) bdc_winde1_runnable,
        (tas_runnable_t) bdc_winde2_runnable,
        (tas_runnable_t) bdc_funktion_runnable
    };
SHARED_VARIABLE(membase_uncached0) volatile
    tas_taskparallel_t bdc_tp = {
        bdc_tp_runnables, bdc_tp_args, 3
    };

// Execution (runs in main thread)
tas_taskparallel_init(&bdc_tp, 3);
tas_taskparallel_execute(&bdc_tp);
tas_taskparallel_finalize(&bdc_tp);
```

1.7.2 Synchronization of Access to Shared Resources

To prohibit the parallel modification of shared resources, which could lead to unpredictable states, the accesses to them have to be synchronized. This is done with ticket locks. For each shared resource a ticket lock is added. Before accessing a resource, the respective ticket lock is acquired and released after the access.

For each such shared resource, which can possibly be accessed by multiple concurrently executed threads, all accesses in these threads have to be found and locks have to be placed. This can be frustratingly complex due to the often big number of “access sites”, the possibility of race conditions, and deadlocks if the locking order is not kept consistent.

To ease the adaption of the source code with respect to synchronizing accesses to shared variables the use of mutator methods (like `get/set`) is preferred wherever possible. With a custom code generator the source code of the mutator functions as well as other code fragments⁹ are produced for a shared variable. The lock and unlock of the related ticket lock are then hidden in the mutator methods:

```
SHARED_VARIABLE(membase_uncached0) volatile ticketlock_t sync_foo;
SHARED_VARIABLE(membase_uncached0) volatile static byte foo;

byte get_foo();
void set_foo(byte new_foo);

/*****
 * get and set functions
 *****/
```

⁹ In detail: `get` function, `set` function, variable and ticket lock definition, ticket lock initialization, function definitions for header file

```

byte get_foo(){
    byte result;
    ticket_lock(&sync_foo);
    result = foo;
    ticket_unlock(&sync_foo);
    return result;
}

void set_foo(byte new_foo){
    ticket_lock(&sync_foo);
    foo = new_foo;
    ticket_unlock(&sync_foo);
    return;
}

/*****
 * function code
 *****/
ticket_init(&sync_foo);
extern ticketlock_t sync_foo;

```

Reading accesses throughout the code base are replaced by the `get` method and writing accesses by the `set` method, respectively.

It is obvious that mutator methods can only be used for atomic access to one variable at a time. If multiple variables should be updated only in one “transaction” because of a kind of inherent relationship, e.g., a sensor value and the exact time of its measurement, then (unchanged) mutator methods cannot be employed. Also, if lots of operations have to be done on a larger data structure, then calling a mutator function separately for each operation is also not the appropriate way because of the impact on performance. Instead, a ticket lock should be acquired before all the operations and released afterwards.¹⁰

The performance impact of mutator methods is considered to be low because they provide good situations for function inlining by a compiler; in any case it is low enough to justify the improved clearness of the source code.

1.8 Summary, Future Work, and Outlook on Application by Application Partners

In this chapter an approach for parallelization of sequential code parts of single-core applications was introduced. It comprises two phases to be performed in a UML-like model of the code: revealing parallelism and optimizing parallelism. The impact on WCET analysis was described. In short, software parallelized with the approach and suitable Parallel Design Patterns and Synchronization Idioms, is timing analysable.

Future work comprises mainly support of the approach by providing timing-analyzable algorithmic skeletons, a way to approximate the WCET speedup and further tool support.

¹⁰ Alternatively a `get` method can return a copy of the structure which can be kept locally for reading operations. However, if the structure is modified, consistency can be an issue when the local copy is written back.

The following subsections shortly describe the parallelization ideas for the applications and the relationship or applicability of the presented parallelization approach.

1.8.1 Avionics

HON applies the parallelization approach from UAU. In the parallelization of the 3D-path-planning application, we apply a parallel pipeline and in this pipeline we implement multiple instances of data parallelism. In the StereoNav application, also the dominant pattern is a parallel pipeline. Inside the StereoNav pipeline, many steps are realized by data parallelism, e.g., images from the two cameras can be processed in parallel.

1.8.2 Construction Machinery

The parallelization approach as developed by UAU is applied by BMA for the construction machinery application. There are some periodic tasks, which were executed by a scheduler interrupting the remainder of the program (called main-loop). Those tasks are now executed on dedicated cores continuously. Also periodic tasks can be extracted from the main loop and be mapped to cores. This matches to the pattern “Periodic Task Parallelism”. The main-loop provides many instances of the “Task Parallelism” Pattern. Timing-analyzable Algorithmic Skeletons (TAS), which are being developed by UAU, are used for the implementation of the Parallel Design Patterns.

1.8.3 Automotive

Data dependencies between *Runnables* were analysed and the worst-case execution time for the parMERASA target architecture was estimated using OTAWA provided by UPS. A schedule was derived from this information using the Mapping Tool developed by BSC. It executes independent Runnables in parallel, i.e. task parallelism is exploited, but precedence constraints are respected. The scheduling also defines the time when communication between Runnables takes place. Thus, no further synchronization is required. The implementation of the complete application as well as the parallelization of single Runnables is subject of the optimization phase. Runnables will be implemented by PDPs to execute a single Runnable on multiple cores.

2 OVERVIEW OF INDUSTRIAL APPLICATIONS

In this section, we provide an overview of the industrial application, which are representatives of avionics, automotive and construction machinery domains. By overview, we refer to the integration status of industrial applications to the parMERASA tools and platform (see Section 2.1) as well as the HON work on the COTS platform, the set of measured parameters for the communication and synchronization overheads (see Section 2.2), and the way application WCET is computed for each one of the application domains (see Section 2.3).

2.1 Integration of industrial applications to the parMERASA tools and platform

Applications	Parallel Design Patterns	parMERASA Platform		COTS	Mapping tool	Rapita	OTAWA
		shared	ODC2				
HON – 3DPP	✓↻	✓↻	✓	✓	N/A	⚠️★	✓
HON – SN	✓↻	✓↻	✓	⚠️	N/A	⚠️	⚠️
DNDE	⚠️↻	✓↻	⚠️	N/A	✓↻	⚠️↻	✓↻
BMA	✓	✓	⚠️	N/A	N/A	⚠️★↻	⚠️

Figure 1 Integration of industrial applications to the parMERASA tools and platform

In Figure 1, we present the current integration status of the industrial applications with respect to the parMERASA tools and platform.

In Figure 1, by ✓ we identify activities that has completed successfully. By ⚠️, we identify on-going activities. By ↻, we mark activities where the industrial applications have influence to the parMERASA tools and platform. By N/A, we mark those activities, which are not available and they are considered to be outside of the project scope. By ★, we identify that preliminary analysis with the RapiTime tools has been applied on a sequential single-core implementation.

By HON-3DPP and HON-SN, we denote the 3D Path Planning and Stereo Navigation applications delivered by Honeywell. DNDE corresponds to the Denso engine control application and BMA is the construction engineering application delivered by Bauer.

The “parallel design patterns” column corresponds whether the applications are parallelized and optimized with the help of the parallel design patterns (see Section 1). For the parMERASA platform, we consider two instances – with shared memory (delivered in m24) and the distributed memory map (delivered in m30). For the latter, we mean that each one of the cores in the many-core architecture has a local memory or cache. In case a cache is used, we need to deploy a cache coherence protocol such as ODC2 (On Demand Cache Coherence). From the WCET analysis perspective, the new memory map results in identifying and placing as much application data as possible in the local memories/caches. In such a way, the application alleviates the stress on the

shared resources (i.e., memory) than have a shared global memory space. As a result, the collisions on the shared memory resources are reduced and we expect the WCET speedup to be improved.

The COTS (Commercial off-the-shelf) platform has been analysed by Honeywell only. According to DoW, the other application partners are not obligated to conduct analysis on COTS target.

The mapping tool has been used for the automotive application, since the DNDE application has high task level parallelism, i.e., the number of the *Runnables* is much higher than the number of cores in the many-core architecture. In the considered applications in the avionic and construction machinery domains, the main level of parallelism is the data-level parallelism, which allows static mapping of threads to cores, i.e., scaling the number of running threads to the number of cores in the many-core architecture.

The considered parMERASA tools for WCET estimation are RapiTime (measurement approach) and OTAWA (static approach). Both tools support analysis of applications running on the parMERASA platform. As the table suggests, there is on-going analysis with both tools on the industrial applications.

2.2 Communication and synchronization overheads – the evaluation strategy

In this section, we provide an overview on the different approaches applied to measure the synchronization and communication overheads.

For each one of the application domains, we envision to evaluate the **synchronization** overheads by:

- Avionics – compare the effect of different synchronization primitives (such as conditional variables and barriers) on the application worst case execution time;
- Automotive – the engine control application does not employ an explicit communication mechanism (e.g. Runnables are not blocked on a barrier), instead the correct application execution is guaranteed by implicitly delaying a Runnable until all its incoming data-dependencies are satisfied. Such an implicit guarantee is realized by assigning a conservative budget to a Runnable (execution window). In case a Runnable finishes earlier, we employ a delay function, which generates a delay until the budget (WCET) expires. The synchronization overhead is defined by the sum of time that the Runnables WCET is increased by the delay mechanism;
- Construction machinery - for the construction machinery application, we measure the synchronization overheads by subtracting two application configurations running for the same application inputs - an application configuration with synchronization primitives to an application without any synchronization primitives. Please note that the latter is employed for evaluation purposes only where the data integrity is not preserved. The application behaviour in both configurations is the same.

We envision evaluating the **communication** overheads by measuring the inter-partition communication delays. Those overheads can be estimated analytically (see the concept of pSWP in Deliverable D5.5) or based on measurements.

For each one of the above-mentioned overhead estimations, we have various platform and application configurations. The platform can employ various network topologies to inter-connect the

cores such as mesh or tree, and has various memory delays depending where the application data is stored – either on-chip or off-chip memory. The provided application configurations include various input dataset sizes, level of parallelism, which results in a various number of threads and cores. As the one guesses, the search space for all possible platform and application configurations is large. Therefore, we decide to perform a few localized experiments to outline the best possible platform and application configurations towards WCET speedups.

2.3 Application WCET in the different application domains

Within parMERASA, we consider three basic ways of estimating the application WCET:

- Observed WCET – we run each one of the industrial applications multiple times and we record what has been the highest application execution time;
- Measurement-based WCET – we run an instrumented application with enabled tracing capabilities. Later, the RapiTime tool chain analyses those traces using the application flow-graph and estimates the application WCET.
- Static-based WCET – we analyse the application source code with the help of static tools such as OTAWA. For the static analysis, we need an accurate timing model of the architecture, bondable delays on the inter-thread synchronization primitives which affect the estimated WCET.

For the avionics and construction machinery industrial applications, the WCET analysis is performed at the level of the application. Contrary, for the automotive industrial application, the WCET analysis is performed at the level of the Runnable. Once the WCET of a Runnable is estimated (any of the afore-mentioned WCET approaches could be employed), the mapping tool is invoked to place those tasks on the multicore platform. Once placed, the application WCET is computed as a sum of the Runnable's WCET in the worst-case path.

Once the application WCET is estimated, we can easily derive the speedup and the efficiency of the parallelized application. We define the speedup as the ratio between the WCET of parallelized application to a sequential/single-core implementation. We define the efficiency as the ratio between the speedup of the parallelized application and the number of cores used by the application.

3 AVIONIC APPLICATIONS

In this section, we introduce the avionic application prototypes, composed of two types of applications – real and synthetic. The real applications are presented by 3D Path Planning (3DPP) and Stereo Navigation (StereoNav) algorithms. A detailed description of the initial parallelized version of both real avionic applications and the corresponding algorithms are available in Deliverable D2.1. A detailed description of the optimized versions of the avionics applications is available in Deliverable D2.5. Furthermore, for each real avionic application versions we consider two implementations. In implementation 1, we host the 3DPP and StereoNav on a “bare-metal” platform, i.e., the applications are executed without OS support. In implementation 2, we host the 3DPP and StereoNav onto the parMERASA software stack, i.e., an execution platform comprising of parMERASA simulator and the tiny avionic library. The experiences with the tiny avionic library are reported in Deliverables D4.5, D4.6, and D4.9. The synthetic avionic applications are presented by a simple tiny avionic benchmark, which exercise inter-/intra-partition communication.

The rest of the section is organized as follows. In Section 3.1, we introduce the initial version of the parallelized avionics applications. In Section 3.2, we provide the list of improvements of the optimized parallelized avionics applications. In Section **Fehler! Verweisquelle konnte nicht gefunden werden.**, we list our current status with respect of the COTS platform. We conclude with Section 3.3, which summarizes the efforts and outline the future directions.

3.1 Parallel applications

In this section, we introduce the preliminary parallelized versions of the avionics applications. In Section 3.1.1, we introduce the initial parallelized version of the 3DPP. In Section 3.1.2, we introduce the initial parallelized version of the StereoNav. In Section 3.1.3, we list the preliminary evaluation results and derive our outlook for the optimization phase.

3.1.1 3DPP – initial parallelization

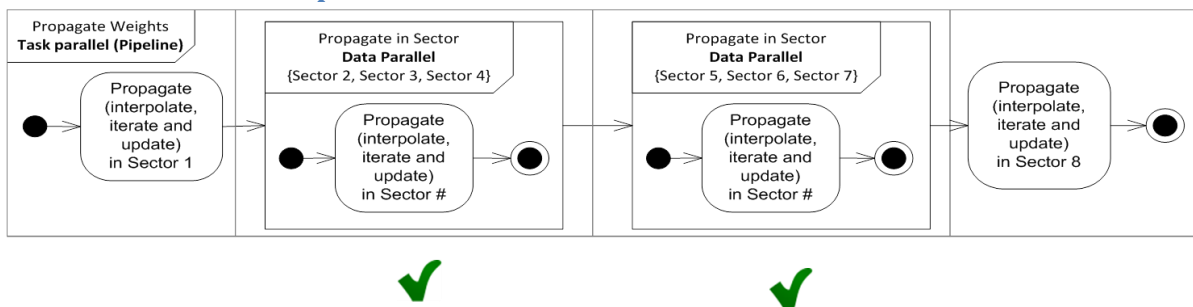


Figure 2 3DPP: initial parallel design patterns

In Figure 2, we introduce the preliminary parallelized version of the 3DPP with parallel design patterns. The application executes in a pipeline manner and employs a data-level parallelism. The list of application processing steps is explained in Deliverable D2.1. The preliminary parallelized version of the 3DPP is developed with conditional variables, which prove to be difficult for WCET analysis. Therefore, in the optimization phase, we plan to substitute the conditional variables with barriers.

3.1.2 StereoNav – initial parallelization

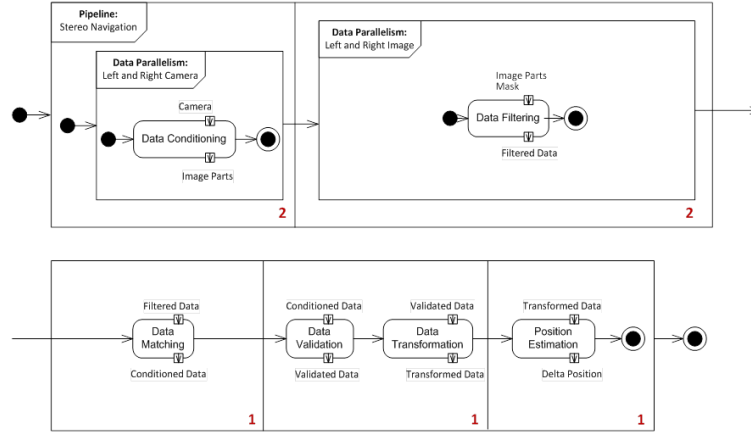


Figure 3 StereoNav: initial parallel design patterns

In Figure 3, we introduce the preliminary parallelized version of the StereoNav with parallel design patterns. The application employs two PDPs – pipeline and data level parallelism. Based on the preliminary version, we envision providing more application configuration exploiting different levels of parallelism using different number of threads. A detailed description of the application processing steps is provided in Deliverable D2.1. In the initial parallelized version of the StereoNav, we exploit a parallelism at the level of the images and frames, where each of the images is processed in parallel and the images are splatted further into frames. In the initial implementation, we do not provide a further parallelization of the most time consuming computation phase of the algorithm – pose estimation. Such a parallelization was left for the optimization phase. Furthermore, the initial application is implemented with conditional variables. Similarly to the 3DPP, we substitute the conditional variables with barriers.

3.1.3 Evaluation and outlook for optimization

In this section, we introduce the preliminary evaluation of the communication overheads and average case execution reduction for various application and platform configurations.

Communication overheads estimation

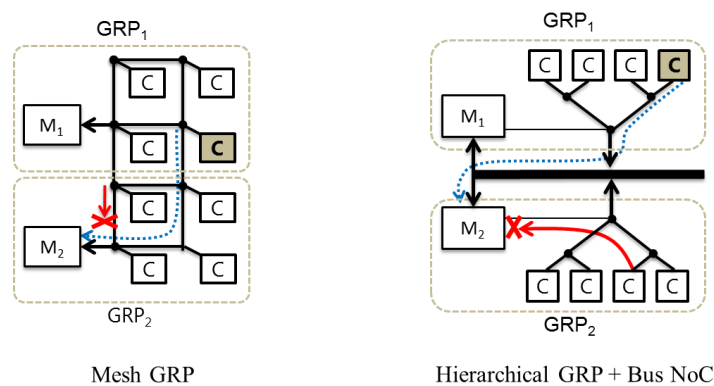


Figure 4 Sources of communication overheads during inter-partition communication

In Figure 4, we introduce two parMERASA platform configurations with respect to the deployed NoC – mesh and hierarchical interconnect + bus. By “M”, we denote a Memory and by “C” a computing

element, i.e., processor. A detailed description of the parMERASA platform and the accompanying terminology can be found in Deliverable D5.5.

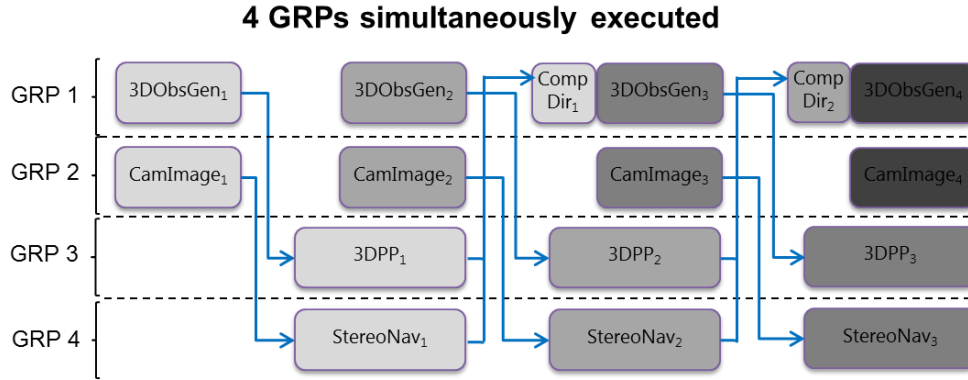


Figure 5 3DPP and StereoNav applications running in parallel on different GRPs. The data generated by ObsGen and GetImage at stage n is consumed by 3DPP and StereoNav respectively at the next stage $n + 1$.

We evaluate the communication overheads by measuring inter-partition delays in case more than one partition is running in parallel. We estimate the inter-partition delays by counting the application execution time variations during inter-partition communication. The application execution times are defined by the observed WCET of the target application. In Figure 5, we consider four different GRPs (a definition of GRP is available in Deliverable D5.5) running in parallel, where the arrows present the direction of data transfers. The GRP3 and GRP4 host the 3DPP and StereoNav applications. The former computes the path between the current vehicle (such as UAV) position and current goal position while avoiding obstacles. The latter is intended for airplane localization in case satellite navigation is unavailable. Moreover, three extra applications have been used to provide and process the data required by 3DPP and StereoNav: 3DObsGen, CamImage, and CompDir. The 3DObsGen and CompDir are mapped on GRP1, while CamImage runs in GRP2. The 3DObsGen provides a 3D grid obstacle map required by the 3DPP. The CamImage provides two images required by the StereoNav. Once the 3DPP and StereoNav complete their execution, the CompDir receives the directions computed by the 3DPP and StereoNav. The CompDir compare those directions and in case of a difference, CompDir suggests correction of the taken direction. In all cases, the data is transmitted using inter-partition communication requests. The four applications are executed following a software pipelining approach. The data generated by 3DObsGen and CamImage at stage n is stored in the memory of the GRP in which the pSWP runs, and it is accessed by the 3DPP and StereoNav respectively at the next stage $n + 1$. By doing so, four pSWP can execute in parallel.

We evaluate the inter-partition communication delays in two different ways - the measured variations of the application execution times (execution time is defined by the observed WCET) and the one estimated analytically. An analytical estimation is possible on the parMERASA platform, by computing the bound for the access latency on each shared resource. Keep in mind that the parMERASA platform features non-interfered parallel execution of partitions in case those partitions have only intra-partition communication.

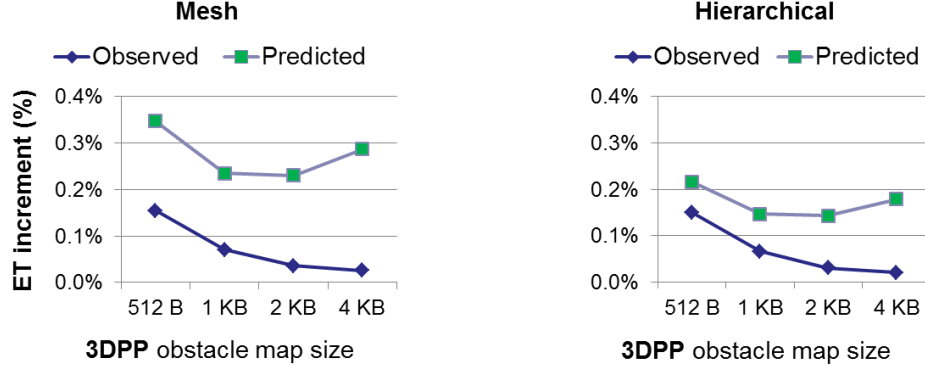


Figure 6 3DPP average case execution time increment due to inter-partition communication

In Figure 6, we introduce the experimental results for the observed and predicted (analytically estimated) execution time increments of 3DPP when 3DObsGen transmits obstacle maps of different sizes (512 bytes, 1 KB, 2 KB and 4 KB) for the next iteration of the 3DPP. Figure 6 (a) and (b) consider a mesh and a hierarchical NoC design respectively. In all cases, the size of the obstacle map considered for the baseline, observed and predicted execution times are the same.

The execution time of 3DPP increases due to interferences produced by inter-partition requests. The increment is very low (less than 1% for the observed and predicted) because the time required to process the obstacle map is much higher than the time required to transmit it. As accepted, the predicted execution time always upper bounds the observed execution time. In the mesh (Figure 6 (a)), the difference between the predicted and the observed execution time is higher than the difference seen in the hierarchical NoC (Figure 6 (b)). The reason is that in case of the hierarchical NoC, only the impact of the memory is considered, while in case of the mesh, the impact of the NoC and the memory is considered. When comparing the predicted execution times of both NoC designs, hierarchical NoC is 3% lower than mesh NoC design for all obstacle map sizes.

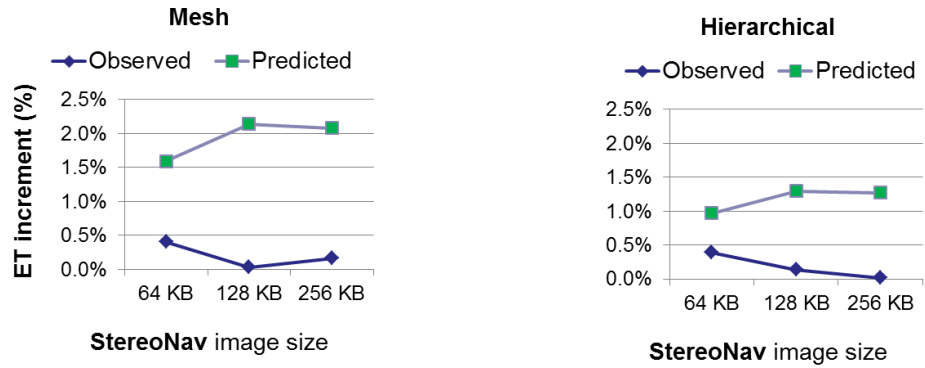


Figure 7 StereoNav average case execution time increment due to inter-partition communication

In Figure 7, we introduce the observed and predicted execution time increments of StereoNav when GetImage transmit images of different sizes (64 KB, 128 KB and 256 KB). Figure 7 (a) and (b) consider a mesh and a hierarchical NoC design respectively. In all cases, the size of the image considered for the baseline, observed and predicted executions times are the same.

Similarly to 3DPP, the execution time of StereoNav increases due to interferences produced by inter-partition communications. Again, by execution time, we refer to the observed worst-case execution of the application. The increment is very low, although higher than 3DPP (less than 2.5% and 1% for

predicted and observed execution times). As expected, the predicted execution time always safely upper bounds the observed execution time. When comparing the difference between the predicted and the observed execution time, StereoNav follows the same trend of 3DPP: the difference in case of the mesh NoC design (Figure 7 (a)) is higher than the one introduced by the hierarchical NoC design (Figure 7 (b)). When comparing the predicted execution times of both NoC designs, hierarchical NoC execution time is on average 10% lower than for the mesh NoC design for all image sizes.

In the results shown in Figure 6 and Figure 7, there is a relation between the amount of data transmitted through inter-partition communication and the amount of computation required. For example, if GetImage transmits 256 KB, StereoNav processes the same amount of data from previous stage, resulting in a small execution time increment. However, such a relation between applications may not occur. Figure 6 shows the observed and predicted execution time increase when running in parallel 3DPP operating on a fixed size obstacle map of 512 bytes. Figure 7 (a) and (b) consider a mesh and a hierarchical NoC design respectively.

parMERASA platform optimizations towards lower WCET times

As we introduced before, we improve the application performance by adding local memory (e.g. caches) to each of the cores on the parMERASA platform. Once the caches are introduced, we need to make sure that the data within the caches is coherent throughout the cores. For this purpose, we employed the ODC2 [34]. To demonstrate that the ODC2 can bring us to a lower WCET and higher performance, respectively, we measure the execution time for the 3DPP application.

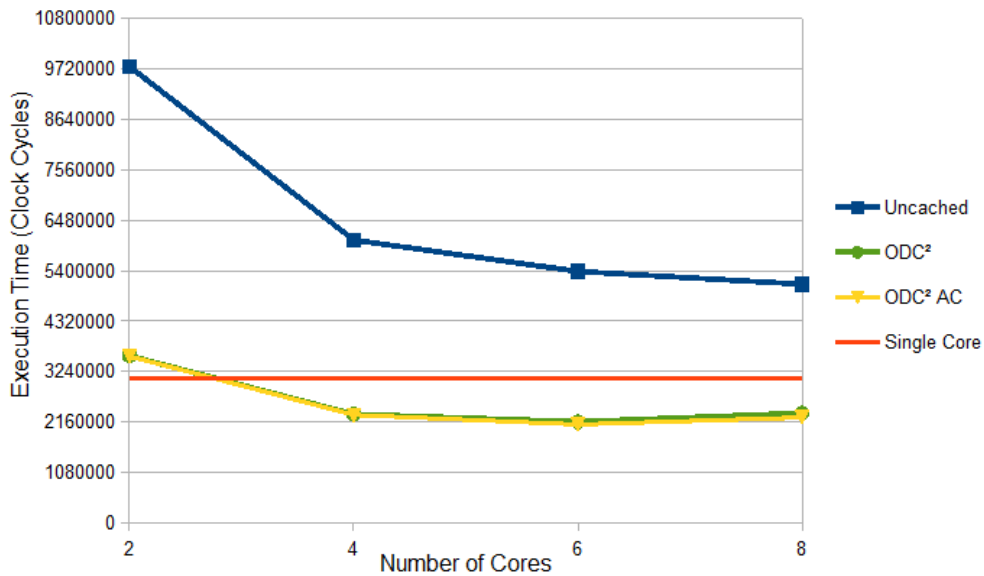


Figure 8 3DPP average case execution time with ODC2 cache

In Figure 8, we introduce a 3DPP execution time (measured as observed WCET) variations with respect to the number of cores and ODC2 policy (see Deliverable D5.5). We compare uncached accesses for multicore platform instance, cached single-core, and two configurations of the ODC2 – with and without address checking, respectively. We employ the single-core instance as a reference point.

As the numbers in Figure 8 suggests, the time-predictable ODC2 delivers significant performance speedup (up to 25 % for 4 cores) compared to cached single-core execution. Therefore, we conclude that predictable caches or scratchpads in multi-core systems are feasible architecture optimizations for lowering the WCET of parallelized applications on multi-core systems.

Summary

Based on the evaluation results and preliminary application analysis, we suggest the following list of improvement. Since both applications are implemented with conditional variables, our preliminary analysis with static analysis tools (such as OTAWA) demonstrates that synchronization idioms such as conditional variables deliver a conservative estimation. In the optimized version of the applications, we envision to substitute the conditional variables with barrier synchronization idioms. Furthermore, our analysis suggests that a single shared data memory to all cores prove to be a bottleneck and increasing the number of cores/threads results in a limited speedup. Another potential direction for improvement is to increase the application efficiency by reducing the number of idle cores and conglomerating multiple threads to a single core, even if only data-level parallelism is available.

3.2 Optimized applications

In this section, we introduce the optimized version of the parallelized avionics applications. In Section 3.2.1, we introduce the optimized version of the 3DPP with the help of PDPs. In Section 3.2.2, we introduced the initial parallelized version of the StereoNav.

3.2.1 3DPP – optimized parallelization

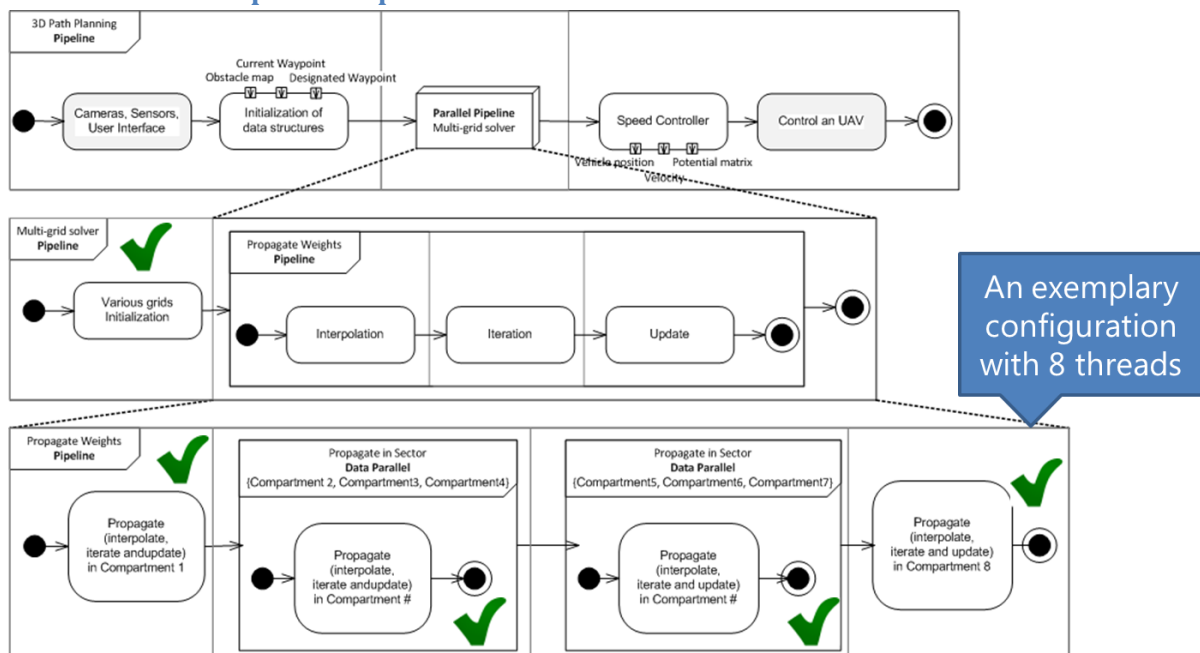


Figure 9 3DPP: optimized parallel design patterns

In Figure 9, we introduce the optimized 3DPP presented with PDPs. For the optimized 3DPP application, we perform or envision the following list of improvements with respect to the initial parallel version:

- We substitute the conditional variables with barriers synchronization mechanism;

- We experiment with these two synchronization primitives on COTS and parMERASA platform;
- We improve the application coding style, so the OTAWA estimations to be less conservative;
- We improve the tiny avionic library to support the new memory map, i.e., each core has scratchpad memory and the communication among cores is established through shared memory. A detailed description is available in Deliverable D4.9.
- We envision to reduce the number of employed cores for the 3DPP computation down to $n/2$ where n is the maximum number of threads in the application. Such a conglomeration of the data parallel threads is possible, since they are not ready at the same time. As a result, we expect to boost the efficiency;
- We provided 5 application configurations - 1 thread (sequential), 2 threads, 4 threads, 8 threads and 16 threads configuration. Please note that 16 thread version is not provided in the prototype Deliverable D2.5;
- We envision to compare the observed WCET, RapiTime WCET, and OTAWA WCET estimations;

In Deliverable D2.5, we provide a detailed description how to execute the 3DPP optimized configurations.

3.2.2 StereoNav – optimized parallelization

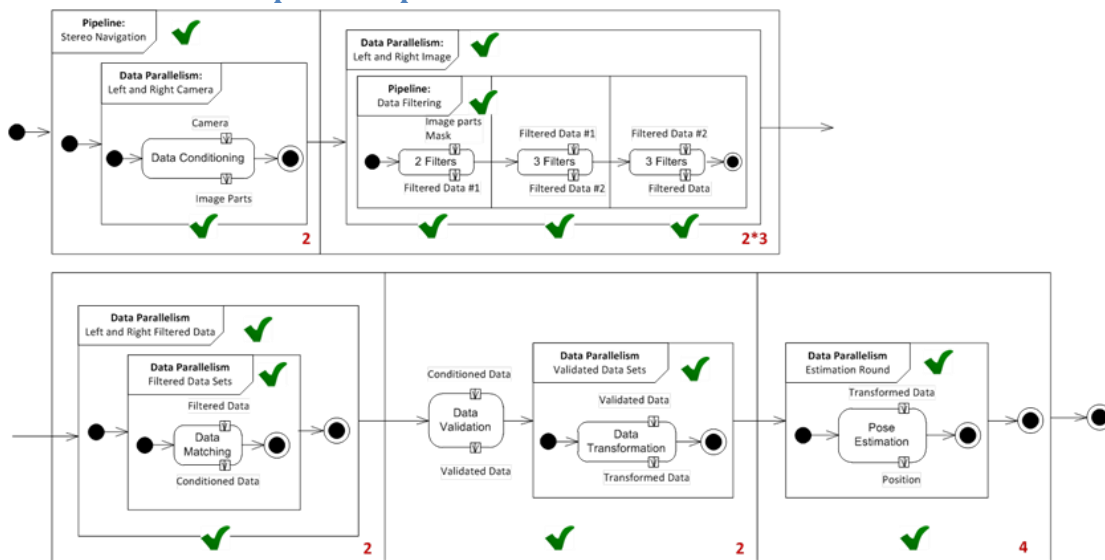


Figure 10 StereoNav: optimized parallel design patterns

In Figure 10, we introduce the optimized StereoNav application presented with PDPs. For the optimized StereoNav application, we perform (or envision) the following list of improvements with respect to the initial parallel version:

We introduce 6 application configurations with varying number of threads from 1 to 48 threads. The configurations are as follows: 1 thread (sequential), 7 threads, 2 configurations with 12 threads, 14 threads, 16 threads, and 48 threads. Please note that we implemented the 48 threads version on a COTS machine only. We exploit all levels of parallelism of the application at the level of images, tiles, convolutions, points, etc. More specifically, we apply software pipelining technique and data-level parallelism on the most computation intensive part of the StereoNav – the Feature Extraction phase.

Such a parallelization has been introduced with the help of pipelining and data level parallelism, where the processing of convolutions, images and tiles is done in parallel.

The software pipelining is delivered by running the application in 7 main steps: rectification, tile splitting, feature extraction, feature matching, circular check, reprojection, and robust pose estimation. The synchronization between two adjacent software pipeline stages is delivered by 2 barriers. The 2 barriers guarantee the data integrity for un-balanced pipeline stages, which communicate to each other. A problem might occur in case a shorter pipeline stage overwrites a value, while it is still not consumed by a longer pipeline stage. In Figure 11 to Figure 16, we provide how each one of the StereoNav applications scenarios looks like with respect to the applied parallelism and number of threads.

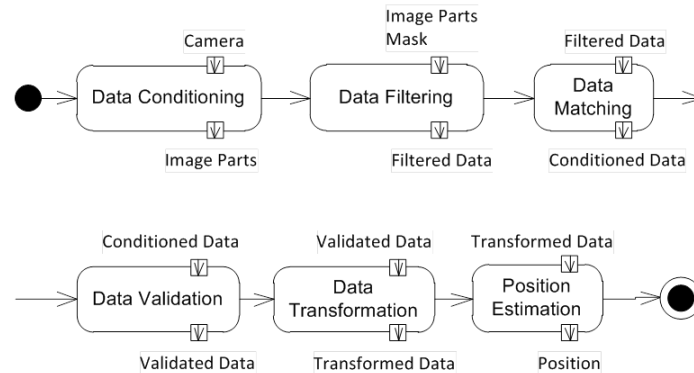


Figure 11 StereoNav - 1thread, sequential execution

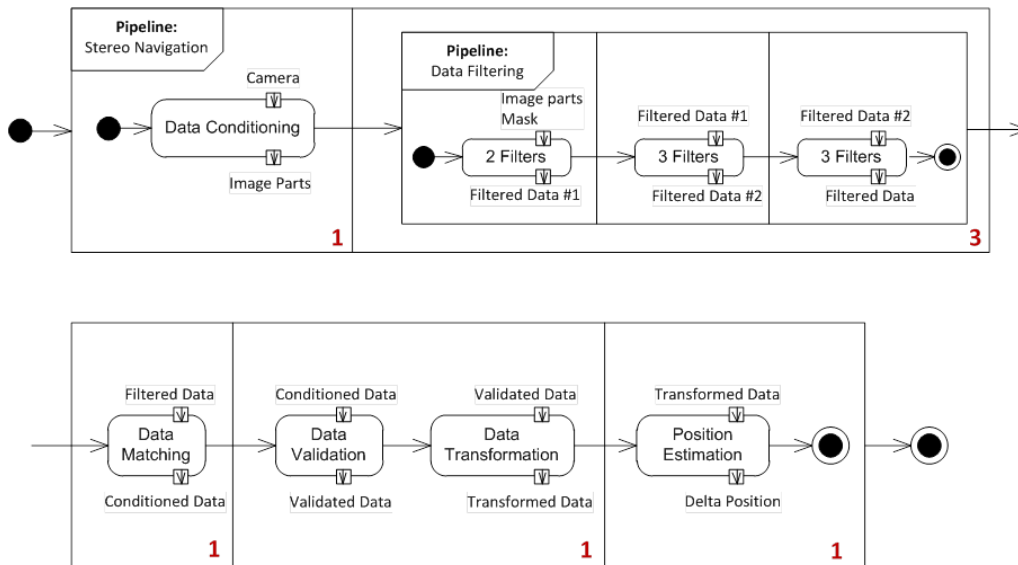


Figure 12 StereoNav - 7 threads, pipeline execution

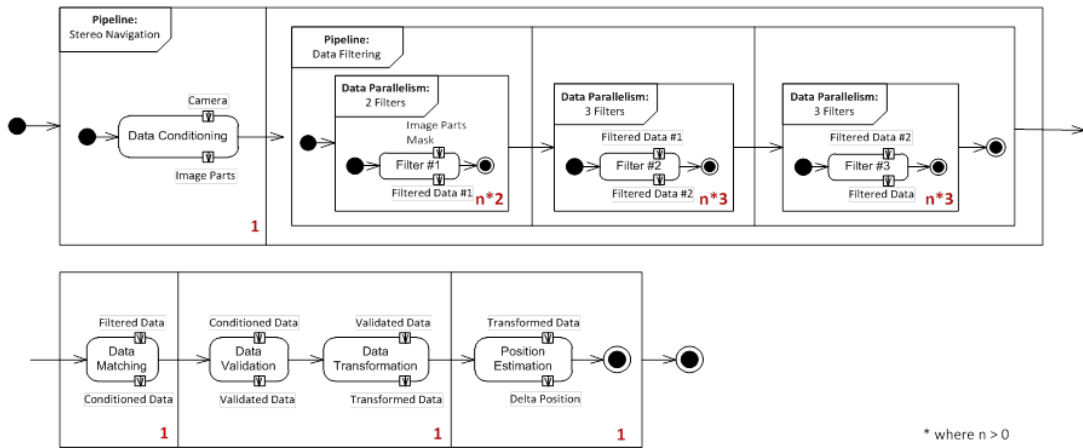


Figure 13 StereoNav - 12 threads, convolution parallelism

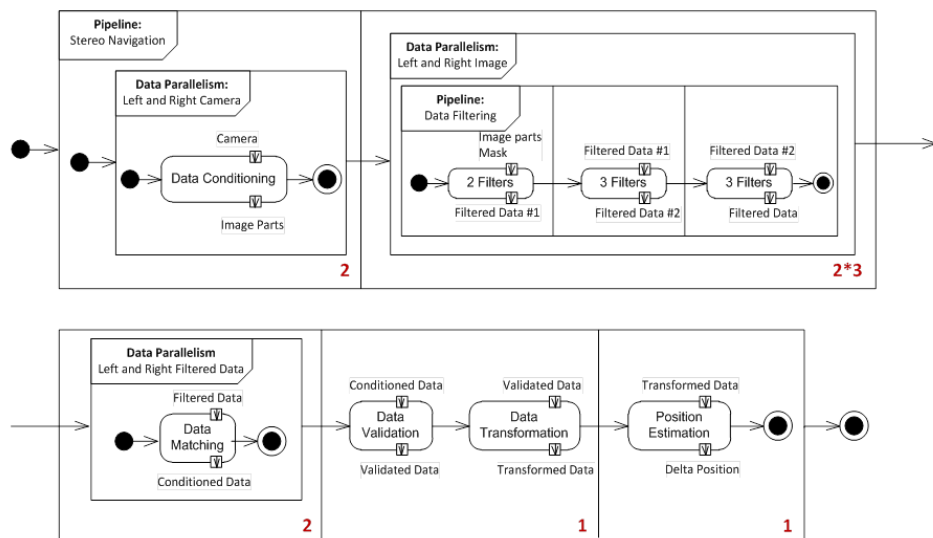


Figure 14 StereoNav with 12 threads, camera-level parallelism

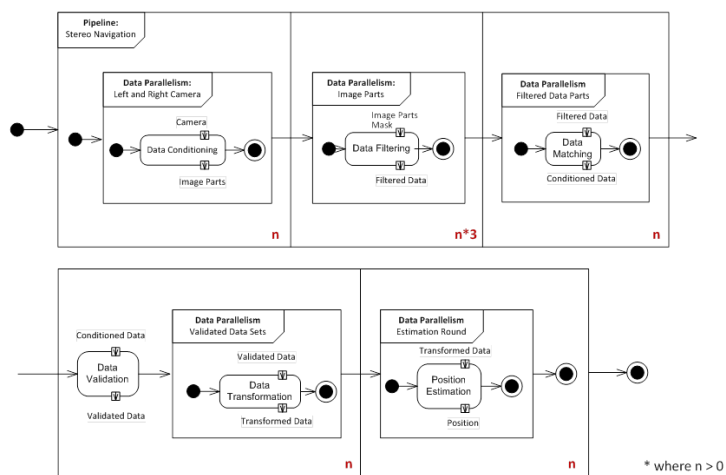


Figure 15 StereoNav - 14 threads, data (image tile) parallelism

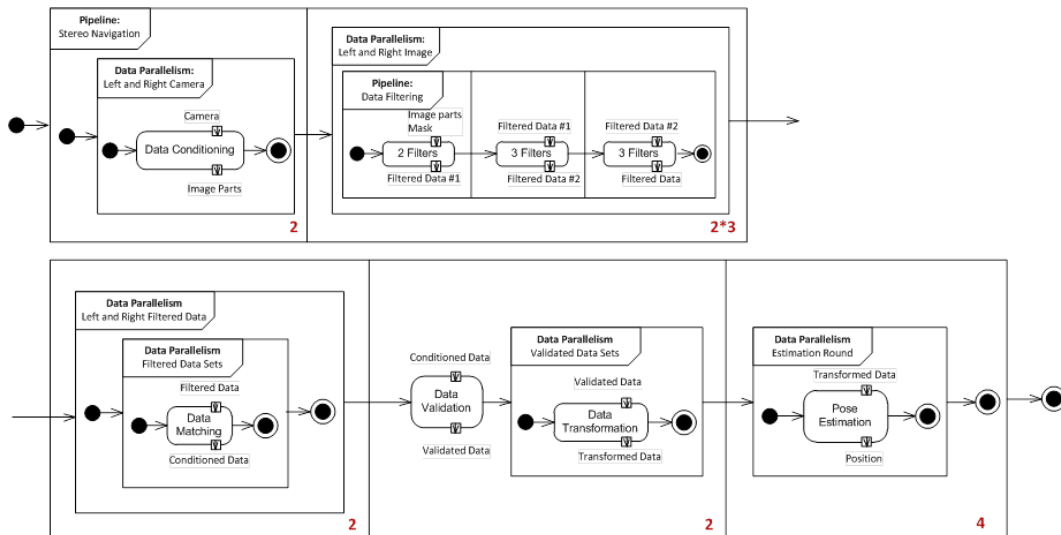


Figure 16 StereoNav - 16 threads, exploit all levels of parallelism

In the coming months, we envision to compare the observed WCET, RapiTime WCET, and OTAWA WCET

In Deliverable D.2.5, we provide a detailed description how to execute each one of the StereoNav optimized configurations.

3.3 Conclusions and outlook

In this section, we provided a description on the initially parallelized and the further optimized two avionics applications – 3DPP and StereoNav, respectively. For the initially parallelized applications, we evaluated the communication overhead in terms of inter-partition interferences and we analysed the potential platform improvements with ODC2 towards lower WCET. We provided an optimized version of the avionics applications with various number of application configurations and we improved the synchronization primitives.

For the final evaluation phase, we envision providing the final study and conclusions for the WCET on parMERASA platform – observed, estimated by RapiTime and OTAWA. The COTS platform will include a study on the observed WCET.

4 AUTOMOTIVE APPLICATIONS

This section describes the parMERASA parallelization approach tailored to automotive applications and the parallel automotive application prototype, a real diesel engine management system.

The AUTomotive Open System ARchitecture (AUTOSAR) standard provides a uniform understanding of automotive software across the whole industry. Therefore, the terminology is used throughout this section for comprehensibility. For unfamiliar readers a reference is made to [36]. Currently, AUTOSAR does not specify mechanisms to execute Runnables of the same task on different cores. Therefore, tasks of the single core configuration are denoted as *s-task*. They can only execute on the same processor they were mapped to. The tasks that are generated by the parMERASA parallelization approach are called *p-task*. They can be split to be executed on different cores.

This section is organized as follows. Subsection 4.1 provides an overview about the parallelization process and progress until month 24 of the parMERASA project. Opportunities for optimizations are outlined. Afterwards, in subsection 4.2, the optimized parallelization process and application are described. The section ends with conclusions and ideas for future work until the end of the project.

There is one controlling application that all road vehicles have in common – the engine management system (EMS). The EMS is one of the most computational intensive applications within a road vehicle. The prototype is based on a diesel EMS, which contains 12 time-triggered (TT) *s*-tasks and one event-triggered (ET) *s*-task. The TT *s*-tasks execute with a fixed period and the activation of the TT *s*-tasks is synchronized to a fundamental period of 1ms. The ET *s*-task is executed depending on the position of the camshaft. That means there are two time scales within the EMS that drift against each other. This is a big issue for the implementation, because these *s*-tasks can interrupt each other and this has to be considered in all other parts of the implementation. The EMS uses a pre-emptive scheduler with fixed priorities in the single core implementation. That means, whenever multiple *s*-tasks are activated to the same tick, priorities decide on the execution order. The camshaft synchronous *s*-task has the highest priority to guarantee low response times. Each *s*-task contains a sequential list of Runnables. Typically, a Runnable is mapped to one *s*-task only. For a more detailed description of the application a reference is made to Deliverable 2.1.

4.1 Parallel application

This section describes the first parallel implementation of the application before optimization. The parallelization uses the information of the single core scheduling.

Runnables communicate with each other through global variables, which are typically stored in an on-chip shared memory. Consequently, *s*-tasks are not independent, because the Runnables inside different *s*-tasks communicate with each other. For that reason, we decided to exploit the parallelism inside each *s*-task – *intra-task parallelism* – first.

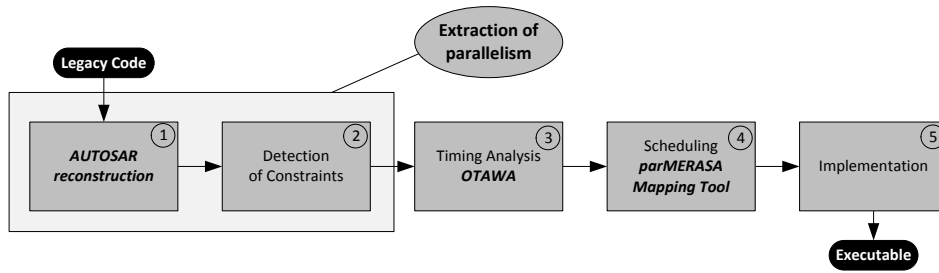


Figure 17: Parallelization process for automotive software

The EMS was parallelized in five steps, as shown in Figure 17. First the legacy application is reconstructed as an AUTOSAR compliant model (1), because communication is explicitly expressed. Then, the Runnable Dependence Graph (RDG) is derived for each s-task (2) and the WCET for each Runnable is estimated using *OTAWA* (3). Afterwards, each s-task is parallelized by the *parMERASA mapping tool* (4) into a p-task, based on the RDG and the Runnables WCET. An absolute schedule is defined, in which the Runnables of a former s-task are mapped to the cores of the target architecture. Here, an execution window is reserved for every Runnable. Thus, no further synchronization mechanisms are required. Finally, the schedule is implemented to run on top of the tiny automotive RTE (Deliverable 4.4) (5). Each of these steps is explained in more detail in the following.

4.1.1 AUTOSAR reconstruction

The static structure of the EMS is formally described in the first step of parallelization. The AUTOSAR Virtual Function Bus (VFB) model is used, because it is the common automotive standard and it explicitly expresses communication. The reconstruction of the source code as AUTOSAR model is done by static analysis of the source code. The modular structure (file hierarchy and naming) of the application is used to reconstruct Software-Components as container for Runnables. S-tasks can be identified automatically by their name. The Runnables are identified by analysis of the s-task's call tree. It is been distinguished between two types of Runnables:

- a) **Cyclic Runnable** – is called directly from the s-task body at a specific point in time
- b) **Server Runnable** – is a non-pure function called from several other Runnables.

Global variables are only written by the modules that define them. Thus, data exchange through global variables is represented as *Sender/Receiver communication* between Software-Components. The former module is transformed into an AUTOSAR Software-Component with a *Sender/Receiver provide port*. A *Sender/Receiver Require Port* is added to all modules that read the global variable. In the implementation read accesses are transformed into an RTE_read-API call. A module function is converted into a *Server-Runnable* of a Software-Component with *Client/Server Provide Port*, if it is directly called by several other Runnables, unless it is a pure function. The Software-Component containing the callee will get a *Client/Server Require Port*.

Figure 18 shows an example for the reconstruction as AUTOSAR compliant model for a part of the diesel EMS. Each box denotes one Software-Component (a former module). The arrows describe Sender/Receiver communication between the ports of the Software-Components.

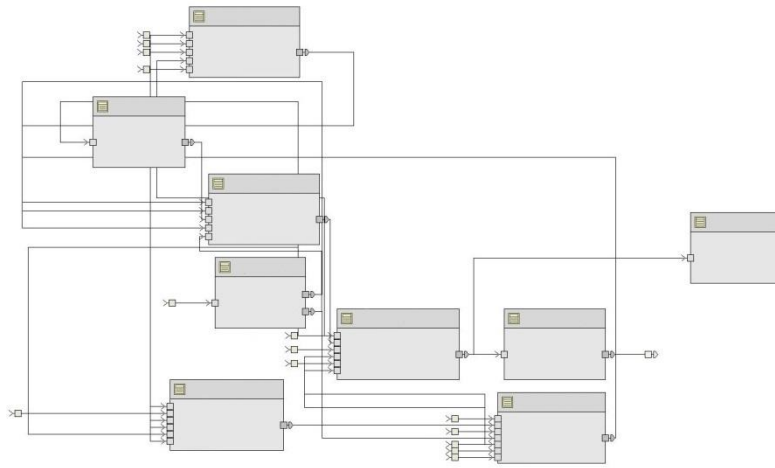


Figure 18: Example for AUTOSAR reconstruction. The figure shows a subset of a diesel EMS.

4.1.2 Detection of Constraints

Data dependencies between instructions impose *precedence constraints (PCs)* that force a sequential execution. Therefore, PCs must be identified in the application to allow an efficient parallelization. PCs can be defined at any level. For example, automatic parallelizing compilers (see [37]) define them at instruction level. *High Performance Computing (HPC)* approaches define them at task level (see [38]). Automotive software comprises several hundred to thousand Runnables, whereby each one comprises only few instructions. Thus, PCs are identified between Runnables. The detection of PCs within one s-task comprises the following steps:

1. The memory access of each instruction is analysed and aggregated per Runnable.
2. The control flow inside the s-task is detected.
3. A PC from Runnable A to Runnable B is identified, whenever a flow-dependence exists from A to B. (A flow-dependence exists, if A writes to the same memory location that is read by B and there is a valid control flow path from A to B.)
4. The PCs are described as a Runnable Dependence Graph (RDG) $G_{RDG} = (V, E)$, where each node $v \in V$ denotes a Runnable and each edge $e \in E$ denotes a precedence constraint.

The detection of memory accesses in the first step is a challenging problem, especially if aliases are used. Nevertheless, research has been conducted in this field for decades and most compilers are able to detect data dependencies with a high accuracy. The detection of the control flow in the second step is possible only within one s-task, because the execution order of Runnables within one s-task is static. Figure 19 shows the RDG for the 20ms periodic s-task.

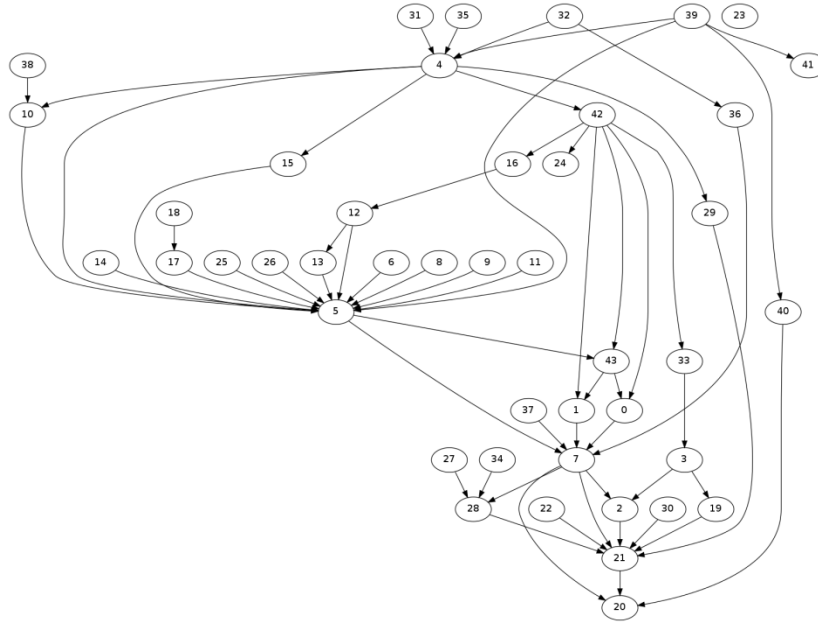


Figure 19: Runnable Dependence Graph for the 20ms periodic s-task.

4.1.3 Timing Analysis

The purpose of this step is the analysis of the timing characteristics of the application, including communication and synchronization mechanisms. Here, OTAWA is used to derive WCET estimates. It was extended to support the architecture of the parMERASA processor. That means, trustworthy WCET estimates can be calculated. Each Runnable as well as the synchronization mechanism were analysed with OTAWA. The WCET for different setups on 2, 4, and 8 cores has been calculated. Moreover, the overestimation and trustworthiness have been investigated by simulations in the parMERASA simulator.

4.1.4 Scheduling

To achieve a predictable system, time windows are defined for the execution of a Runnable. This is analogous to the paradigm of Logical Execution Times (LETs) described in [39]. Therefore, an absolute schedule is defined for each p-task by the parMERASA Mapping Tool. The RDG of the s-task (defined in step 2) and the WCET for each Runnable (calculated in step 3) are required as input. The parMERASA Mapping Tool exploits the *intra-task parallelism* of AUTOSAR applications by allocating Runnables to cores. The Mapping Tool is based on a partitioned scheduling variant of the best-fit decrease heuristic, which prioritizes Runnables with higher combined utilization. Thus, the longest chain of dependent Runnables is allocated first. See deliverable D2.3 for further details.

TASK1ms		Advance until
Core0(totalU:0.011904)		
[Core0]	R1	17264
[Core0]	R2	20020
[Core0]	R3	35158
[Core0]	R4	35712
Core1(totalU:0.011848)		
[Core1]	R5	450
[Core1]	R6	3594
[Core1]	R7	5532
[Core1]	R8	6524

[Core1]	R9	7352
[Core1]	R10	7794
[Core1]	R11	8172
[Core1]	R12	8550
[Core1]	R13	8880
[Core1]	R14	9036
[Core1]	R15	9144
[Core1]	R16	9252
[Core1]	R17	9360
[Core1]	R18	9468
[Core1]	R19	9528
[Core1]	idle	35158
[Core1]	R20	35544
Core2(totalU:0.011848)		
[Core2]	idle	35158
[Core2]	R21	35544

Table 1: Example for the Mapping Tool output of the 1ms p-task.

Table 1 shows the exemplary Mapping Tool output for the 1ms cyclic p-task. In this example a GRP with 4 cores was assumed as target architecture. The speed-up for this mapping is 1.29 for every execution of the 1ms p-task. Table 2 shows the speed-ups for all other p-tasks that have been parallelized with the Mapping Tool. The overall application speed-up is 1.7.

P-task	sequential	parallel	speed-up
1ms	46012	35712	1.29
4ms	123688	67180	1.84
5ms	3490	3490	1.00
8ms	107174	37712	2.84
16ms	462116	209210	2.21
20ms	46110	26982	1.71
32ms	116086	58666	1.98
64ms	45688	23880	1.91
96ms	40816	18684	2.18
128ms	130578	88210	1.48
1024ms	39596	26786	1.48

Table 2: Speed-up per p-task execution on a 4-core parMERASA processor.

4.1.1 Implementation

The final step (5) of the parallelization is the implementation of the parMERASA Mapping Tool output. Each p-task is now implemented on multiple cores, which are synchronized to the ticks of a global conceptual clock. The Runnable mapped to core 0 in the previous example are transformed to:

```
task_1ms_core_0() {
    R1();
    advance(17264);
    R2();
    advance(20020);
    R3();
    advance(35158);
    R4();
}
```

```

    advance(35712);
}

```

All other cores are transformed accordingly. The advance instruction is a delay mechanism provided by the tiny automotive RTE. It delays the execution until an absolute point in time. The point in time is the start time + the WCET of the Runnable, which is executed before. If the WCET has already been reached the function returns immediately. Simulations showed a low overhead for the synchronization mechanism, see Table 3.

Cores	I-cache	D-cache	Immediate return	Delay	Maximum
2	perfect	4 - 256 KB	104	164	164
2	perfect	perfect	104	164	164
2	4 - 256 KB	perfect	524	136	524
4	perfect	4 - 256 KB	104	100	104
4	perfect	perfect	104	100	104
4	4 - 256 KB	perfect	824	112	824
8	perfect	4 - 256 KB	104	125	125
8	perfect	perfect	104	125	125
8	4 - 256 KB	perfect	1469	109	1469

Table 3: Overhead for the execution of the advance function on different processor and memory setups.

We observed that the size of the I-cache has a big influence on the overhead, if the advance instruction has to return immediately. The advance instruction must return immediately, if the WCET of the Runnable has been reached and no further delay is required. Nevertheless, the total overhead incurred by the advance instruction is low, as shown in Table 4. The total overhead of the application is 0.32%.

Task	overhead
1ms	0.2%
4ms	0.2%
5ms	0.3%
8ms	0.8%
16ms	0.6%
20ms	3.3%
32ms	0.3%
64ms	0.2%
96ms	0.5%
128ms	0.2%
1024ms	0.1%

Table 4: Synchronization overhead per task based on WCET on a quad-core processor with 256 KB I-cache.

4.1.2 Potential for Optimization

The example in Table 1 also shows that only 3 out of 4 cores are used, which is undesirable. The reason for this result is a high number of data dependences among Runnables. The critical path (maximum combined utilization) can prevent an efficient usage of hardware resources. Figure 20 illustrates this problem for another p-task. The critical path (R1, R2, and R3) has a very high execution time compared to all other Runnables in the p-task. For this reason, the optimization focused on the exploitation of such unutilized cores.

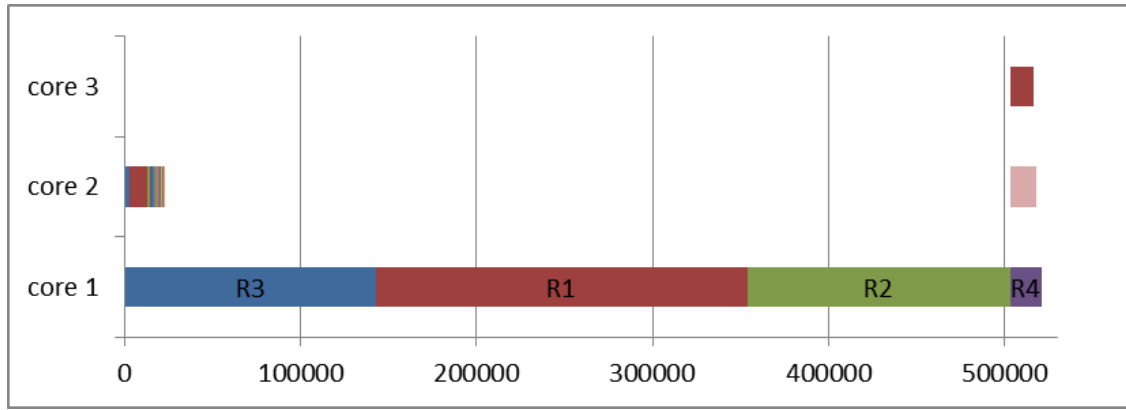


Figure 20: Critical path leads to low utilization.

4.2 Optimized applications

This section describes the first level optimization of the parallel EMS. As explained before, the core utilization can be bad, if the critical path within one p-task is very long. Thus, we propose three different optimizations.

The first optimization is the usage of supertasks (task-interleaving), in which RDGs of different p-tasks are merged and scheduled by the parMERASA Mapping Tool.

Another opportunity to improve the utilization is the parallelization of Runnables to decrease the length of the critical path by parallelization of Runnables. Here, the pattern-supported parallelization approach is applied (see Section 1.2).

The third possibility for optimization is *inter-task parallelism*, which exploits opportunities for parallel execution of p-tasks. A parallel execution of p-tasks is possible, if a communication mechanism allows simultaneous access to shared resources without corrupting them. Therefore, precedence constraints between Runnables must be classified.

Figure 21 shows the optimized parallelization process. Two steps were added: the Pattern-supported Parallelization (4) and the Classification of Constraints (3). This process now makes use of the PDPs to exploit intra-Runnable parallelism. Moreover, the classification of constraints allows the usage of the clustered architecture of the parMERASA processor, because p-tasks can execute in parallel.

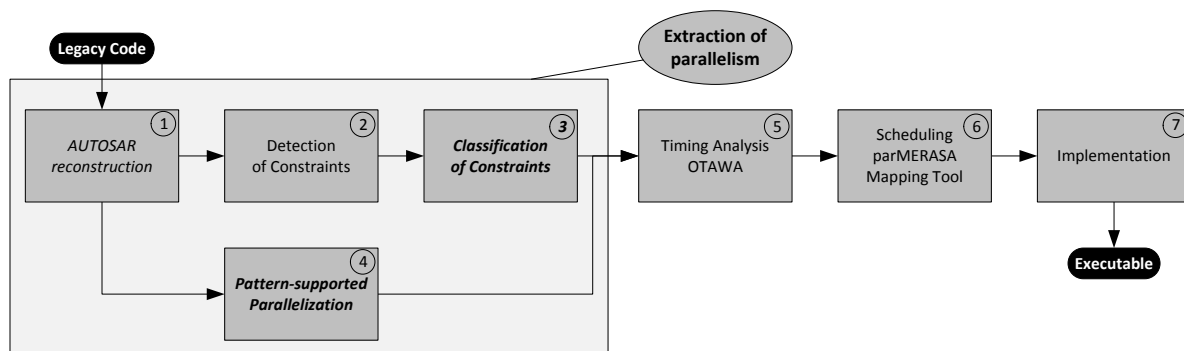


Figure 21: Optimized parallelization process.

4.2.1 Supertasks

As explained before, TT p-tasks are scheduled to the ticks of a fundamental clock. A supertask is composed by the Runnables of p-tasks that are scheduled to the same tick. This makes it possible to

exploit unutilized cores, as shown in Figure 20. The speed-up gained by supertasks is shown in Table 5. The overall application speed-up with the presented supertasks is 2.35.

Supertask	sequential	parallel	speed-up
1ms_4ms	169700	67180	2.53
1ms_5ms	49502	35712	1.39
1ms_4ms_8ms	276874	70174	3.95
1ms_4ms_8ms_16ms	738990	236422	3.13
1ms_4ms_5ms_20ms	219300	67180	3.26
1ms_4ms_8ms_16ms_32ms	855076	252164	3.39
1ms_4ms_8ms_16ms_32ms_64ms	900764	263576	3.42
1ms_4ms_8ms_16ms_32ms_96ms	895892	262322	3.42
1ms_4ms_8ms_16ms_32ms_64ms_128ms	1031342	294026	3.51
1ms_4ms_8ms_16ms_32ms_64ms_128ms_1024ms	1070938	303690	3.53

Table 5: Speed-up per supertask execution on a 4-core parMERASA processor.

4.2.2 Pattern-supported Parallelization

The pattern-supported Parallelization approach has been applied to one Runnable of an engine management system, see Figure 22. The implementation is outstanding and planned for the final phase of the project. For the final phase of the project it is planned to apply the approach to other Runnables, especially the Runnables in the critical path of p-tasks.

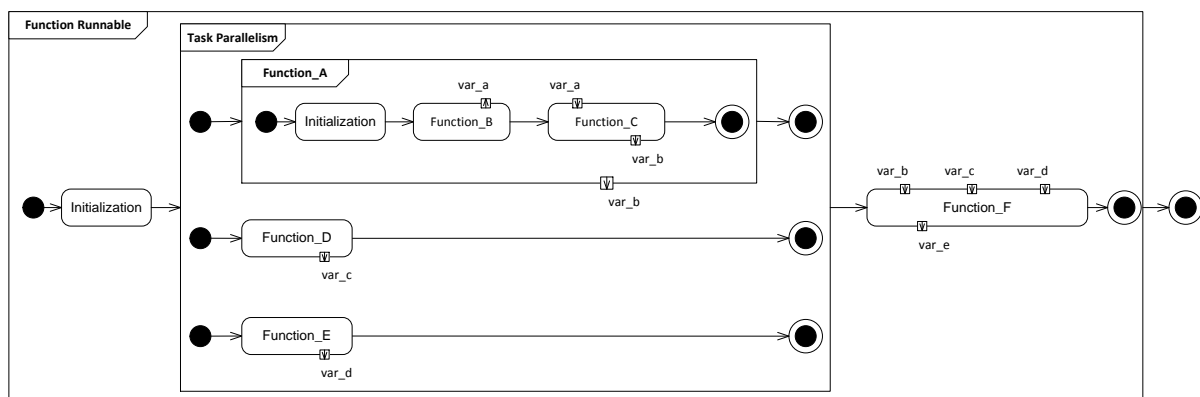


Figure 22: Example of Runnable parallelization with PDPs.

4.2.3 Classification of Constraints

The scheduling of automotive single core software implies that communication between s-tasks (Runnables) is asynchronously by nature. For that reason, a method to classify precedence constraints in *strong* and *weak* is proposed in order to reduce the number of constraints. The former one describes constraints that must be respected during the scheduling, whereas the others must not be respected. This allows a parallel execution of Runnables that would have to execute in a sequential order otherwise. However, the data flow is kept by replacing it with wait-free communication through a buffer. A precedence constraint between two Runnables of different s-tasks is classified as *weak* if, for example:

1. The producing Runnable executes much more frequently than the consuming Runnable. (e.g., all Runnables in 1ms s-task produce output four times before they are consumed by Runnables of 5ms s-task),

2. The time scales of producing Runnable and consuming Runnable drift against each other (e.g., Runnables of the ET s-task vs. TT s-tasks)

In a first experiment tasks were grouped and mapped to clusters as follows:

- a) 1ms p-task on cluster 1,
- b) 4ms and 8ms p-tasks on cluster 2,
- c) 16ms, 32ms, 96ms, 64ms, and 128ms p-tasks on cluster 3,
- d) 1024ms p-task on cluster 4,
- e) 5ms and 20ms p-tasks on cluster 5.

For this mapping we calculated a total application speed-up of 3.59. The integration on different clusters is still on-going and planned to be completed in the final phase of the project.

4.2.4 Optimized Memory Layout

In addition to the optimization of the application, the memory layout has been investigated mitigate the access cost to shared memory. We found that the instruction segment of the application (ca. 600 KB) is several times bigger than the data segment (ca. 200 KB). For that reason, we performed an experiment with OTAWA. The WCET for several cache configurations was calculated, as shown in Figure 23. The red bars, denoted as I-cache (perfect D-cache), describe configurations with a perfect D-cache, where the size of the I-cache was changed (x-axis). The blue bars describe configurations with perfect I-cache, where the size of the D-cache changes. The results confirmed the assumption that the instruction cache has a bigger impact on the WCET than the data cache. These findings suggest that a scratchpad can reduce the WCET of the application. Therefore, we investigate the effect of a core local scratchpad for instructions until the end of the project. Whether this result holds for other applications will depend on the properties of the application.

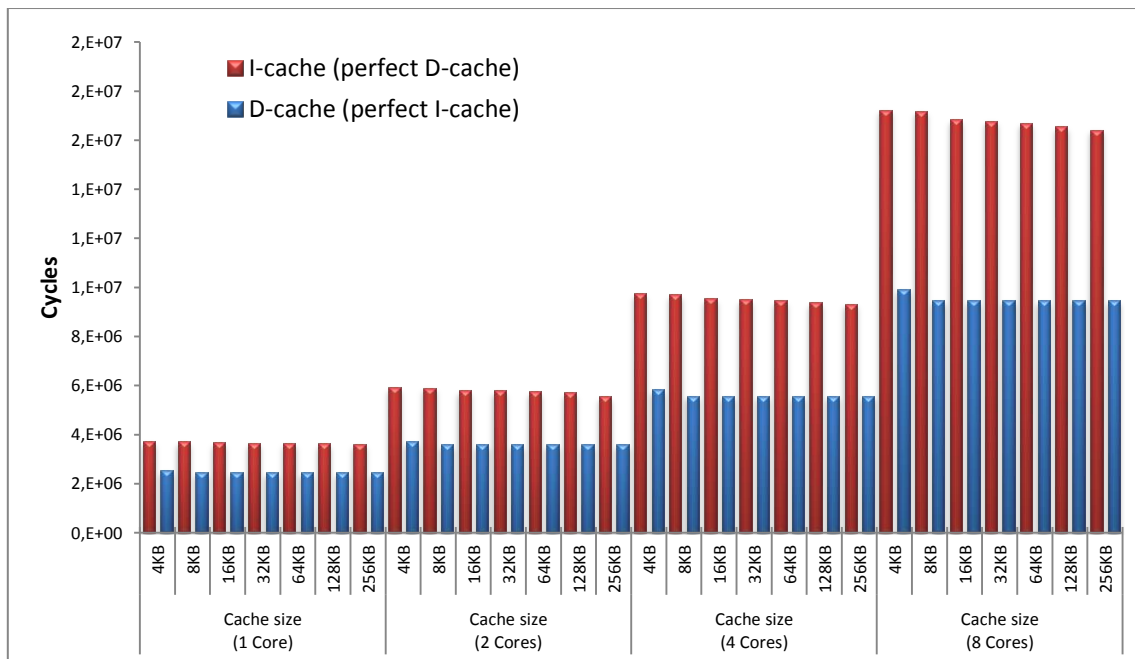


Figure 23: Impact of the cache on the WCET. The figure shows the sum of WCETs for all Runnables in the application.

4.3 Conclusions and outlook

In this section a parallelization approach for automotive legacy software was presented. A first prototype has been developed until month 24 of the project. Based on this, optimizations have been

proposed. This includes the formation of supertasks, the parallelization of Runnables with PDPs, and the classification of precedence constraints. The process has been refined and the first optimization (with supertasks) improved the speed-up from 1.7 to 2.35. The process can extract parallelism at Intra-Runnable level (with PDPs), at intra-task level (parMERASA Mapping Tool), and at inter-task level (classification). The impact of the cache was investigated and an adaption of the memory layout was proposed.

First simulation studies have been conducted on the parMERASA simulator (WP5) using the tiny automotive RTE (WP4). More details can be found in Deliverable D2.5. Feedback has been provided to WP3 (Tools), WP4 (tiny automotive RTE), and WP5 (Simulator).

4.4 Future Work

Future work concerns the optimization of the application. First, the implementation of Runnables that have been parallelized with PDPs is done. Additionally, Runnable with a high load will be parallelized with the PDPs. This activity will be supported by UAU. This will also require an adaption of the scheduling, because Runnables can allocate more than one core. Second, inter-task parallelism is investigated (classification of precedence). This requires the implementation of a buffered communication mechanism and a refactoring of the EMS. Third, the optimal size of scratchpad and cache is investigated. In parallel the tools provided by Rapita will be used to collect runtime traces of the execution. Furthermore, it is planned to check temporal execution constraints. Eventually, a feedback will be given.

5 CONSTRUCTION MACHINERY

5.1 Parallel application

The BAUER application being parallelized in parMERASA is the software running on the electronic control unit of a foundation crane. It executes the BAUER Dynamic Compaction (BDC), see details in D2.1, chapters 5.2.1 - 5.2.3. Unfortunately, there is no sequential version for the parMERASA simulator. It would require the implementation of a complex scheduler that is closed-source property of Sensor-Technik Wiedemann (STW, the supplier of the electronic control unit). But there is a WCET analysis of the sequential version available, which is taken from the real system using RapiTime.

The software runs in two phases: first, it initializes the system, i.e., the machine is being started. Then the main loop is run as a while (TRUE)-loop until the machine is being turned off. For parallelization the parMERASA parallelization approach from UAU has been applied. In the Analysis phase several possibilities for parallelism have been identified: On the one hand, there are numerous periodic tasks, e.g., for communication between electronic control units like the drivers control screen, keyboards and others. We use several cores only for these periodic tasks. Therefore, no complex scheduler is needed anymore. On the other hand, the main loop can be parallelized using algorithmic skeletons developed in work package 2. It contains functions for controlling different parts of the machine, e.g., for rotating or moving the foundation crane or the subprogram running BDC.

One of the first tasks in the project parMERASA was to analyze the software of the foundation crane BAUER MC128. Results of the program analysis are numerous activity and pattern diagrams. Some of them are presented in this chapter. The diagrams contain no shared resources (e.g., global variables); otherwise, the presentation would be too complex. Two important activity and pattern diagrams are presented below - vBetrieb and vLogo_mc.

Figure 28 shows the detailed presentation of vBetrieb, which can be itemized as Task Parallelism instance. The upper four activities Pumpe1 to vModus cannot be parallelized because of their structure or rather they are too small to be considered in detail or to be parallelized. In favor the four activities vLogo_allg to vBohrverfahren contain many possibilities to exploit parallelism. Next, vLogo_mc is considered more accurate because it contains many parallel executable activities. The detailed form of activity vLogo_mc can be seen in Figure 29. It consists of the activity Init_vLogo_mc and the Task Parallelism instance vLogo_mc_tasks. This in turn can be broken down into two more (very comprehensive) Task Parallelism instances and the “small” activity Hintergrundbeleuchtung. The Task Parallelism instance Funktionen_Oberwagen is limited for the sake of convenience to activities that are executed only by BAUER MC128. If those of the other foundation cranes series would be involved, several case distinctions would take place and the chart would be much more confusing. In Funktionen_alle_MCs activities are included that run on all foundation cranes. A brief overview of all applied Parallel Design Patterns is given in Table 6. The center column indicates how many activities are executed in parallel, which can be mapped to Parallel Design Patterns. The last column indicates which Patterns are applied. There can be seen that only the Parallel Design Patterns Task Parallelism (total 16x) and Periodic Task Parallelism (total 6x) are applicable. Therefore, overall 22 Patterns can be applied, two of them for the initialization and one for the activities which have been executed by the scheduler so far. As a result there are 10 activities for parallel execution in the initialization and 119 activities (eight from the scheduler, 12 periodic tasks contained in the main control loop and 99, which are called in Task Parallelism instances) in the main program.

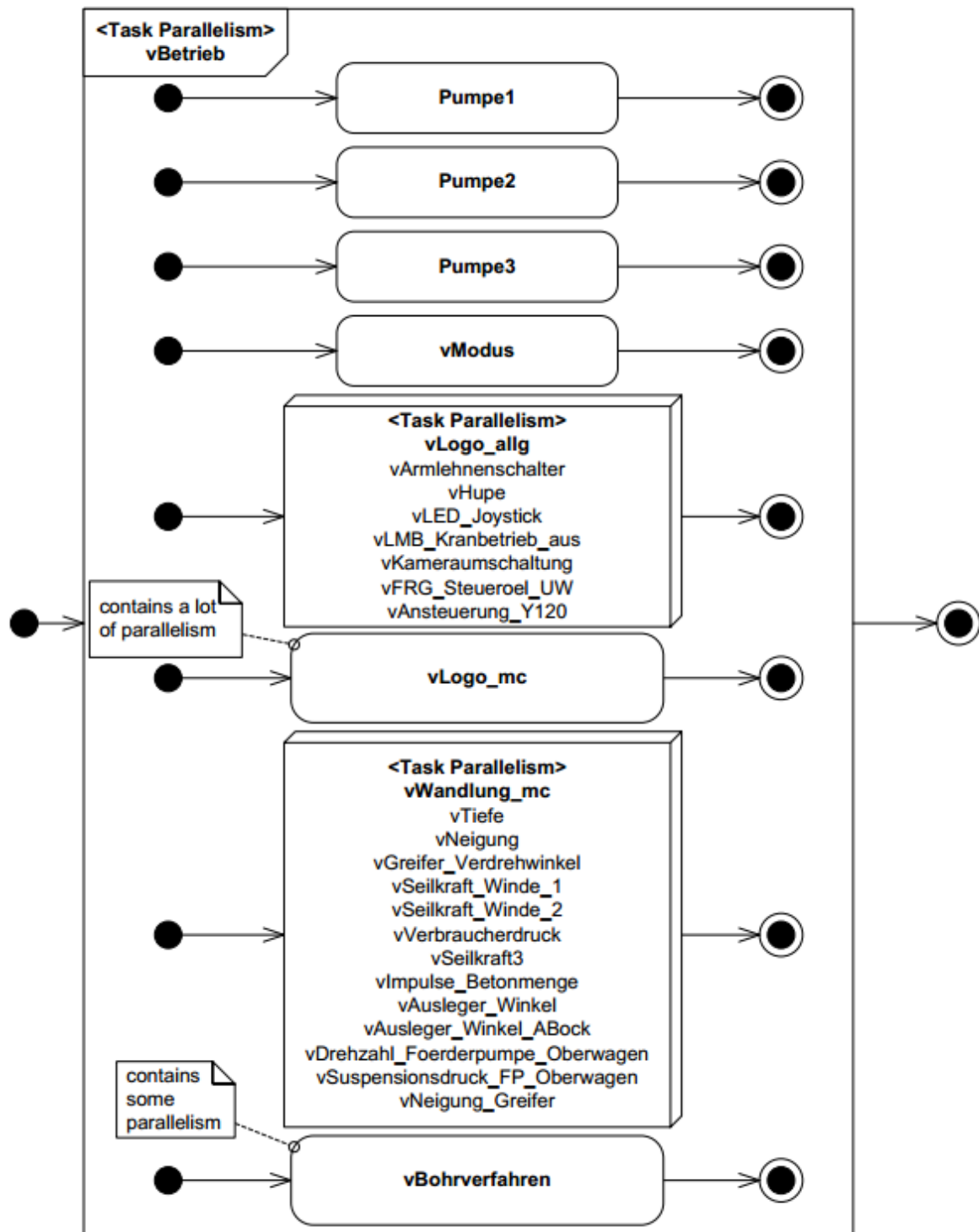


Figure 24: APD of vBetrieb: Eight activities, whereof the half parallelizes as well, execute parallel.

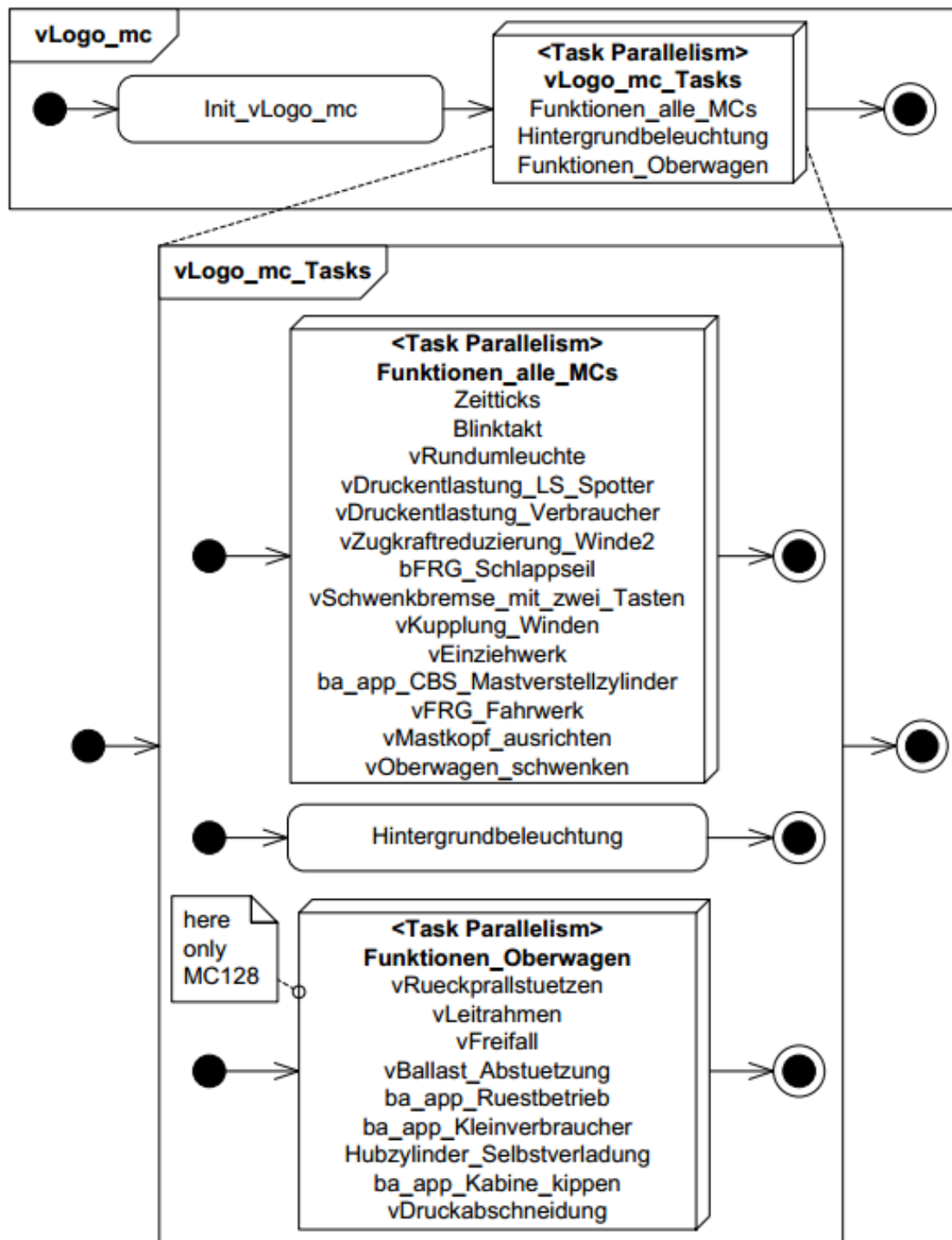


Figure 25: APD of vLOGO mc: There is a short initialization and a small Task Parallelism instance. This includes besides backlight two activities with a total of 25 parallelizable activities.

Table 6: Overview of applied Parallel Design Patterns: Upper two for initialization, middle for the scheduler, lower 19 for the main program.

Name	#Activities	Applied Pattern
Init_PWM	3	Task Parallelism
Init Procedures & Measurements	7	Task Parallelism
Middleware and User Tasks	8	Periodic Task Parallelism
main_loop	9	Task Parallelism
vBetrieb	8	Task Parallelism
vLogo_allg	7	Task Parallelism
vLogo_mc_Tasks	3	Task Parallelism
Funktionen_alle_MCs	13	Task Parallelism
vFRG_Fahrwerk	3	Task Parallelism
vfahrwerk	3	Task Parallelism
Funktionen_Oberwagen	10	Task Parallelism
vFreifall	4	Task Parallelism
vWandlung_mc	13	Task Parallelism
vfunktion_bdc	3	Task Parallelism
vCan2Communication	3	Periodic Task Parallelism
vCan4Communication	2	Periodic Task Parallelism
vTiefenmessung	2	Periodic Task Parallelism
vTiefenmessung2_Tasks	2	Periodic Task Parallelism
Fault Handling	3	Periodic Task Parallelism
vFehler_Meldung_Neu	13	Task Parallelism
vFehler_Task_1000ms	6	Task Parallelism
Everything Else	4	Task Parallelism
Sum Task Parallelism (main program)	99	Task Parallelism
Sum Periodic Task Parallelism (main program)	12	Periodic Task Parallelism

5.2 Optimized application

In the optimized version, the code is now running on twelve cores. Two of them are used only for periodic tasks; the main-loop now runs on 10 cores. The structure of the main control loop can be seen as a sequence diagram in Figure 31. The distribution between the different cores is done with the Timing Analyzable Algorithmic Skeletons (TAS). Previously the assignment has been validated by UAU for its potential speedup by calculation. The version which should produce a good WCET speedup with few cores and few shared variables was implemented. In the chart below, various configurations and their theoretical execution time are shown. On the X-axis, the number of cores and on the Y-axis the execution time in cycles is shown.

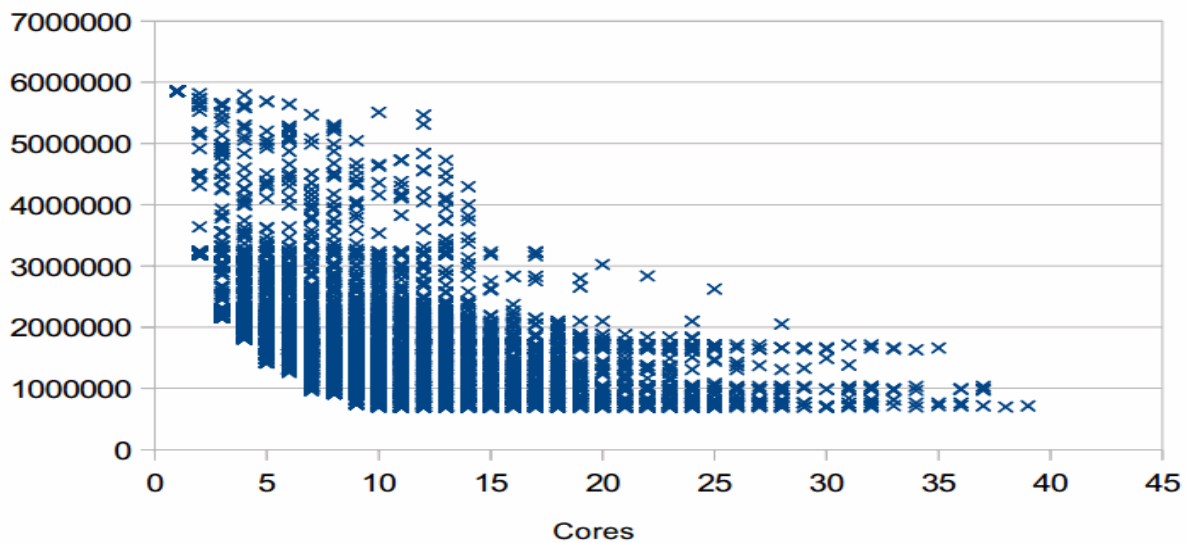


Figure 26: Theoretical execution time for various partitions of the main-loop. On the X-axis are the numbers of cores, which are used by the main-loop. On the Y-axis, the expected execution time in cycles is shown. The overhead for synchronization is not included in the calculation.

At this time, the WCET cannot be determined by tools. Therefore, the WCET speedup has not been determined yet. However, the average speedup could be determined, which is relatively small. This is due to high busload in the parMERASA architecture. The cores, waiting for work from TAS, are waiting at a barrier. This means that they interfere very often in a wait loop on a shared variable to read. This also delays cores that are currently on the execution of a program part. The Timing Analyzable Algorithmic Skeletons are not designed for a high average speedup, but for a good WCET speedup. Measurements and calculations must show now whether we actually achieve a good speedup.

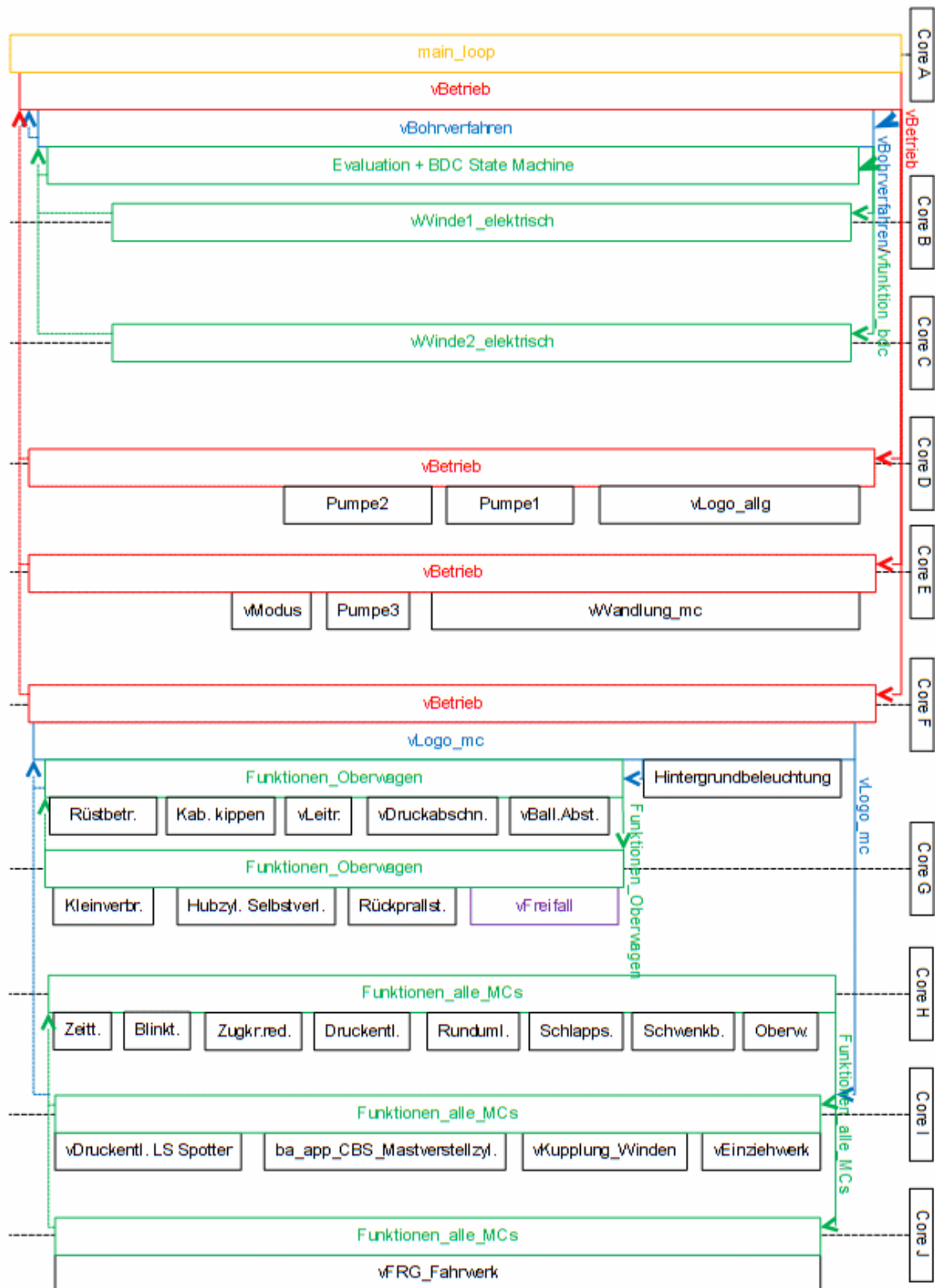


Figure 27: Sequence diagram of the optimized Bauer application

5.3 Conclusions and outlook

As a next step, the WCET needs to be determined. Making optimizations for a good WCET is only possible after the analysis. However, the application resulting from the previous theoretical calculations should bring a good WCET speedup. The Timing Analyzable Algorithmic Skeletons are designed for a good WCET. Currently, only the average time is measured and for hard real-time applications, this is not a good measure for a speedup to determine.

6 CONCLUSIONS

Future embedded systems will feature more often processors with multiple cores. To gain benefit from these additional resources requires multithreaded applications. However, these applications must sustain timing analysability, hence have defined worst-case execution times (WCETs).

This deliverable presents a parallelization approach developed by UAU for embedded hard real-time systems and its application and adaption in case-studies from three different domains. The approach starts sequential code parts in a legacy single-core application. It comprises two phases, which are executed in a model of the software. In the first phase with analysis of the existing software an Activity and Pattern Diagram (APD) models a high degree of parallelism with Parallel Design Patterns (PDPs) out of a Pattern Catalogue as well as data dependencies. In the second phase this high degree of parallelism is optimized respecting the trade-offs of the target platform. After this is completed the existing source code is evolved according to the optimized model.

Timing analysability is sustained because (a) the existing single-core code is supposed to be analysable; (b) parallelism is introduced only with PDPs as structures known to be analysable, and (c) because for the implementation of PDPs only a small number of analysable Synchronisation Idioms (SIs) like ticket locks and fetch-and-increment-barriers are employed.

The BMA application consists mainly of several periodic tasks and a main control loop. The periodic tasks can be executed on dedicated cores with a static cyclic schedule; hence not interrupting the control loop any more. The model-based parallelisation approach was directly applied by BMA for the main control loop in the application. In a preliminary analysis the measured execution times for the control loop being executed on a single core and multiple cores were compared showing a good potential for speedup. In the evaluation phase of the project the WCET will be calculated and compared for (a) a single-core version with a theoretic scheduler, (b) a version with three threads where two threads are executing periodic tasks and one thread executed the main loop, and (c) several optimized versions. Yet it is not clear if the desired eight-fold WCET speedup can actually be reached due to the high number of control dependencies typical for the nature of the application.

The situation is different in the automotive domain, which was in the focus for DNDE, where application structure is much more strictly defined than for the construction machinery domain. The AUTOSAR standard leads to structures consisting of Tasks on a higher level and Runnables on a lower level. With the adapted parallelization approach presented by DNDE it is possible to exploit parallelism on multiple levels in a kind of time-triggered architecture. The introduction of supertasks leads to an additional speedup. Preliminary results running on top of the tiny automotive RTE are presented and in the final evaluation these results will also be compared with WCET numbers gained with RapiTime.

HON provides two applications from the avionics domain and performs studies on a COTS platform. The degree of parallelism in the 3DPP and StereoNav applications were increased with parallel design patterns. Also a switch in the 3DPP from conditional variables towards timing-analysable barriers as implemented in the kernel-library was made. This has strong influences on analysability. Both applications are very data-intensive and, therefore, a high speedup is theoretically possible. Also preliminary results on communication and synchronisation effort are shown leading to the conclusion that predicted, i.e., values which can be calculated by timing analysis tools, are very close

to actual values. The results available for the P4080 will be acquired on the now selected Intel i7 platform soon.

In general the way from single-core to multi-core software was paved and with modifications the application partners were able to follow it. Compared to D2.4, the state of the applications presented is optimized in various means. Future work will mainly be an evaluation of the complete applications with the timing analysis tools by UPS and RTP, as well as for HON, a continuation of the selected COTS platform.

7 REFERENCES

- [1] OMG Unified Modeling Language™ (OMG UML), Superstructure Version 2.4.1, 2011.
- [2] M. Alt, H. Bischof, and S. Gorlatch. Program development for computational grids using skeletons and performance prediction. *Letters*, 12:157–174, 2002.
- [3] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Ottawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer Berlin Heidelberg, 2011.
- [4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.
- [5] A. Bonenfant, I. Broster, C. Ballabriga, G. Bernat, H. Cassé, M. Houston, N. Merriam, M. de Michiel, C. Rochange, and P. Sainrat. Coding guidelines for wcet analysis using measurement-based and static analysis techniques. Technical Report IRIT/RR-2010-8-FR, IRIT-Institut de recherche en informatique de Toulouse, March 2010.
- [6] D. K. G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, Department of Computer Science, University of York, UK, 1996.
- [7] H. K. Cho, T. Kelly, Y. Wang, S. Lafortune, H. Liao, and S. Mahlke. Practical lock/unlock pairing for concurrent programs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12, Los Alamitos, CA, USA, 2013. IEEE Computer Society.
- [8] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, October 1989.
- [9] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, March 2004.
- [10] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [12] G. Gebhard, C. Cullmann, and R. Heckmann. Software Structure and WCET Predictability. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18, pages 1–10, Dagstuhl, Germany, 2011.
- [13] M. Gerdes. *Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores*. PhD thesis, University of Augsburg, 2013.
- [14] M. Gerdes, R. Jahr, and T. Ungerer. parMERASA Pattern Catalogue: Timing Predictable Parallel Design Patterns. Technical Report 2013-11, Fakultät für Angewandte Informatik der Universität Augsburg, 2013.
- [15] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat. Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 671–676, March 2012.
- [16] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, Nov. 2010.
- [17] R. Jahr, M. Frieb, M. Gerdes, and T. Ungerer. Model-based Parallelization and Optimization of an Industrial Control Code. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme X, Schloss Dagstuhl, Germany, 2014, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 63–72, Schloss Dagstuhl, April 2014. fortiss GmbH, München.
- [18] R. Jahr, M. Frieb, M. Gerdes, T. Ungerer, A. Hugl, and H. Regler. Large construction machine control code: Pattern-based parallelization for a many-core. Unpublished: submitted to SIES 2014, under review, 2014.
- [19] R. Jahr, M. Gerdes, and T. Ungerer. On Efficient and Effective Model-based Parallelization of Hard Real-Time Applications. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX, Schloss Dagstuhl, Germany, 2013, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 50–59, Schloss Dagstuhl, April 2013. fortiss GmbH, München.
- [20] R. Jahr, M. Gerdes, and T. Ungerer. A pattern-supported parallelization approach. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 53–62, New York, NY, USA, 2013. ACM.

- [21] R. Jahr, M. Gerdes, T. Ungerer, H. Ozaktasy, C. Rochangey, and P. G. Zaykov. Effects of structured parallelism by parallel design patterns on embedded hard real-time systems. Unpublished: submitted to ECRTS 2014, under review, 2014.
- [22] K. Kennedy, K. S. McKinley, and C. W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):329–341, July 1991.
- [23] M. Kim, H. Kim, and C.-K. Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *MICRO*, pages 535–546. IEEE, 2010.
- [24] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, Dec. 2002.
- [25] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Patterns for Parallel Application Programs, 1999.
- [26] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Reengineering for parallelism: an entry point into PLPP for legacy applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(4):503–529, March 2007.
- [27] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.
- [28] H. Ozaktas, C. Rochange, and P. Sainrat. Automatic WCET Analysis of Real-Time Parallel Applications. In *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30, pages 11–20, Dagstuhl, Germany, 2013.
- [29] M. Poldner and H. Kuchen. On implementing the farm skeleton. *Parallel Processing Letters*, 18(1):117–131, 2008.
- [30] C. Rochange. An Overview of Approaches Towards the Timing Analysability of Parallel Architecture. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18, pages 32–41, Dagstuhl, Germany, 2011.
- [31] Y. Sato, Y. Inoguchi, and T. Nakamura. Whole program data dependence profiling to unveil parallel regions in the dynamic execution. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 69–80, 2012.
- [32] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Syst.*, 28(2-3):157–177, Nov. 2004.
- [33] Radojkovic, P., & Girbal, S., & Grasset, A., & Quiñones, E., & Yehia, S., & Carzola, F. C. On the Evaluation of Impact of Shared Resources in Multithreaded COTS Processors in Time-Critical Environments. *ACM Transactions on Architecture and Code Optimization*, 8(4), (2012)
- [34] Pyka, A., Rohde M., Zaykov P. G., Uhrig S. Case Study: On-Demand Coherent Cache for Avionic Applications, Workshop on High-performance and real-time embedded systems, (2014)
- [35] Radojkovic, P., & Girbal, S., & Grasset, A., & Quiñones, E., & Yehia, S., & Carzola, F. C. On the Evaluation of Impact of Shared Resources in Multithreaded COTS Processors in Time-Critical Environments. *ACM Transactions on Architecture and Code Optimization*, 8(4), (2012)
- [36] AUTOSAR consortium. AUTomotive Open System ARchitecture (AUTOSAR). <http://www.autosar.org>, January 2012.
- [37] David I. August, J. Huang, T. B. Jablin, Hanjun Kim, T. R. Mason, P. Prabhu, A. Raman, and Yun Zhang. Automatic extraction of parallelism from sequential code. In V. Pankratiyus, A. Adl-Tabatabai, and W. Tichy, editors, *Fundamentals of Multicore Software Development*, chapter 9, pages 201 – 238. Chapman & Hall / CRC Press, 2011.
- [38] Oliver Sinnen. Graph Representations. In *Task Scheduling for Parallel Systems*, chapter 3, pages 40 – 73. Wiley, Newark, 2007.
- [39] Christoph Kirsch and Ana Sokolova. The Logical Execution Time Paradigm. *Advances in Real-Time Systems*, page 103, 2012.