

parMERASA

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

D 2.6 – Final Evaluation Results on Parallel Industrial Applications

Nature:

Dissemination Level:

Due date of deliverable:

Actual submission:

Responsible beneficiary:

Responsible person:

R - Report

PU - Public

2014-09-30

2014-09-30

HON

Theo Ungerer

Grant Agreement number:

Project acronym:

Project title:

Project website address:

Funding Scheme:

Date of latest version of Annex I
against which the assessment will be made:

Project start:

Duration:

Period covered:

FP7-287519

parMERASA

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

<http://www.parmerasa.eu>

STREP

SEVENTH FRAMEWORK PROGRAMME
THEME ICT – 2011.3.4 Computing Systems

June 20, 2012

October 1, 2011

36 month

2013-10-01 to 2014-09-30

Project coordinator name, title and organisation:

Tel: + 49-821-598-2350 Fax: + 49-821-598-2359

Prof. Dr. Theo Ungerer, University of Augsburg

Email: ungerer@informatik.uni-augsburg.de

Release Approval

name	role	Date
Pavel Zaykov	WP2 Leader	2014-09-30
Hans Regler	WP2 Partner	2014-09-30
Bert Bøddeker	WP2 Partner	2014-09-30
Theo Ungerer	Coordinator	2014-09-30

DELIVERABLE SUMMARY

In this deliverable, we incorporate the final results from the parMERASA project and we derive the overall conclusions based on the conducted experiments with the industrial applications. For each industrial application prototype, we measure the Worst Case Execution Time (WCET) from the parMERASA platform and we estimate the WCET with the parMERASA WCET tools - RapiTime and OTAWA. Once we obtain the WCET, we compute the WCET speedup and efficiency and derive conclusions for the effectiveness of particular parallelization.

For completeness of the WCET-aware parallelization discussion, we suggest to refer to already conducted experiments, reported in parMERASA deliverable D2.4. Furthermore, we provided a prototype deliverable accompanying the current deliverable D2.6, which describes in detail how to access, configure and run the target application configurations on the parMERASA simulator platform. Please note that the supporting prototype deliverable was not explicitly required by the project Description of Work (DoW).

In the current deliverable, we cover the following tasks from the parMERASA DoW:

Task 2.6: Evaluation of Different Program and Multi-core Configurations

- Evaluation of Different Program and Multi-core Configurations
- Parallel applications will be scaled targeting different levels of parallelism.
- Parallel applications will run on several configurations of the parMERASA multi-core with differing number of cores, interconnect variations, memory and I/O resources.
- Communication and synchronisation overhead, average and WCET speedups will be finally assessed.
- Deriving an efficiency table for different core numbers, network and memory configurations of the parallel applications.
- HON will investigate possible optimisations and design recommendation for getting less pessimistic WCET and achieving WCET speedup on COTS target. Furthermore HON will perform final tests on COTS and parMERASA multi-core platforms, will report on findings with respect to WCET speedup, WCET scalability, certification cost. At the end a technology readiness level assessment should be provided.
- Target after month 36 is a clear assessment of:
 - 1) The reachable parallelism degree of the parallelised applications that allow companies to assess the efficiency of their applications for implementation on different multi-cores from high core numbers of 64 down to low core numbers of 8, 4, or 2;
 - 2) Impact of parallelization on WCET and WCET speedup;
 - 3) Impact of inter-communication and synchronisation patterns on WCET estimates.

Conclusions of task 2.6:

We perform WCET evaluation and analysis over a broad set of application configurations and multi-core configurations for each one of the industrial prototypes. In all industrial prototypes, an application “sweet spot” was identified while experimenting with various levels of parallelism. The application “sweet spots” were identified with respect to the optimal WCET speedup and efficiency. The experiment results suggest that the industrial application achieve up to 6 times WCET speedup with an optimal number of processor cores varying from 8 cores to 12 cores depending on the application configuration. A high number of processor cores (such as 16 or more) results in high deviations in execution time and hence high WCET overestimation.

A technology readiness assessment has been performed and the outcome is reported in parMERASA deliverable D6.10.

As a result of the above listed accomplishments, we consider that the objectives of Task 2.6 have been successfully reached.

TABLE OF CONTENTS

1	Introduction.....	7
2	Parallelization Approach with Parallel Design Patterns for Hard Real-Time Systems	8
2.1	Motivation	8
2.2	Structured Parallelism by Parallel Design Patterns	8
2.3	Parallelization Approach for Hard Real-time Embedded Systems	10
2.4	Tool Support for the Parallelization Process	10
2.5	Quantification for the Effectiveness of the Parallelization	11
3	Industrial Applications – analysis and evaluation	13
3.1	WP2 integration to parMERASA tools and platform	13
3.2	Parallelization, integration, and porting of the industrial applications	13
3.3	Set of evaluation criteria	13
4	Avionics Applications.....	15
4.1	Application and platform configurations	15
4.1.1	Avionics application configurations.....	15
4.1.2	Avionics platform configuration	16
4.2	WCET experimental results with the avionics applications.....	18
4.2.1	3D Path Planning application	18
4.2.2	Stereo Navigation application	25
5	Automotive applications	34
5.1	Automotive application	35
5.2	Intra-task parallelism.....	35
5.2.1	Mapping Runnables.....	35
5.2.2	Supertask	36
5.2.3	Quantitative Analysis.....	36
5.2.4	Summary.....	39
5.3	Inter-Task parallelism	39
5.3.1	Determining the data flow	39
5.3.2	Quantitative analysis	41
5.3.3	Prototype Evaluation	45
5.3.4	Summary.....	48
5.4	Intra-Runnable Parallelism	48
5.4.1	Parallelization	49
5.4.2	Implementation.....	53
5.4.3	Evaluation	57

5.4.4	Conclusion	61
5.5	Combined Approach.....	62
5.5.1	Experimental results.....	62
5.5.2	Summary.....	63
6	Construction Machinery.....	64
6.1	Single core application on the parMERASA platform.....	64
6.2	Parallel applications using timing analysable algorithmic skeletons	65
6.3	Simulation of parallel Code	68
6.4	WCET analysis.....	69
6.5	Conclusions and outlook	71
7	Conclusions.....	72
7.1	Summary on the WCET-aware parallelization on a multi-core system.....	72
7.2	WP2 progress with respect of the parMERASA SMART Objectives	74
8	References.....	76

1 INTRODUCTION

In this deliverable, we introduce the final number of application configurations for the three industrial prototypes. We provide further application improvements, which favour the worst-case execution time (WCET) analysis by the parMERASA WCET tools – RapiTime and OTAWA, which has matured. For each one of the industrial prototype configurations, we measure the WCET, WCET speedup or idle time (slack), and WCET efficiency. Furthermore, we compare these WCET numbers among three common WCET methodologies – measurement-based and estimated by measurement-based analysis (RapiTime) and static-based analysis (OTAWA).

The wide range of application configurations spawns from sequential execution (1 core) to a high level of parallelism with high number of cores in the multi-core system (up to 16 cores). Throughout, all industrial prototypes we have observed WCET speedup and the optimal parallelization point (referred as sweet spot) has been identified.

The experimental results suggest that application configuration from 4, 8 and up to 12 cores (depending on the application) using barrier synchronization proves to be the most efficient ones in terms of WCET speedup. A higher number of cores results than 12, results in high execution time deviation, which contributes to more conservative WCET overestimation. Throughout the application sweet spots, the application WCET speedup varies up to 6 times compared to sequential execution. Further WCET speedup might be achieved in case the applications are completely redesigned from scratch with parallelised algorithms in mind.

During the parMERASA project execution, both WCET tools – RapiTime and OTAWA has matured enough to be able to analyse parallelised industrial applications on a multi-core platform. Even more, the experimental results confirm that static WCET analysis is capable to achieve very tight WCET with respect to the measurement-based WCET, when an accurate model of the architecture is available and initial memory hierarchy preconditions are set correctly. Contrary, a high WCET overestimation is sobered in case the architecture model and/or memory is misconfigured.

The rest of the deliverable is organized as follows. In Section 2, we summarize the workflow and benefits of using structured parallelism with the help of design patterns. In Section 3, we outline the common set of experimental and application setup employed in each one of the industrial domains. In Section 4, Section 5, and Section 6, we summarize our findings for each one of the domains. In Section 7, we outline the conclusions and summary on the coverage on the project SMART objectives as defined by DoW.

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

2.1 Motivation

First multi-core processors are available for embedded systems. They could also be attractive for systems with hard real-time requirements: (a) By parallelization, existing programs can be accelerated. (b) More complex algorithms or additional functionality can be implemented with the newly available computation power. (c) The truly parallel execution of program parts can reduce jitter in update intervals. (d) Because the same computation power can be provided at lower frequency compared to a single-core processor, the clock frequency can potentially be reduced leading to energy savings.

Some work effort is necessary to port existing single-core applications to multi-core processors [10]: If software is already designed in small pieces, they can be scheduled onto multiple cores with suitable tools. In the parMERASA project, this is the case for the automotive application by DNDE (see Section 5) and the periodic tasks of the control application of the crawler crane by BMA (see Section 6). The situation gets more complicated if larger pieces of sequential code shall be parallelized. Examples are in parMERASA the data-intensive next-generation avionic algorithms by HON (see Section 4) and the main control loop of the BMA control code (see Section 6). For this, a transition path starting from a sequential single-core program to a timing analysable parallel program with structured parallelism was developed and is presented shortly in the remainder of this chapter.

2.2 Structured Parallelism by Parallel Design Patterns

In the “high performance” and embedded domain without the need for timing analysis, there are already many different approaches for parallel programming with unstructured parallelism (e.g., POSIX Threads) and more structured parallelism¹ (e.g., OpenMP) for shared memory systems. In addition, automatic parallelization can be applied mostly leading to unstructured parallelism.

Traditional approaches for structured parallelism often involve libraries and schedulers for dynamic dispatching of work; such strategies are not analysable with reasonable overhead because of their dynamic nature. Hence, parallel programming means similar to POSIX threads are in the focus in the remainder. The main reason for this is the small abstraction from the hardware. The targeted hardware platform is the predictable and scalable many-core parMERASA platform (see other deliverables). It features a NoC and virtual shared memory and was designed with focus on reducing interferences between threads.

The main challenge for the analysis of parallel software is still to reconstruct the interplay of the threads of the parallel software. Hence, for the static timing analysis with the OTAWA tool-set by UPS it is necessary to describe design-time knowledge as “parallel flow facts” by an additional XML file revealing the interaction of different pieces of code identified by ID-tags in the source code (see Deliverable 3.2). This preparation effort can be reduced a lot by structured parallelism.

¹ “The structured approach to parallelism proposes that commonly used patterns of computation and interaction should be abstracted as parameterisable library [...]”, Murray Cole in his presentation “Why structured parallel programming matters”, 2004-09-03 in Edinburgh

In addition, the potential over-estimation, i.e., the difference between the calculated worst-case execution time and the true WCET, can be high if synchronization primitives are applied which are hard to analyse. Hence, it is necessary to limit the set of allowed synchronization primitives to those, which are analysable. On the parMERASA platform, the fetch-and-increment barrier and ticket locks are available for process coordination and progress coordination (see [3, 9]).

Besides this, it is necessary (cf. [9]) that software (a) has a fixed and static mapping of program parts to threads and cores, (b) limited usage of shared dynamic data structures, and (c) no indeterminism by race-conditions. In addition, well-known programming guidelines for single-core HRT systems have to be applied.

One way leading to structured parallelism is with parallel design patterns (PDPs). They describe in an abstract textual way best-practice solutions for parallel situations. In the scope of parMERASA, the industrial applications by BMA, HON, and DNDE were investigated and four PDPs were isolated (see Table 1) fulfilling the requirements for analysability as mentioned above. These PDPs are described in a schema extended with hard real-time aspects in [3]. Figure 1 shows an overview of the concepts PDP and synchronization idioms, which represent platform specific code fragments for synchronization primitives. The enforcement of structured parallel programming leading to an implementation with only known and analysable structure and SIs has positive effects on both the WCET preparations and the WCET analysis results; this is described in detail in article [9].

Table 1: Analyzable PDPs found in the four in the four industrial applications

Application	Task Parallelism	Periodic Task Parallelism	Data Parallel	Pipeline
HON 3DPP			X	X
HON Stereo-Nav			X	X
BMA Crawler Crane	X	X		
DNDE Engine Control	X	X		

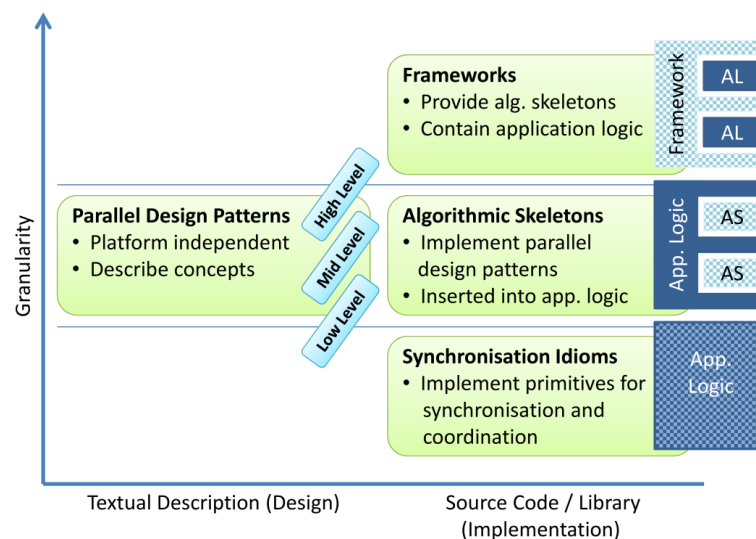


Figure 1 Overview of the concepts for parallelization with Parallel Design Patterns (PDPs). The illustrations on the right show the interplay with application logic (custom code) and the source-code parallelization concepts; checked contain synchronization elements

2.3 Parallelization Approach for Hard Real-time Embedded Systems

The *pattern supported parallelization approach* provides a way to enrich existing sequential pieces of single-core code with PDPs. It is a model-based approach with three steps and four artefacts as illustrated in Figure 2. The starting point is the existing sequential software.

The model of the software with a high degree of parallelism is constructed by in-depth analysis and integration of PDPs from a defined Pattern Catalogue. Placing the PDPs, i.e., the actual parallelization process requires knowledge of the PDPs and the structure of the application to parallelize. The resulting model is typically done as *Activity and Pattern Diagram (APD)*, which is an extension of the UML2 Activity diagram (see Figure 3). The resulting model does not consider trade-offs imposed by the target platform. By optimization, a refined model with lower degree of parallelism tailored for the target platform is built. As last step, the existing software is modified according to the optimized model leading to the parallel implementation. A WCET analysis is finally possible with the parallel software. Details of this parallelization process are described in articles [8, 7, 5] and Deliverable 2.4.

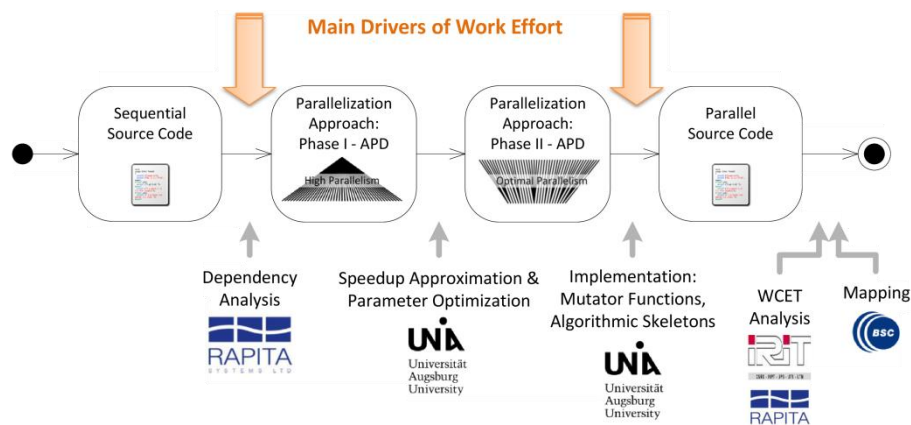


Figure 2 Overview of the pattern-supported parallelization process and suggested tools to ease the process (analysis, optimization, and implementation)

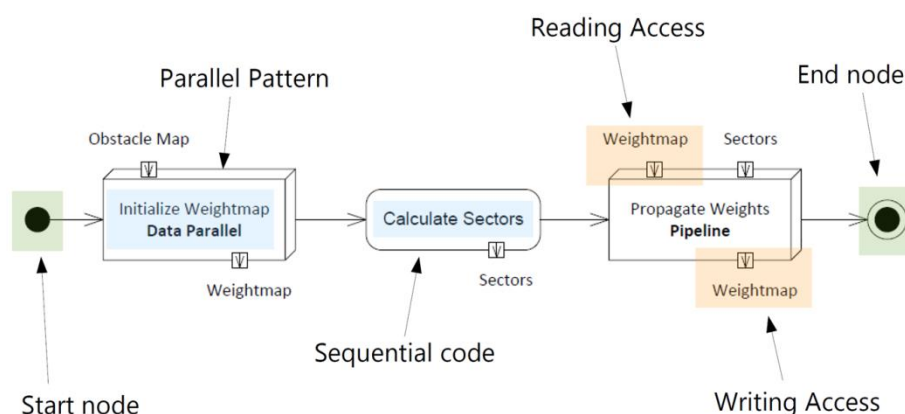


Figure 3 Example of an Activity and Pattern Diagram (APD) with PDP at the left and right and a sequential code block in the center; of importance is the modeling of data input and output by pins implicitly representing data dependencies.

2.4 Tool Support for the Parallelization Process

Tool support is extremely important to reduce manual work effort in the parallelization process. As shown in Figure 2, the main work effort is caused by dependency analysis and

implementing the parallel code. In the parMERASA project, tool support was developed as follows:

- The Parallelization Assistant by RPT can help in the code analysis of the software;
- The model-based optimization can be done with a custom tool by UAU, see [5];
- The implementation of the PDPs can be eased with the Timing analyzable Algorithmic Skeletons (TAS), which are platform-specific implementations for PDPs (cf. Figure 1);

Since Deliverable 2.4, the most progress was made with the skeleton library TAS. The implementation was completed for the PDPs *Task Parallelism*, *Data Parallelism*, and *Parallel Pipeline* according to guidelines for timing analysable code [1, 2]. For example, the assignment of tasks is done statically as well as the mapping of tasks to threads and cores with running one thread on one core; there are no dynamic memory allocation, no work stealing or other kinds of dynamism.

Also ways to strongly reduce the work effort for an analysis with OTAWA (see Figure 4) were developed. With TAS, most parallel control commands are hidden in the TAS library; they can hence be annotated with IDs independent from their use. Only the execute call to a skeleton must have its own ID; this is also the case for the initialization and finalization of a skeleton if these function calls are in the scope of the WCET analysis. The skeleton with these IDs can then be described in a comparably simple XML file and with a code generation tool the more complex OTAWA XML can then be built.

The TAS are applied for the parallelization of the BMA control code. In addition, it is used as reference for the intra-Runnable parallelization of the DNDE engine control code. The code and the tools are released as open source and details to the use are described in technical report [4].

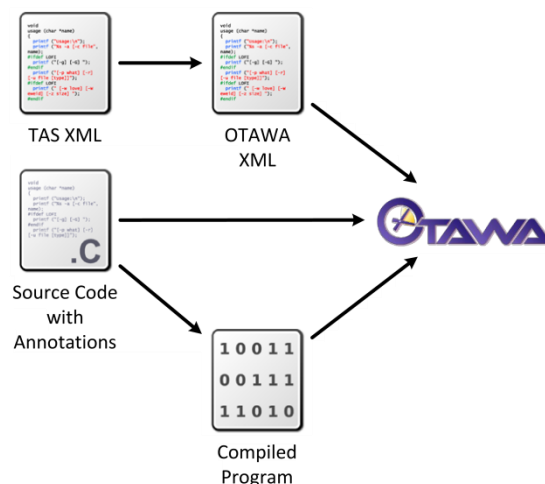


Figure 4 Information flow for WCET analysis with OTAWA of code with Timing Analysable Algorithmic Skeletons (TAS)

Further possible and partly available tool-support is described in article [6] and parMERASA Deliverable 2.4.

2.5 Quantification for the Effectiveness of the Parallelization

Not only the well-known speedup should be considered to measure the effectiveness but, especially if the potential to execute “more” software, also slack time [6] describing the time available until the

deadline. Hence, it is a good measure for the potential to run algorithms that are more complex or to provide further functionalities. This is especially of interest for the applications of BMA and DNDE, where “as fast as possible” cannot be the goal of the parallelization: The system works well as long as defined deadlines are not missed, hence also with a single core.

Figure 5 shows a generalized example derived from the BMA control application. A set of periodic tasks is executed with a main control loop; the deadline and the considered time interval is 7 ms.

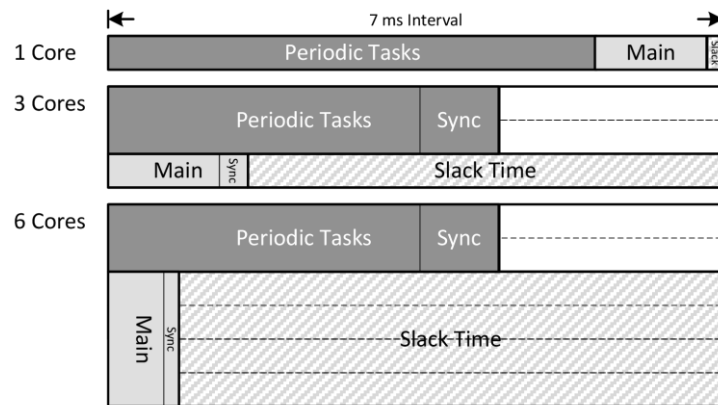


Figure 5 Graphical representation for utilization and slack time in a 7 ms interval for one, three, and six cores running the construction machine control code

In the single-core implementation of the application, about 0.8 ms of a 1 ms interval are needed for periodic tasks and 0.2 ms remain for the main control loop. It takes about 6 ms to complete an iteration of the main loop, which is below the deadline of 7 ms. The slack time is hence 0.2 ms per 7 ms.

With more cores, both the periodic tasks and the main control loop are slowed down by the synchronization and parallelization overheads (represented by *Sync* in Figure 5; a constant factor of 0.25 is assumed). With three cores in total, the slack time is $7 \text{ ms} - 6 * 0.2 \text{ ms} * 1.25 = 7 \text{ ms} - 1.5 \text{ ms} = 5.5 \text{ ms}$ per 7 ms interval.

For example, with six cores, the main control loop can be parallelized and executed on multiple cores with a speedup; let it be 2 as conservative assumption for 4 cores. This leads to a higher slack time either (a) for a sequential code part ($7 \text{ ms} - 1.5 \text{ ms} / 2 = 6.25 \text{ ms}$ per 7 ms interval), (b) for a parallel code part in the same style as the existing control loop with a speedup of 2 ($6.25 \text{ ms} * 2 = 12.5 \text{ ms}$ per 7 ms interval), or (c) a highly parallel code part efficiently using all four cores available for the main loop ($6.25 \text{ ms} * 4 = 25 \text{ ms}$ per 7 ms interval).

This shows that rating of the slack time available on more than one core must always be done in the given context, i.e., what the slack will be used for. Moreover, speedup of sequential code parts increases slack time, i.e., idle time on all cores. Hence, speedup is still a valid optimization goal because it increases the potential to execute more software of the same style.

3 INDUSTRIAL APPLICATIONS – ANALYSIS AND EVALUATION

In this section, we introduce our strategy for the evaluation of the industrial applications. In Section 3.1, we provide an overview of the development and integration status of the original and the optimized industrial applications. In Section 3.2, we reveal more details on the parallelization, integration, and porting efforts. In Section 3.3, we define the set of evaluation criteria, which we deploy for the evaluation of the industrial applications, parMERASA analysis tools, and parMERASA architecture optimizations.

3.1 WP2 integration to parMERASA tools and platform

In the table to follow, we summarize the current status of the WP2 integration with respect to the parMERASA tools and platform. The presented Table 2 is an updated version of Figure 1 in parMERASA D2.4.

Table 2 Integration status of industrial applications to the parMERASA tools and platform

Apps	PDPs	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	✓	✓

As the presented data suggests, we have successfully accomplished the project objectives and WCET evaluation of the industrial prototypes.

3.2 Parallelization, integration, and porting of the industrial applications

For each one of the industrial domains during Phase 1 and Phase 2 in the parMERASA project, we performed parallelization and optimizations of the corresponding industrial applications. We ported those applications to the target many- (multi-) core computing architectures and modified them accordingly to accommodate the parMERASA analysis tools (OTAWA and RapiTime). Last but not least, we evaluate the industrial applications using a set of evaluation criteria.

Below, we summarize the activities associated with the parallelization, optimization, porting, and evaluation. We provide a list of parallelization and optimizations performed over the industrial application use-cases. We define as an application use-case to be an instance of an industrial application with a set of Parallel Design Patterns (PDPs) and Synchronization Idioms (SI). For each application use-case, we define the number of threads, deployed Parallel Design Patterns (PDPs) [1], the employed synchronization idioms and also other code optimization towards lower WCET. We present the porting status of the applications, i.e., the integration status to the target architectures. For each one of the architecture, we list the architecture features that are exploited. As specified in parMERASA DoW, HON examines also COTS multi-core architecture. We define the experimental setup in terms of platform and application configurations. Platform configuration might be parameters such as: the size of memory, on-chip interconnects topology, etc. For the application configuration, we left it blank because it highly defers per application.

3.3 Set of evaluation criteria

In this section, we present a list of evaluation criteria that we employed for the evaluation of the application and the architecture optimizations.

Set of evaluation criteria	
BCET	Best Case Execution Time
ACET	Average Case Execution Time
WCET	Worst Case Execution Time
WCET Speedup	single-threaded WCET vs multi-threaded WCET execution (HON) or idle time - slack (DNSO & BAU)
WCET Efficiency	Speedup / number of cores
Execution time variability	WCET-BCET

For the industrial applications, we measure the Best Case Execution Time (BCET) as the min value in the observed execution times or the estimated by the analysis tools. We define Average Case Execution Time (ACET) as an average and Worst Case Execution Time (WCET) as the longest execution of the application.

Once, the execution times are measured, we are able to compute the *WCET Speedup* as the ratio between the single-threaded WCET to the multi-threaded WCET execution. Since we consider a multi-core system, the multi-threaded application always runs on a multi-core processor. We also define the *WCET Efficiency* as the ratio of between the WCET speedup and the number of used cores. Therefore, a single-thread implementation will always have WCET efficiency equal to 1. For applications which react on external events such as interrupts, it is difficult to define WCET speedup and efficiency, so we use as an evaluation criteria of the multi-core execution the idle time, i.e., slack.

For some of the application configurations, we also measure the execution time *variability* because a highly variable execution will introduce high conservatism in the WCET estimation as well as the slack will be increased.

Evaluation methodology for BCET and WCET	
M Measurement-based	Maximum from of all measurements
R Rapitime-based	Hybrid analysis on traces
O OTAWA-based	Static analysis

In parMERASA project, we measure/estimate BCET and WCET employing three methodologies. The first is Measurement-based (noted as M) employing a max out of all observations on the execution times. The second is hybrid – based on measurements (application traces) and the application graph. We refer to it as RapiTime-based (noted as R), since we employ RapiTime toolset to estimate it. The third is static analysis on the application graph and a model of the architecture. We refer to it OTAWA-based (noted as O), since we employ OTAWA for its estimation.

In the sections to follow, we discuss the WCET findings for each of the industrial domains.

4 AVIONICS APPLICATIONS

In this section, we introduce the examined avionics application configurations and present a study conducted on application Worst-Case Execution Time (WCET), WCET speedup, and WCET efficiency.

More precisely, we measure the application WCET by using:

1. The WCET observed (measurement-based) by parMERASA platform built-in tracer, with zero cycles overhead (non-intrusive);
2. The WCET observed (measurement-based) by RapiTime toolchain, including instrumentation overhead;
3. The WCET estimated by OTAWA toolchain with static analysis;
4. The WCET estimated by RapiTime toolchain applying hybrid approach, which is a combination of measurement-based + code analysis;
5. The WCET calculated as a sum of the maximum execution time of the function calls on the critical application path using the Parallel Design Patterns by UAU;

4.1 Application and platform configurations

In this section, we shortly introduce the application and platform configurations.

4.1.1 Avionics application configurations

As it was introduced in the previous parMERASA deliverables, the avionics domain is presented by two industrial applications – 3D Path Planning (3DPP) and Stereo Navigation (SN), respectively. With respect to parMERASA deliverable D2.4, we optimize the avionics applications to ensure a fair comparison among WCET obtained with the measurement tools, i.e., measurement-based parMERASA platform built-in tracer, Static analysis with OTAWA tool, and Measurement-based analysis with RapiTime toolchain.

In the tables to follow, we introduce application configurations with the help of Parallel Design Patterns (PDPs). In [Table 3](#), we present the final parallelization and optimization deployed for the avionics applications.

Table 3 Parallelization and optimization deployed for 3DPP and SN avionics applications

App. use- cases	N thr	Deployed PDPs					Sync idioms	Other opt.
		Task	Periodic task	Periodic & event- triggered task	Data (SPMD)	Pipeline (Cons- Prod)		
3DPP-NoOS-1T	1	---	---	---	---	---	cond. variables, barriers	Improved mapping, core utilization
3DPP-NoOS-2T	2	---	---	---	2	1		
3DPP-NoOS-4T	4	4	---	---	4	1		
3DPP-NoOS-8T	8	8	---	---	8	1		
3DPP-NoOS-16T	16	16	---	---	16	1		
SN-NoOS-1T	1	---	---	---	---	1	barriers	
SN-NoOS-7T	7	7	---	---	---	1		
SN-NoOS-12T_CAM	12	12	---	---	5	1		
SN-NoOS-12T_CONV	12	12	---	---	5	2		
SN-NoOS-14T	14	14	---	---	14	2		
SN-NoOS-16T	16	16	---	---	16	2		

For the 3DPP application, we define five application configurations – varying from 1 thread (sequential) up to 16 threads, referred as: 1T, 2T, 4T, 8T, and 16T. The 3DPP application configurations employ data and pipeline parallelism. Furthermore, we explore two types of synchronization among application threads – with *barriers* and *conditional variables* located in the shared-memory. The input data for 3DPP application is a 3D Obstacle map, defined as a 16x16x8 computational grid.

For the SN application, we define six application configurations – varying from 1 thread (sequential) up to 16 threads. As listed in Table 4, SN application configurations use respectively 1T, 7T, 12T, 14T and 16T. For the SN application configuration with 12 threads, we define two implementations with respect to the employed parallelism, referred as SN-NoOS-12T_CONV and SN-NoOS-12T_CAM. A more detailed description of the SN application configurations is presented in the experimental results section (as part of this deliverable). All SN application configurations receive, as input, a frame composed of two images (namely, left and right image). We experiment with images sized to 160x160 pixels.

Table 4 Variables settings for both 3DPP and SN applications configuration

App. use-cases	Configuration file	Variable name	Variable value
3DPP-NoOS	init.h caCmConfig.h	#define PROCESSORS	1, 2, 4, 8, 16
		#define X_COMPARTMENT_NUM	1, 2, 4
		#define Y_COMPARTMENT_NUM	1, 2
		#define Z_COMPARTMENT_NUM	1, 2
SN-NoOS	init.h snPilotStudyCfg.h	#define PROCESSORS	SN_ALL_THD_NUM
		#define SN_CONFIGURATION	SN_CFG_SEQ_1
			SN_CFG_PIP_7
			SN_CFG_PIP_12_CAM
			SN_CFG_PIP_12_CONV
			SN_CFG_PIP_14
			SN_CFG_PIP_16_CAM

In Table 4, we summarize the parameters used to define the target avionics application configuration. For 3DPP application, the number of threads (i.e., the variable “PROCESSORS”) shall be set according to the total number of compartments given by $X_COMPARTMENT_NUM * Y_COMPARTMENT_NUM * Z_COMPARTMENT_NUM$ expression. For example, in case of sequential 3DPP application configuration, we should set $X_COMPARTMENT_NUM=1$, $Y_COMPARTMENT_NUM=1$, $Z_COMPARTMENT_NUM=1$, and $PROCESSORS=1$. For 16 threads 3DPP configuration, we should set $X_COMPARTMENT_NUM=4$, $Y_COMPARTMENT_NUM=2$, $Z_COMPARTMENT_NUM=2$, and $PROCESSORS = 16$.

For SN application, the desired application configuration shall be specified by setting $SN_CONFIGURATION$ only. For example, the sequential SN application configuration is defined by $SN_CONFIGURATION = SN_CFG_SEQ_1$, while for 14 threads SN configuration we should set $SN_CONFIGURATION = SN_CFG_PIP_14$.

4.1.2 Avionics platform configuration

In Table 5, we list the integrations of avionics applications with respect to the architecture support. As presented, we analyse shared memory and ODC2 cache on the parMERASA many-core platform and few popular COTS multi-core architectures.

Table 5 Porting status of the industrial applications

App. use-cases	parMERASA many-core platform			COTS multi-core architecture		
	Shared mem	Scratchpad	ODC2 cache	P4080	AMD	Intel
3DPP-NoOS	✓		✓	✓	✓	✓
SN-NoOS	✓		✓			

In Table 6, we summarize the parMERASA many-core platform configurations employed for the aforementioned avionics application configurations. We assume that all application threads are executed in a single cluster. We choose not to split the avionics applications among multiple clusters, because they are data intensive and such splitting might result in creating a bottleneck in the application throughput. As reported in previous parMERASA deliverable, multiple avionics applications might be executed in parallel while mapped in different clusters. As a result, it is possible to estimate the inter-cluster communication overheads and conclude the scalability of the architecture when we switch from one application to multiple applications running in parallel. In the current deliverable we experiment with single application only.

Table 6 Experimental setup - application and platform configuration for analysis

App. use-case	Platform configuration				Application configuration		
	Cluster num	Memory	NoC topology	ODC2		Code modifications	
3DPP- NoOS	1	10 cycles delay (on-chip mem)	Tree	icache=4KB, repl, dcache=4KB, lru repl, 4-way assoc, icache hitlatency=1, icache_misspenalty=1, dcache hitlatency=1, dcache misspenalty=1.	perf.	Debugging is disabled	Initialization, finalization skipped from the WCET analysis
SN- NoOS	1	23 cycles delay (off-chip mem)	Tree	icache=4KB, repl, dcache=4KB, lru repl, 4-way assoc, icache hitlatency=1, icache_misspenalty=1, dcache hitlatency=1, dcache misspenalty=1.	perf.	Debugging is disabled	Initialization, finalization skipped from the WCET analysis

A cluster is configured to share a single on-board SRAM memory with a delay equal to 10 clock cycles for 3DPP application, and a single off-chip memory with a delay of 23 clock cycles in the case of SN application. Each thread is mapped onto a single core, where cores and the shared-memory within a cluster are connected with a “tree”-type of network-on-chip (NoC). Depending on the node position in the tree-type interconnect, we assume that the worst-case delay is computed as follows: $zll1$ (no NoC)=2, $zll2=4$, $zll4=5$, $zll8=6$, $zll16=7$, and the worst-case traversal time is computed by $zll[\#cores] + (\#cores) * mem_delay$. For the implementations with the ODC2, each core has data cache (dcache) and instruction cache (icache). Note that both dcache and icache are 4-way associative, size of 4KB, hit delay and miss penalty equal to 1 cycle. The cache hierarchy is modelled by setting perfect replacement policy for low-level instruction cache and least-recently used (lru) policy in the whole cache hierarchy for both dcache and icache. The application use-cases support the two ODC2 configurations: address-checking and write-through, respectively.

Throughout our experiments, two measurement-based techniques have been applied: 1) non-intrusive and zero-delay measurement by enabling parMERASA many-core platform built-in tracing

capability, and 2) employing Rapita toolset (RVS). In both cases, we trace a list of functions (previously defined before running each simulation), as well as gather information (per core) concerning function calls and their execution times. Below, we briefly explain how to use each of them:

1) *Built-in tracer*. The parMERASA architecture tracer is activated by modifying parMERASA configuration files, as well as setting additionally an output file (so called, dumpfile) for each core specified in parMERASA platform configuration file (param_file), as depicted in Table 7.

Table 7 Variables settings for enabling parMERASA many-core platform built-in tracer

Configuration file	Variable value
caos_debug.hpp	#define SOFTTRACER_DBG
soc_lib.conf	'-SOFTTRACER'
param_file (for 2 threads or cores)	-dumpfile0=dump.out -dumpfile1=dump.out

2) *Rapita toolset (RVS)*. A key step for using RVS in parMERASA many-core platform is to replace the original PowerPC cross-compiler (powerpc-elf-gcc), linker (powerpc-elf-ld) and assembler (as) by a set of RVS wrappers provided by Rapita. The RVS wrappers allow tracking instrumented source code (RVS lpoints), consequently used by Rapita toolset to monitor and record timing information of the target application.

4.2 WCET experimental results with the avionics applications

In this section, we introduce the WCET experimental results with the avionics applications and derive conclusions on the presented experiments.

4.2.1 3D Path Planning application

In this section, we briefly describe the parallelization configurations for the 3DPP application and our WCET measurement approach.

4.2.1.1 3D Path Planning application configurations

In the 3DPP application, data and pipeline Parallel Design Patterns (PDPs) are applied, where the input 3D obstacle map is split into several compartments. Each compartment is processed by one thread and mapped to a core in the parMERASA many-core platform. In Figure 6, we depict the size of the 3D obstacle map for each one of the parallel application configurations - 2T, 4T, 8T and 16T, respectively. Then, the processing is performed similar to a wave-front starting at corner (i.e., compartment(0,0,0)), where the border planes of neighbour compartments are necessary to process the following compartments. Therefore, using a producer-consumer pipeline, only compartments with the same Manhattan distance to compartment(0,0,0) can be processed in parallel (because of the aforementioned data dependencies). For example, in the case of 8 threads, only 4 threads are simultaneously processable in each stage (see Figure 7).

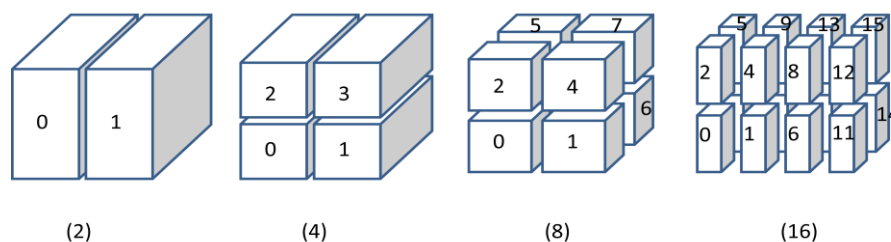


Figure 6 Various 3D obstacle map sizes for each of the 3DPP application configurations

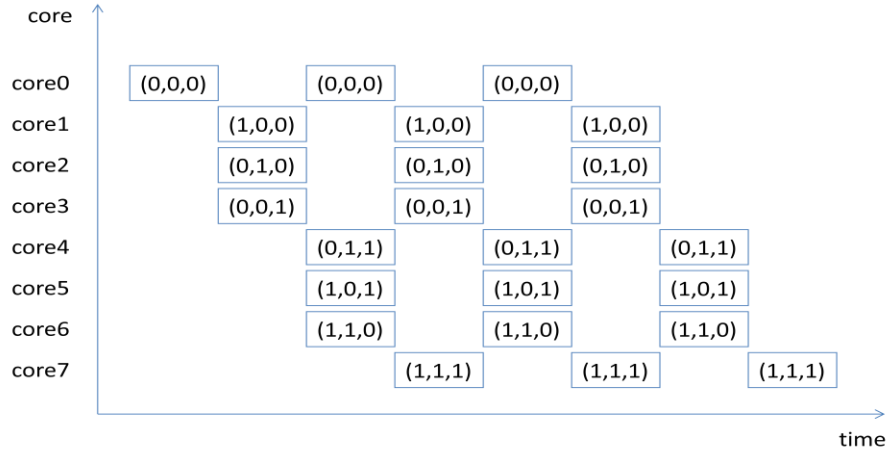


Figure 7 Producer-consumer chain in the 8T 3DPP application configuration

Given the fact that the 3DPP application exploits a data parallelism (represented by the compartments), where all the threads execute the same code, it is intuitive that the WCET of the parallel implementations of 3DPP application is determined by the thread with the longest execution time. Particularly, the WCET of 3DPP application is measured for *CompartmentThd* function call, which encapsulates the whole algorithm (i.e., computation and synchronization) to be executed by every single thread. As a result, the WCET of 3DPP application can be calculated as:

$$3DPP\ WCET = \max\{wcet_1, wcet_2, \dots, wcet_N\},$$

where N corresponds to the total number of threads for each application configuration and $wcet_1$ represents the WCET of *CompartmentThd* function call for thread 1. The next section presents different measurement techniques used in our experiments to estimate the application WCET.

4.2.1.2 Experimental results

In this section, we present the impact of the parallelization on the WCET estimation for all 3DPP application configurations. We compute the WCET, WCET speedup, and WCET efficiency using the following five techniques:

1. Obs. Simulator - observed WCET, which is collected by parMERASA simulator's built-in tracer. The Obs. Simulator indicates the highest execution time observed among all running threads. This technique is non-intrusive and generates zero cycles delay.
2. Obs. Rapita – observed WCET delivered by RapiTime toolset using the application control dataflow graph and measurement-based technique with code instrumentation. In the experiments, we evaluate the newly introduced instrumentation code overhead.
3. OTAWA – it is the WCET estimated by OTAWA tool applying static code analysis technique and a model of the target architecture.
4. Computed Rapita – it is the WCET calculated by RapiTime tool (*wcalc*) employing timing and statistical information gathered during the application execution.
5. Computed Simulator – it is a WCET calculated by aggregating the WCET of all functions invoked among all executed threads. Note that this computed value should be always equal to or higher than the WCET observed from the simulator (i.e., Obs. Simulator).

We consider that a discussion concerning the accuracy of each WCET technique is beyond the scope of our analysis. Instead, we focus our analysis on the trend followed by each of the WCET techniques. Nevertheless, we briefly analyse the reasons behind the variation in the measurements obtained with the aforementioned techniques.

3DPP WCET analysis

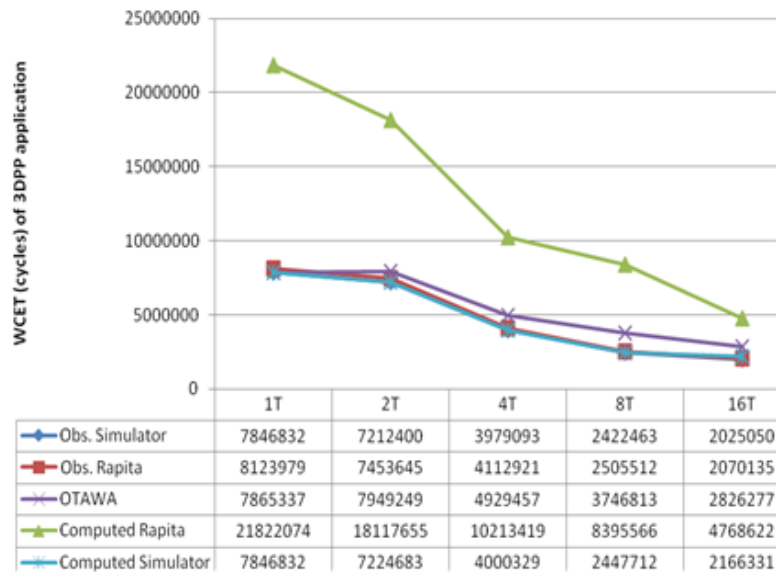


Figure 8 WCET for 3DPP application configurations

In Figure 8, we compare the **WCET** for all 3DPP application configurations. The results suggest that the WCET estimated by OTAWA is very tight (i.e. with very low conservatism) to the Observed WCET. The tightness suggests static WCET analysis is feasible for multi-core systems when using an accurate architecture model, which correctly models essential system properties, such as non-linear system behaviours when the number of cores competing to access the shared resources increases. Furthermore, the main effort spent on getting the tight WCET is in setting the initial state of the memory hierarchy, influenced by the pre-initialization procedures.

In Figure 8, the results suggest that the RapiTime overhead due to the instrumentation code is minor, just 3% higher on average. The Obs. Rapita values are based exclusively on the actual execution time obtained during the simulation. Moreover, if a target function is called multiple times in a thread and the execution time of each call is different, Rapita tool will always take into account the largest execution time for computing the observed WCET of the thread.

The WCET computed by Rapita suggests potential worst-case scenarios (i.e., not observed during our simulations) which can become 2.4 times more pessimistic (see Figure 8, 8T 3DPP application configuration) than the Obs. Simulator values. In order to compute this WCET estimate, Rapita tools first analyse the structure of the application code along with the observed execution time gathered during the simulation. From such an analysis, Rapita tools are able to identify worst-case execution path in the code which might not have been triggered during the application execution with our inputs. As a result, the Computed Rapita approach combines measurement-based analysis with the aforementioned static analysis of the source code. Although the Computed Rapita WCET estimation is higher than the obtained with other WCET estimation techniques, we would like to emphasize that the trend for the WCET remains the same throughout all experiments.

Based on the 3DPP WCET presented in Figure 8, we conclude that a high number of cores do not always lead to a high WCET decrease. The listed WCET numbers suggest that the “sweet spot” with regard to the WCET parallelization is with 4 and 8 cores/threads. The next sections analyse the reasons for the place of the sweet spot.

3DPP WCET Speedup and Efficiency

In this section, we assess the **WCET speedup** and **efficiency**. The WCET speedup is calculated as the WCET for a single thread (sequential) application configuration divided by the WCET for a multithread application configuration. For 3DPP application, the theoretical WCET speedup is initially bound up to the half of the total number of running threads, as explained in the previous section. For example, the theoretical speedup for the application configuration with 8 threads is 4 times (see Figure 7). We bound the initial theoretical speedup under the following assumptions: i) the cache hit/miss rate does not change when the number of cores is increased; ii) the workload is well balanced among all threads; iii) there is no overlapping between compartments and the size of the compartment decreases proportionally with the total number of threads; iv) the synchronization and waiting time among threads is not taken into account in the calculation of theoretical WCET speedup.

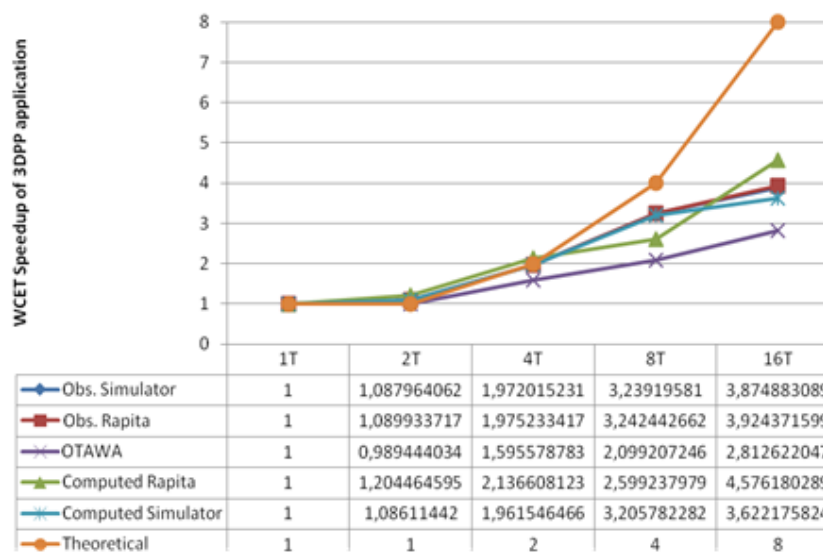


Figure 9 WCET speedup for 3DPP application configurations

Figure 9 illustrates the WCET speedup for all 3DPP application configurations. Note that the WCET speedup for each measurement technique is computed with respect to its corresponding WCET of the sequential application configuration. As the experimental numbers suggest, for 2T application configuration the Observed Simulator and Observed Rapitime WCET speedup is higher than the Theoretical WCET speedup. A similar trend is observed in 4T application configuration for the Computed Rapita. These numbers provoke an in-depth analysis on the assumptions we posed during the computation of the Theoretical WCET speedup.

As a summary, we concluded that the theoretical WCET speedup was not correctly computed, mainly because of the fact that:

- Each core has a limited cache size. For multithreaded application configurations, the total amount of cache available in the system increases proportionally with the number of threads, given that each thread is mapped onto a single core. Table 10 shows the Obs. Simulator WCET of *iteration_ph* function call, which is the most computational intensive function in the 3DPP application. We study the effect of the data cache size and cache misses on the Obs. Simulator WCET for a subset of the 3DPP application configurations – 1T, 2T, and 4T, respectively. The experimental results suggests that when a small data cache is employed (4KB), increasing the number of cores decreases the cache miss rate (and improves the WCET speedup), because much lower number of cache misses can be observed. Contrary, when we

have a large cache size and there is almost no cache misses, even in the sequential application configuration, the cache misses remains the same. Therefore, with the large cache size, we expect to have WCET speedup very close to the theoretical one but never exceeding it.

Table 8 Impact of the dcache size on the average execution time of iteration_ph function for 1T, 2T and 4T 3DPP application configurations

Cache size	1T		2T		4T	
	Obs. Simulator - WCET	Cache miss	Obs. Simulator - WCET	Cache miss	Obs. Simulator - WCET	Cache miss
4KB	6395828	19616	3189781	9281	1578209	3771
1MB	6039228	224	3025236	168	1519800	121

- There is a fine-grained overlapping among the cores and the execution is performed in a pipeline manner with overlapping of the pipeline stages during the application execution.

We define the WCET efficiency as the ratio between the WCET speedup and the number of working threads. Given the aforementioned explanation about the WCET speedup, it is clear that in a multithread implementation of the 3DPP application, the theoretical WCET efficiency should always be 0.5 (as shown in Figure 7), due to only 50% of all threads are processed in each iteration. The same assumptions used in the calculation of the theoretical WCET speedup can be applied to the theoretical WCET efficiency.

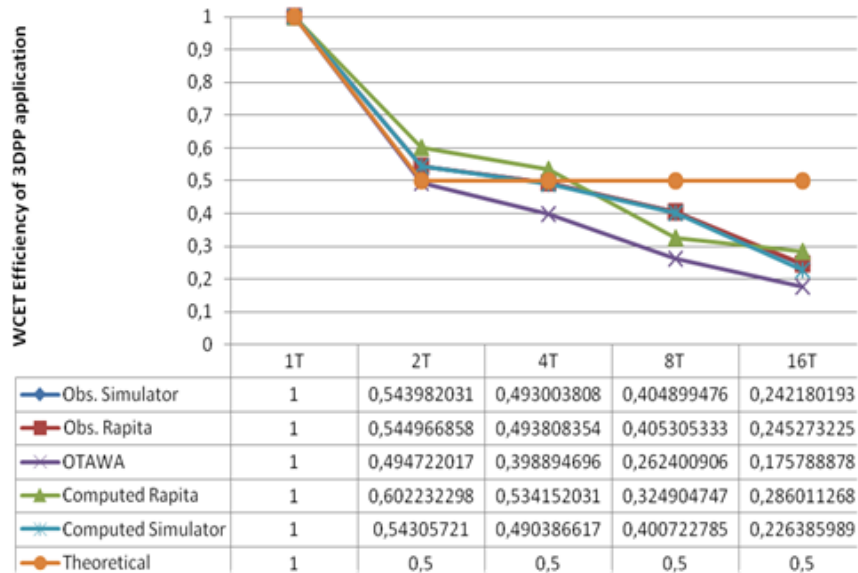


Figure 10 WCET efficiency for 3DPP application configurations

In Figure 10, we present the computed WCET efficiency. As the WCET speedup numbers suggested, the WCET efficiency for 2T and 4T 3DPP application configurations is higher than the Theoretical WCET efficiency.

We concluded that the low WCET speedup and efficiency with high number of threads is due to:

- Shared on-chip memory delays. The parMERASA multi-core platform employs “tree” type of NoC where the data travelling delay varies depending on the location of the core in the NoC.

The NoC delay also affects to the synchronization time between threads due to the shared-memory implementation.

- The overlapping between different compartments. The actual size of the compartment processed by each thread is bigger than the one used to calculate the theoretical WCET speedup. For example, while the theoretical compartment size for 16T implementation is 1/16 of the original 3D map, the actual compartment size is approximately one-eighth of the 3D map. Another consequence of the overlapping implies a simultaneous access to common memory locations while several threads read their compartment. As results, a load-unbalancing between threads can be observed. In Figure 11, we present the standard deviation of the three most computationally intensive functions in the 3DPP application with respect to the threads that they are invoked. Note that for the sequential application configuration the standard deviation is zero because of all function calls belong to the same core. The data shown in Figure 6 suggest that the variation of the computation time has a minor grow up to an 8 cores and has a steep increase with 16 cores application configuration reaching 15,9% for *iteration_ph* function call.

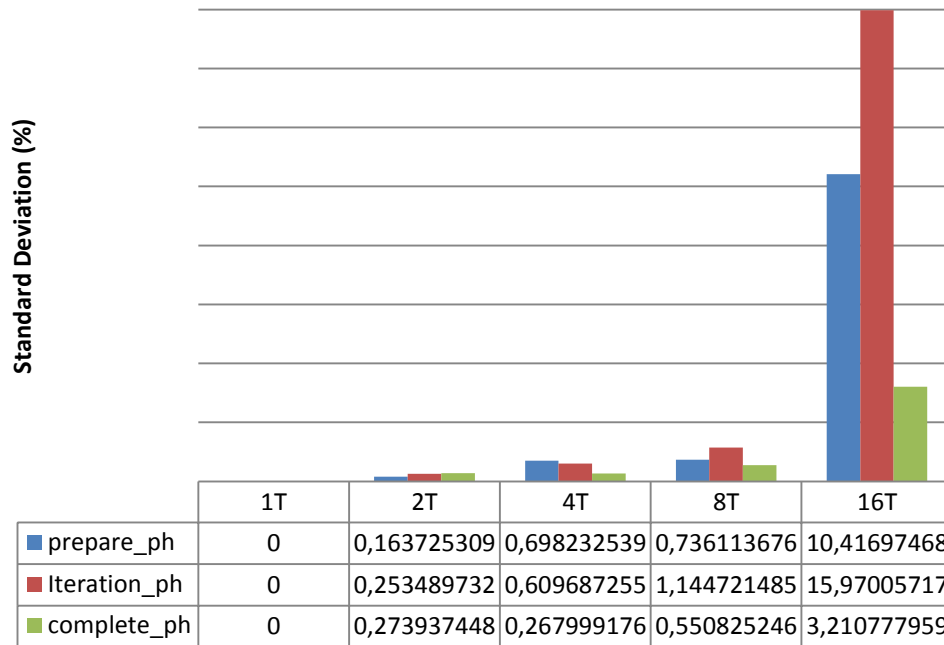


Figure 11 Standard deviation of the execution time of the most computationally intensive functions in the 3DPP application configurations

As a summary, we conclude that i) for low number of threads – 2T and 4T application configurations, the estimated and computed WCET speedup and efficiency is very close to the theoretical WCET speedup and efficiency; ii) with high number of threads, such as 16 threads, the WCET speedup is approximately 4 times, which only represents a 50% of the expected theoretical speedup. Our results also suggest that the WCET speedup and efficiency for 2T and 16T present atypical values regarding the trend values. While the former is above the theoretical value, the latter is even below the empirical trend.

3DPP application configurations with various synchronization primitives

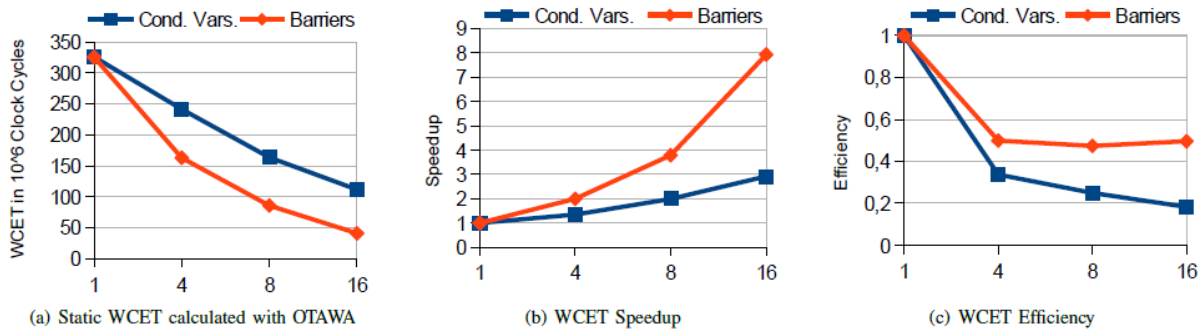


Figure 12 Synchronization primitives – conditional variables and barrier synchronization

In Figure 12, we evaluate the impact of different synchronization primitives on the 3DPP application WCET. For sake of brevity, we assume in this example only that we have a perfect cache, i.e., we ignore the contribution of the memory hierarchy contentions to the WCET. Figure 12 presents the WCET, WCET speedup, and WCET efficiency estimated by the OTAWA tool. As the number suggests, 3DPP application configurations with the barrier synchronization scales well as the number of cores increases. Therefore, we recommend that for higher WCET speedup to employ barriers instead of conditional variables.

3DPP application execution on a COTS platform

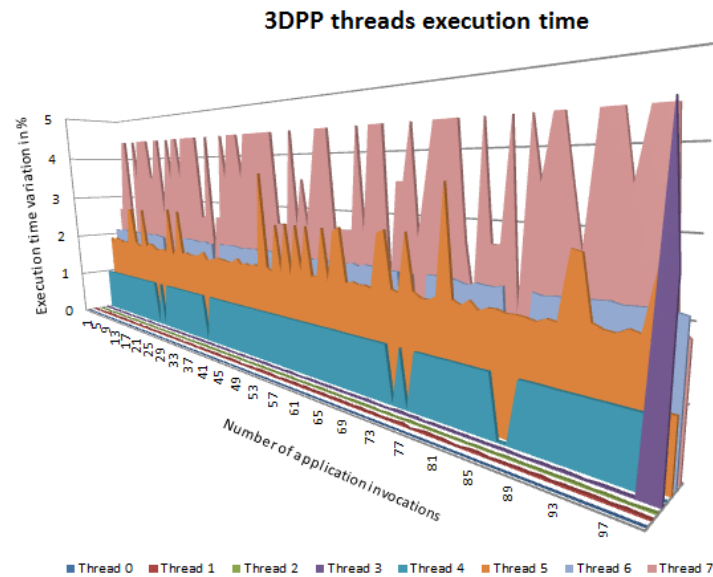


Figure 13 3DPP executed on COTS multi-core platform

Figure 13 depicts the execution time (ET) variation per thread for 8T 3DPP application configuration executed on a Commercial of the Shelf (COTS) platform with a real-time operating system (RTOS) using a preemptive scheduling. The threads in the 3DPP application are synchronized by barriers. We compute the execution time variation as $(ET - ET_{min}) * 100 / ET_{max}$, where ET is the measured execution time that each thread takes over 100 application iterations, ET_{min} and ET_{max} are the minimum and maximum of these measured thread execution times, respectively. As the presented results suggest, the variation in the execution time is minor (less than 5%) due to accesses to the shared platform resources (such as the memory hierarchy). The minor ET variation suggests that the WCET estimation of parallel applications running on COTS multi-core platforms might be achieved with minor safety margin over the ET.

Lessons learnt from 3DPP application

For real avionics application such as 3DPP, we conclude that the WCET-aware parallelization can achieve up to 4 times WCET speedup for 16 threads application configuration (see Figure 9). Since the WCET efficiency is relatively low with high number of threads, we consider the optimal application configurations to be up to 4 or 8 threads, where the WCET speedup is close to the theoretical WCET speedup. Furthermore, our analysis suggests that the parallelization should not exceed 8 threads (i.e., 8 cores), which is the sweet spot where the synchronization overhead (and memory access delay in parMERASA platform) becomes dominant factor in the execution time. The low WCET efficiency with high number of threads might be compensated by applying a preemptive operating system, which schedules other threads on the same processors. Note that all presented WCET estimations on the parMERASA were based on simple mapping scheme, where 1 thread runs on 1 core. Furthermore, our WCET analysis suggests that the barrier synchronization primitives are better than the conditional variables.

In summary, we conclude that the overall WCET is strongly influenced by:

- the ratio between computation time and synchronization;
- the parallelization granularity and load-balancing of the code decomposition among the cores, and;
- the size of the local (per core) data cache;

All the aforementioned factors jointly affect the execution time of each core. In case of 2 threads implementation, the dominant factor causing an empirical WCET higher than the theoretical one is the cache miss rate, because the application workload is well balanced and both threads are statically mapped on the top of tree-NoC. As for the 8T 3DPP application configuration, we observe a poor scaling of the application as increase the number of threads/cores, since data traveling time becomes increasingly significant for those cores located in the lower level of tree-NoC. This scalability issue is even more noticeable for the 16 threads implementation due to the load-unbalancing between different threads.

4.2.2 Stereo Navigation application

In this section, we briefly describe the parallelization configurations for the SN application and our WCET measurement approach.

4.2.2.1 Stereo Navigation application configurations

We first introduce the SN application configurations evaluated in our experiments, and subsequently, we state the WCET measurement strategy employed in the SN WCET experimental results.

The SN application is intended for aircraft localization when there is loss of the Global Navigation Satellite System operation. Basically, SN application receives periodically frames composed of two independent images (namely left and right images), and subsequently, applies a set of image processing techniques, such as the extraction of special features, features processing and representation in a 3D space. At the end of the SN application execution, the aforementioned 3D re-projected points from two consecutive frames (for instance, frame 1 in t_0 and frame 2 in t_1 , as depicted in Figure 14) are used by SN application to estimate a vehicle relative movement during the time interval $[t_0, t_1]$ based on relative positions of these two sets of re-projected feature points.

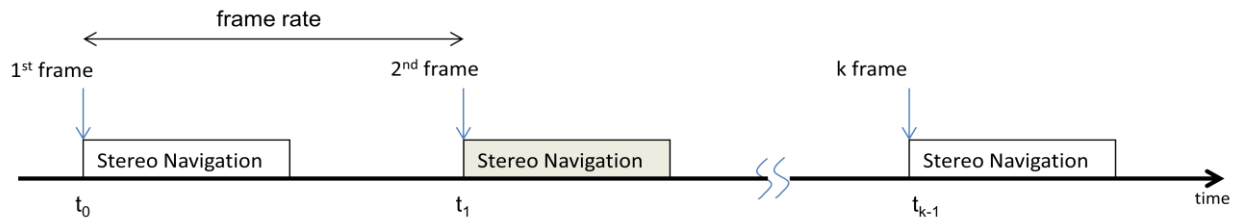


Figure 14 SN application estimates the vehicle movement from the mutual position change of features extracted from two consecutive frames

In parMERASA project, we compare the WCET, WCET speedup, and WCET efficiency for a sequential application configuration with a set of multi-thread implementations. A detailed description of the SN application functionality is provided by parMERASA deliverable D2.1. Moreover, parMERASA deliverable D2.4 also presented an optimized/parallelized version of the SN application. More precisely, we split and grouped the functions into five different pipelining stages, where additional parallelism might be exploited for a particular function (inside a stage) according to the application configuration. As a result, we prepared five multi-thread application configurations (the same as the ones presented in parMERASA deliverable D2.4):

- 7 threads SN application configuration (7T). In the stage 1, the Feature Extraction function is further split into three sub-tasks, resulting in an additional three-stage pipeline inside the stage 1. Therefore, a total of seven threads/cores are run in this application configuration, i.e., three threads/cores for stage 1 –Feature Extraction function– and one thread/core for each of the rest pipelining stages –stage0, stage2, stage3 and stage4–.
- 12 threads SN application configuration (12T_CAM). This application configuration exploits the data parallelism by means of processing two independent images (i.e., left and right images) in parallel. This means, for example, two threads/cores are used in stage 0, i.e., one thread/core per image processing. Note that six threads/cores are assigned to stage 1 (Feature Extraction function), in order to exploit simultaneously both data parallelism and the aforementioned sub-tasks level parallelism.
- 12 threads SN application configuration (12T_CONV). The Feature Extraction function mainly consists of eight convolutions operations processing the images with different filters. This application configuration exploits the parallelism of mapping each convolution to a dedicated thread/core. As a result, eight threads/cores are assigned to stage 1, while the rest of pipelining stages uses one thread/core each.
- 14 threads SN application configuration (14T). This application configuration also exploits simultaneously both data parallelism and sub-tasks level parallelism of Feature Extraction function in stage 1, as well as exploits data parallelism of Splitting To Tiles, Feature Matching, 3D Reprojection and Robust Pose Estimation functions in stage 0, stage 2, stage 3 and stage 4, respectively.
- 16 threads SN application configuration (16_CAM). There are two main differences between this application implementation and the aforementioned 14T configuration. First, 16T configuration applies data parallelism to the whole stage 0 (i.e., to both Rectification and Splitting To Tiles functions). Second, four dedicated threads/cores are assigned to Robust Pose Estimation Function in stage 4. As a result, the number of threads/cores per stage is 2 (stage 0), 6 (stage 1), 2 (stage 2), 2 (stage 3) and 4 (stage 4).

For the SN application, we measure the end-to-end WCET, which represents the WCET required by the SN application to process a frame and outputs the desired results (see Figure 14). The end-to-end WCET includes the maximum WCET of all stages as well as the required synchronization between different stages.

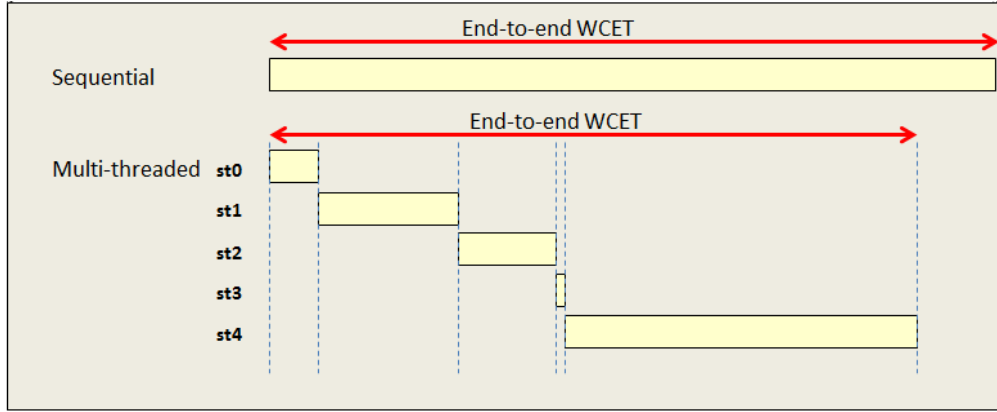


Figure 15 End-to-end WCET measurement for SN application configurations

The next section analyses the end-to-end WCET. Particularly, we compare the WCET, WCET speedup and WCET efficiency for one of the most computation intensive stage - Feature Extraction stage (denoted as **st 1** in Figure 14). We measure the WCET following the very same techniques as in 3DPP application, namely:

1. Obs. Simulator - observed WCET, which is collected by parMERASA simulator's built-in tracer. The Obs. Simulator indicates the highest execution time observed among all running threads. This technique is non-intrusive and generates zero cycles delay;
2. Obs. Rapita – observed WCET delivered by RapiTime toolset using code instrumentation and measurement-based technique. The newly introduced instrumented code has a profiling delay;
3. OTAWA – it is the WCET estimated by OTAWA tool applying static code analysis technique and a model of the target architecture;
4. Computed Rapita – it is the WCET calculated by RapiTime tool (*wcal/c*) employing timing and statistical information gathered during the application execution;

In Figure 15, we depict the end-to-end Obs. Simulator WCET for all SN application configurations: Seq, 7T, 12T_CAM, 12T_CONV, 14T and 16_CAM, respectively. For sake of simplicity, we employ different colours to present the execution time of each SN application processing stage. The number within each coloured stage indicates the number of threads/cores employed. Note that because of the additional three-stage pipeline placed inside the stage 1 (as explained in the previous section), three numbers are used to denote the number of threads/cores assigned to each stage of this inner pipeline. For example, in the case of 12T_CONV configuration, “2 3 3” indicates that two threads/cores are used in the first stage of the inner pipeline, three in the second and three in the last stage. Thus, a total of eight threads/cores are assigned to the Feature Extraction function in stage 1.

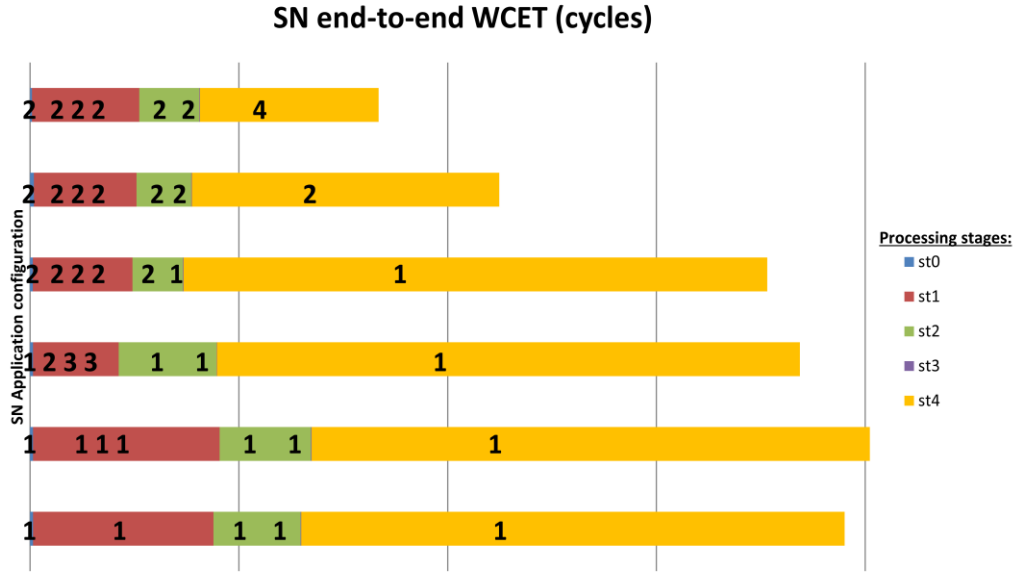


Figure 16 End-to-end WCET (in cycles) for SN application configurations

In Figure 16, one might notice that there is workload unbalancing among the stages. For example, the execution time of stage 0 and stage 3 is not even visible in Figure 16. A way to balance the workload would be merging one stage to another. Note that the analysis of such an optimization in the workload balancing is not conducted in our experiments.

The results presented in Figure 16 also reveal that the WCET of 7T SN application configuration experiences an increase because of no parallelism is exploited. In 7T SN application configuration, all threads still need to be executed sequentially (just like in the sequential implementation), incurring in additional synchronization overhead between two adjacent pipeline stages.

Nevertheless, the rest of multi-thread configurations (i.e., 12T_CONV, 12T_CAM, 14T and 16_CAM), has a noticeable WCET reduction as a result of the application parallelization. For instance, the Obs Simulator WCET for 16_CAM SN application configuration is reduced to more than half of the end-to-end execution time of sequential implementation. However, although the overall end-to-end WCET falls down as the number of threads/cores increases, we cannot conclude in the same way for all pipelining-stage WCET. For example, in 12T_CAM, 14T and 16_CAM application configurations, stage 1 (i.e., Feature Extraction) has the same parallelization scheme. The WCET of stage 1 indicates a growing trend (see Figure 17) in the WCET, which is a result from increasing the total number of threads/cores employed throughout the entire application.

In the next section to follow, we prove further WCET analysis for the SN stage 1, which we consider as one of the three most computationally intensive stages. We did not perform WCET analysis on stage 4, because its implementation has probabilistic analysis and we expect very high overestimation compared to the achieved with Obs. Simulator technique.

SN WCET analysis

In Figure 17, we present WCET analysis of **st 1** (i.e., Feature Extraction function) for all SN application configurations. For both Obs. Simulator and Obs. Rapita, we employ the left side Y-axis resolution, while for Computed Rapita, we employ the right side Y-axis resolution. The reason for double Y-axes resolution is because Computed Rapita WCET is much higher and more pessimistic than the Obs.

Simulator WCET. It is important to point out that the aforementioned pessimism only tempts to consider the difference between the maximum observed execution time and the calculated WCET. However, while no guarantee is given that the worst-case path has been exercised (and observed) during measurements, *this difference does not mean anything about the accuracy of the WCET estimate*. At best, it could tell how far the calculated WCET might be from what has been observed.

Nevertheless, while the pessimism introduced for 3DPP application was reasonable, the explanation behind this significant pessimism calculated for SN application is twofold:

- 1) The use of barriers for multi-core synchronisation in SN application was found to be incompatible with the updated approach used by Rapita for calculating WCET and WCWT for barriers. Calculating these timings requires all barrier calls across all cores to be in the same sequence. In the Stereo Navigation application, this was not the case, as some barriers were shared by a subset of threads, and other sets of barriers shared by different subsets.
- 2) Some elements of the code in SN application were written in such a way which made a non-pessimistic analysis extremely difficult. The elements of the code which contribute the most to the worst case timing of the application contain multiple nested loops, which are very difficult to analyse for Rapita toolset.

Despite the issues described above, Rapita toolset is able to calculate timing information for SN application. *However*, these times are likely to be either optimistic or pessimistic. This is due to the inability of Rapita toolset to analyse the synchronisation primitives which were used in the application (as described above). The result of this is that any part of the code which waits at a synchronisation primitive simply uses the observed waiting time as a basis for a worst case calculation, rather than doing a detailed analysis to calculate a potentially more optimistic or pessimistic one.



Figure 17 WCET of Feature Extraction function (st 1) for all SN application configurations

From the results introduced in Figure 17, we conclude that all measurement techniques suggest the same WCET performance behaviour as the number of threads/cores increases. Particularly, the observed WCET with Obs. Rapita is, in average, 3.2% higher than the Obs. Simulator WCET. As already mentioned previously in the results analysis section for 3DPP application, such a difference is mainly because of the overhead introduced by the instrumentation points (Rapita I-points) in the

source code, which are needed by Rapita toolset to collect the essential information during the simulation for providing further both the observed and calculated WCET.

The WCET achieved with 12T_CAM SN application configuration is higher than the obtained with 12T_CONV application configuration. Although, both application configurations have 12 threads, their WCET comparison is not straightforward because of:

- 1) 12T_CAM SN application configuration exploits data parallelism (i.e., processing independently left and right images), while 12T_CONV configuration exploits the parallelism of convolution operations carried out by Feature Extraction function;
- 2) 12T_CONV SN application configuration exploits a parallelism based on eight threads/cores, while 12T_CAM configuration only requires 6 threads/cores.

Therefore, for a better understanding of the impact of the total number of threads/cores on the WCET of Feature Extraction function, we suggest to compare 12T_CAM, 14T and 16_CAM SN application configurations. In SN stage 1, all these application configurations share the same type of parallelization. As the results suggest in Figure 17, the 12T_CAM SN application configuration is an inflection point from which an increased number of threads in any of the other SN processing stages results in a WCET increase in **st. 1**. As a result, we expect that higher parallelization might result in an application end-to-end WCET increment.

For sake of brevity, Figure 18 only depicts the WCET estimated by OTAWA tools for all SN application configurations. The solid line represents the actual OTAWA estimates (i.e., Seq, 7T and 12T_CONV SN application configurations), while the dashed line indicates the projected OTAWA estimates following the same trend demonstrated with the Obs. Simulator, Obs. Rapita and Computed Rapita.

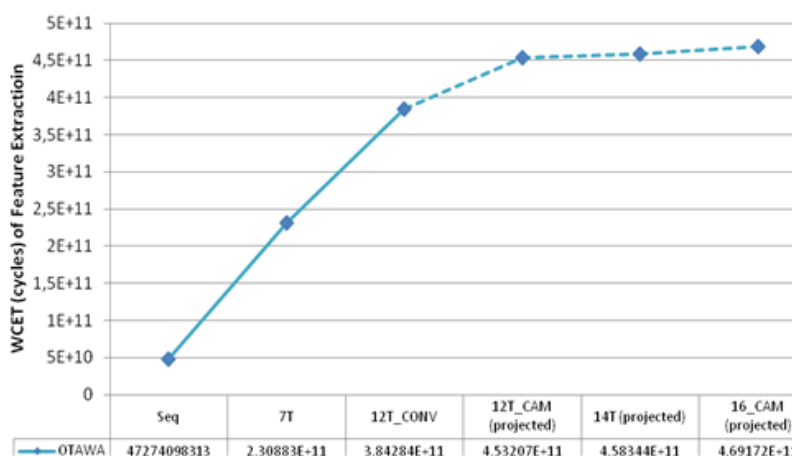


Figure 18 WCET of Feature Extraction function (st 1) estimated by OTAWA for all SN application configurations

In Figure 18, The WCET estimates suggest a pessimistic and growing trend as the number of threads/cores increases, comparing to the Obs. Simulator WCET. Particularly, the overestimation goes from 55 times for sequential implementation to 258 times and 950 times, for 7T SN configuration and 12T_CONV SN configuration, respectively. Such an overestimation is tightly influenced by the pre-initialization of the memory hierarchy, e.g., the time required to fill out the cache memories. Particularly, the growing trend suggests that the analysis of pre-initialization

procedure is not done precisely by OTAWA tool, therefore the static WCET computation yields to hugely over-estimated cache miss counts. When using higher number of cores, this effect is combined with the increasing worst-case memory access times, and consequently, canceling the potential benefits of parallelization. By conducting a precise analysis on the aforementioned pre-initialization phase, a tight WCET estimate is expected from OTAWA tool, as already shown for 3DPP application.

SN WCET Speedup and Efficiency

In this section, we define the theoretical WCET speedup, and compare WCET **speedup** and **efficiency** among different SN application configurations. Particularly, we conduct analysis on **st 1** (Feature Extraction). The theoretical WCET speedup and efficiency are calculated by assuming that: i) the cache hit/miss rate does not change when the number of cores is increased ii) the synchronization and waiting times between threads are neglected. As a result, the theoretical WCET speedup and efficiency are calculated as follows:

- 7T SN application configuration: no parallelism is exploited in this configuration, since the Feature Extraction function is simply split into three stages, which are executed sequentially. Therefore, the expected Theoretical WCET speedup (ignoring the synchronization and communication overheads) is the same as the sequential implementation, i.e., 1x.
- 12T_CAM SN application configuration: data parallelism is exploited by Feature Extraction functions, i.e., two threads/cores are employed for each of the aforementioned inner pipeline stages inside the stage 1. Assuming that the total execution time of Feature Extraction function (namely X) can be decomposed for each inner stage, such as X1, X2 and X3. The theoretical speedup is defined as: $X/((X1/2)+(X2/2)+(X3/2)) = 2x$.
- 12T_CONV SN application configuration: this configuration exploits the parallelism at sub-tasks level of convolutions. Particularly, the number of convolutions per inner stage is respectively 2, 3 and 3. Therefore, the theoretical speedup can be calculated as: $X/((X1/2)+(X2/3)+(X3/3)) = 2,737x$.
- 14T and 16_CAM SN application configuration: since these configurations exploit the same kind of data parallelism used in 12T_CAM configuration, we then can conclude that the theoretical speedup for these configurations is 2x.

The calculation of Theoretical WCET efficiency for stage 1 (Feature Extraction function) is straightforward. Taking into account the number of threads/cores assigned to stage 1, the theoretical efficiency can be obtained dividing the theoretical speedup by the number of threads/cores. Therefore, the Theoretical WCET efficiency is as follows:

- Seq = 1;
- 7T = $1/3=0,33$;
- 12T_CAM = $2/6=0,33$;
- 12T_CONV = $2,737/8=0,342$;
- 14T = $2/6=0,33$;
- 16_CAM = $2/6=0,33$;

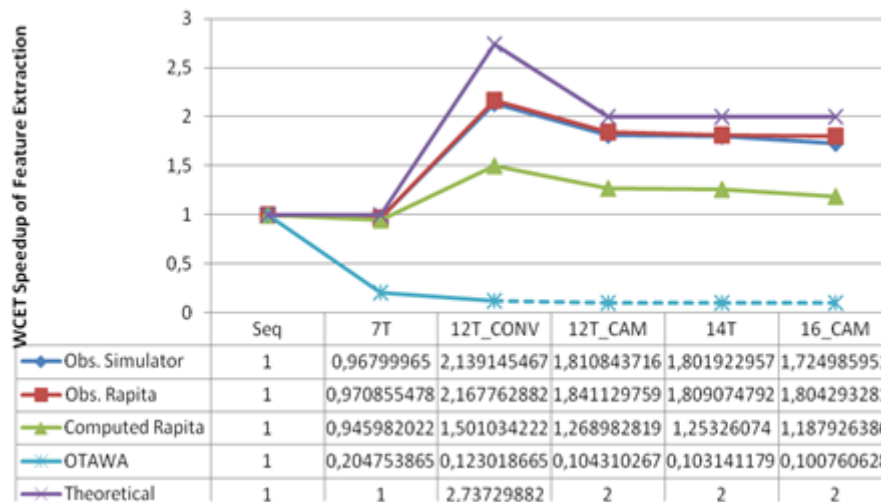


Figure 19 WCET speedup of Feature Extraction function (st 1) for SN application configurations

Figure 19 and Figure 20 illustrate the WCET speedup and WCET efficiency of stage 1 (Feature Extraction function), respectively. The dashed line indicates projected WCET estimated by OTAWA tool, as described in Figure 18. The presented results again indicate that there is no benefit using a simple pipeline parallelism (i.e., 7T) from the speedup and efficiency point of view, since a slowdown can be observed as a consequence of communication overhead incurred by the inter-threads synchronization. On the other hand, configuring the SN application with 12 threads/cores, 12T_CONV configuration provides a higher speedup (2,139x) but at cost of a lower efficiency (0,267) with respect to 12T_CAM configuration (1,81x and 0,302, using as a reference the Obs. Simulator measurement). It should be noted that eight threads/cores are assigned in the former, while the latter uses only six threads/cores.

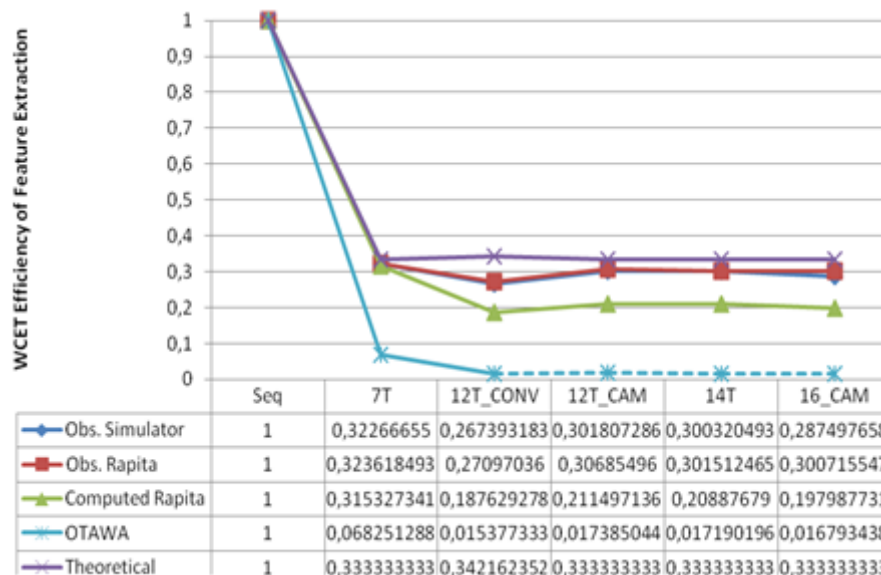


Figure 20 WCET efficiency of Feature Extraction function (st 1) for SN application configurations

We compare the WCET speedup and efficiency of Feature Extraction function among SN 12T_CAM, SN 14T and SN 16_CAM application configurations. From the WCET results, we conclude that the improvement in the end-to-end WCET of SN application by going from 12T_CAM to 16_CAM (as shown in Figure 17), is being partially countered by the degradation of WCET taking in place inside each processing stage. The justification of this latter would be the system has already reached to a saturation point due to the shared resources contention induced by a more intensive competition

among the cores. Therefore, we conclude that the sweet spot for SN application is with 12 threads. Note that the speedup and efficiency obtained for OTAWA WCET estimates is lower due to the overestimated values used in this calculation.

Lessons learnt from SN application

The computational complexity of SN application multiple times exceeds the 3DPP application. Such high SN computational complexity leverages a full advantage of the multi-threading by exploiting data and tasks (i.e., pipelining) parallelism. For this purpose, we first split the SN application into five processing stages, where additional parallelization (e.g., data parallelism and/or sub-task parallelism) are further applied to the tasks of each stage according to their computational characteristics. As a result, six configurations of SN application have been analysed in parMERASA project. Our WCET experimental results suggest that no improvement in SN end-to-end WCET is observed for SN 7T application configuration, because all threads are still executed sequentially with communication overhead caused by the inter-thread synchronization. As a result, the end-to-end WCET for 7T application configuration is higher than the end-to-end WCET for the sequential implementation of SN application.

On the other hand, an improvement of the end-to-end WCET is feasible by increasing the parallelization of SN application (e.g., with 16_CAM SN application configuration). However, our analysis also reveals that there is always a trade-off between application end-to-end and per pipeline stage WCET improvements. Therefore, for significant application end-to-end WCET speedup optimization, we recommend parallelization on processing stages, which have a major contribution to the end-to-end application WCET and operates under the sweet spot.

5 AUTOMOTIVE APPLICATIONS

In Deliverable 2.4 describes a parallelization approach tailored to automotive applications. First evaluation results and possibilities for optimization are presented. This section presents the final evaluation results for the parallelization of a diesel engine management system (EMS) performed during the parMERASA project.

Automotive applications allow the extraction of parallelism at three levels. *Inter-task* parallelism exploits the parallel execution of tasks, see Figure 21. The benefit is task bodies and Runnables can remain the same on the multi-core platform. The application tasks are scheduled on the available cores. AUTOSAR already supports the parallel execution of tasks. But, interferences caused by shared resources often make such implementations inefficient. We present an approach to overcome these limitations. A speed-up of up to 4.5 is observed.

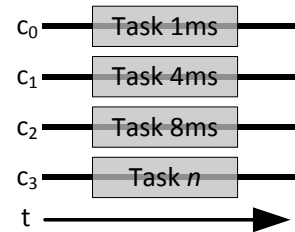


Figure 21 Illustration of inter-task parallelism in automotive applications

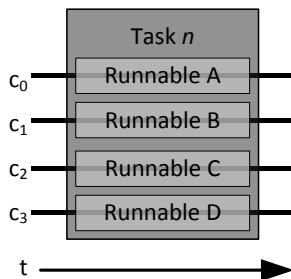


Figure 22 : Illustration of intra-task parallelism in an AUTOSAR task.

Intra-task parallelism distributes the Runnables of a task² over the available cores, see Figure 22. The benefit of this approach is the task scheduling and Runnables can remain unchanged on the multi-core platform. The parMERASA Mapping Tool is introduced to distribute Runnables in that sense. We observe a speed-up of 3.33 for a single task. *Supertasks* are introduced to improve the speed-up of tasks scheduled to the same point in time. We observe a speed-up of 2.66 instead of 2.01 with this improvement.

The most fine-grained level of parallelism can be found inside a Runnable. In *intra-Runnable* parallelism single functions or instructions are distributed over the available cores, see Figure 23. AUTOSAR does not support parallelization at such a fine-grained level as Runnables are the smallest schedulable entity. Nevertheless, the advantages of this approach are twofold. First, the task scheduling can be preserved. Second, the WCET of frequently used Runnables or the length of the critical path can be reduced. A speed-up of 2.3 is observed for two time critical Runnables.

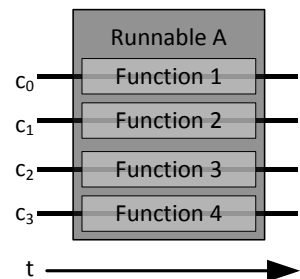


Figure 23: Illustration of intra-Runnable parallelism.

We used the tools provided by WP3 to analyse, parallelize, and measure a diesel EMS. OTAWA has been used for analysis and parallelization. RapiTime PARMERASA has been used for measurements with the parMERASA simulator provided by WP5 executing the EMS application on top of the systems software (tiny automotive RTE) provided by WP4.

This section is structured as follows. Section 5.1 briefly described the examined application. Section 5.2 describes the parMERASA Mapping Tool for the extraction of intra-task parallelism. Moreover, the supertasks are introduced. Section 5.3 presents timed implicit communication as approach for

² here task is meant in the sense of Runnables that are executed in the same task body of the legacy single core configuration due to similar priority and activation event. In a strict AUTOSAR sense, Runnables distributed over several cores do not belong to the same task any more.

the extraction of inter-task parallelism. Section 5.4 describes the parallelization of single Runnables. Finally, Sections 5.5 gives a combined approach.

5.1 Automotive application

We use a diesel EMS as example application, because this is a typical real-time control application from the automotive domain. Almost all road vehicles have it in common. Moreover, the application is sufficiently complex and computational intensive to (potentially) gain benefit from parallelization. On one hand, a sensor at the camshaft of the engine triggers an interrupt every 30° of a revolution and, on the other hand, the status of a set of sensors is actively sampled periodically (*polling*). The examined engine control contains eleven periodic tasks (1, 4, 5, 8, 16, 20, 32, 64, 96, 128 and 1024ms) and one sporadic task (triggered by the camshaft sensor; denoted as *crBas*). We assume maximum 4000 revolutions per minute and 12 interrupts per revolution (every 30°). This leads to a minimum period of 1.25ms for *crBas*. Further details about the examined application can be found in Deliverable 2.1.

5.2 Intra-task parallelism

This section describes the parMERASA Mapping Tool for the exploitation of intra-task parallelism for automotive applications (with task we mean AUTOSAR task). It has been developed in close collaboration of the project partners DENSO and BSC. A first prototype implementation is described in the Prototype Deliverable 2.3. First performance evaluation of the approach is conducted in Deliverable 2.4. Furthermore, results are published in [2]. In the following we briefly describe previous findings and explain the improvement by the introduction of supertasks.

5.2.1 Mapping Runnables

A task comprises a set of Runnables, which are executed in a fixed order (control flow) on the *single-core ECU* (SCE). Parallelizing the execution requires to respect data dependencies between Runnables, that means the order in which they access shared memory locations. Therefore, precedence constraints among Runnables are detected by static analysis of the code in combination with the control flow. The applied technique is similar to the presented in [3] used by automatic parallelizing compiler. Dependencies are summarized in a graph $G = (V, E)$, where each node $v \in V$ denotes a Runnable and each edge $e \in E$ denotes a precedence constraint. This graph is generated for every task and it is scheduled afterwards. This approach also respects data dependencies caused by calls to a server-Runnable, which typically maintain an internal state. The mapping is based on a variant of the worst-fit decrease heuristic, which prioritizes tasks with higher combined utilization. A static schedule is defined for every task. The allocation time of a Runnable on a core is bound by its WCET (calculated with OTAWA provided by WP3). The advance instruction provided by the tiny automotive RTE (WP4) is used to guarantee the WCET is always reached. This introduces synchronization points and this makes it possible to execute tasks with low overhead even for a large number of cores. The latter one is also investigated in Deliverable 2.4.

This approach has several advantages. First, the task scheduling can remain unchanged on the multi-core processor, because only one task executes. Second, server-calls do not need to be protected, because data dependencies are respected as precedence constraint between Runnables accessing the same server-Runnable. Third, the synchronization overhead is low.

5.2.2 Supertask

The evaluation of the parMERASA Mapping Tool in Deliverable 2.4 proposes a potential improvement. Under unfavourable conditions, precedence constraints can force sequential execution of Runnables of the same task. Therefore, we propose to allow interleaving of tasks scheduled to the same point in time. Figure 24 illustrates the interleaving of two tasks (n and m) forming a supertask. The Runnables E, F, and G in task m are not delayed until task n has finished execution. They can execute earlier, if no precedence constraint with the Runnable D in task n exists.

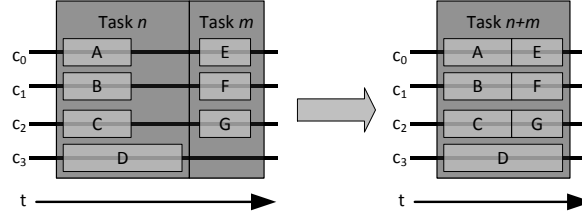


Figure 24: Interleaving of two tasks called supertask.

5.2.3 Quantitative Analysis

The parMERASA Mapping Tool is evaluated in two steps. First, the performance for every task is investigated. The improvement with supertasks is presented afterwards.

5.2.3.1 Setup

We consider a 2- and 4-core parMERASA processor with tree NoC. Each core has a private instruction scratchpad that contains the statically mapped code (Runnables) executed on this core. Moreover, every core has an 8-way data cache with a size of 256KB. The NoC and the on-chip memory are sources of interference. Given the tree traversal time L_t , the memory latency L_m , and the number of cores c the *upper bound delay* (UBD) for a memory request is $UBD = L_t + (c - 1) \cdot L_m$. On a 2-core processor we consider $L_t = 1$ cycle and $L_m = 10$ cycles. Hence, $UBD = 11$. On a 4-core processor we consider $L_t = 2$ cycle and $L_m = 10$ cycles. Hence, $UBD = 32$. For each router we consider a round-robin arbitration policy, which makes L_t independent from the number of cores. A memory request is stalled by all other cores in the worst-case. OTAWA has been configured with this setup and the WCET has been calculated for each Runnable and task.

5.2.3.2 Results Task

Table 9 shows the speed-up and efficiency for every task of the EMS parallelized with the parMERASA Mapping Tool using 2 cores. In addition to Deliverable 2.4 the crank-angle task has been added. On one hand, a perfect speed-up of 2 is achieved for τ_8 . The tasks τ_{crBas} and τ_{32} also provide a high speed-up with 1.9 and 1.8, respectively. But, the tasks τ_5 and τ_{128} achieve only a small speed-up, on the other hand. This illustrates that the achievable speed-up heavily depends on the number of Runnables and the data dependencies among them.

Task	WCET _{sequential}	WCET _{parallel}	Speed-up	Efficiency
τ_1	69358	64072	1.08	0.54
τ_4	242392	143846	1.69	0.84
τ_5	8079	8079	1.00	0.50
τ_8	160377	80199	2.00	1.00
τ_{16}	539341	309319	1.74	0.87
τ_{20}	41850	25070	1.67	0.83
τ_{32}	405005	227056	1.78	0.89

τ_{64}	85962	58846	1.46	0.73
τ_{96}	65192	52423	1.24	0.62
τ_{128}	250317	219311	1.14	0.57
τ_{1024}	312811	252450	1.24	0.62
τ_{crBas}	538714	291084	1.85	0.93

Table 9: Intra-task speed-up for each task of the EMS using 2 cores

The speed-up on a 4-core processor has been investigated in a subsequent step. Table 10 summarizes the results. The highest improvement is provided by τ_8 where the speed-up increases from 2.0 to 3.33. The efficiency is lower or equal to 0.5 for all tasks with the exception of τ_8 . Almost all tasks profit from two additional cores, albeit sometimes only to a small extent. No improvement is observed for τ_5 and τ_{128} . The reason is, the number of Runnables is too small to utilize all cores or precedence constraints prevent parallel execution of Runnables. One way to overcome this issue is the combination of tasks to supertasks. This is investigated in the following section.

Task	WCET _{sequential}	WCET _{parallel}	Speed-up	Efficiency
τ_1	104260	95845	1.09	0.27
τ_4	371453	210032	1.77	0.44
τ_5	12426	12426	1.00	0.25
τ_8	249165	74842	3.33	0.83
τ_{16}	840580	422562	1.99	0.50
τ_{20}	65412	32749	2.00	0.50
τ_{32}	612863	322600	1.90	0.48
τ_{64}	132771	84462	1.57	0.39
τ_{96}	102593	82300	1.25	0.31
τ_{128}	391206	342665	1.14	0.29
τ_{1024}	469303	379605	1.24	0.31
τ_{crBas}	833225	437248	1.91	0.48

Table 10: Intra-task speed-up for each task of the EMS using 4 cores

5.2.3.3 Results Supertasks

For the evaluation of the supertask approach all possible task combinations have been evaluated. The results for 2 cores are listed in Table 11. The column PT lists the speed-up for parallelized tasks (previous section) executed in sequential order. The column ST lists the speed-up when task interleaving is applied in the form of a supertask. All supertasks provide a high speed-up with the exception of 23. The reason is τ_5 contains only one Runnable. The values for efficiency are greater than 0.78 for all other supertasks, which is a very good result.

ID	Supertask	PT	ST	Efficiency
1	$\tau_1 + \tau_4$	1.50	1.65	0.82
2	$\tau_1 + \tau_4 + \tau_5 + \tau_{20}$	1.50	1.83	0.92
3	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20}$	1.68	1.81	0.90
4	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32}$	1.71	1.80	0.90
5	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64}$	1.69	1.78	0.89
6	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{128}$	1.59	1.65	0.83
7	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{128} + \tau_{1024}$	1.52	1.61	0.81
8	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{96}$	1.67	1.75	0.87
9	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{96} + \tau_{128}$	1.57	1.63	0.82

10	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{96} + \tau_{128} + \tau_{1024}$	1.51	1.56	0.78
11	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{96}$	1.68	1.77	0.88
12	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{20}$	1.62	1.88	0.94
13	$\tau_1 + \tau_4 + \tau_8$	1.64	2.00	1.00
14	$\tau_1 + \tau_4 + \tau_8 + \tau_{16}$	1.69	1.89	0.95
15	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32}$	1.72	1.64	0.82
16	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64}$	1.70	1.77	0.88
17	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{128}$	1.54	1.75	0.88
18	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{128} + \tau_{1024}$	1.49	1.64	0.82
19	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{96}$	1.68	1.74	0.87
20	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{96} + \tau_{128}$	1.57	1.62	0.81
21	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{96} + \tau_{128} + \tau_{1024}$	1.51	1.55	0.78
22	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{96}$	1.84	1.82	0.91
23	$\tau_1 + \tau_5$	1.07	1.15	0.58

Table 11: Intra-task speed-up for supertasks using 2 cores

Using 4 cores for the scheduling the Runnables of supertasks improves the speed-up slightly. The results for this experiment are listed in Table 12. We observe 12 out of 23 supertasks achieve a speed-up greater than 2. The highest speed-up is achieved by parallelizing supertask 22 on 4 cores resulting in a 2.66 times shorter WCET. On an overall basis, supertasks profit more from two additional cores than plain tasks.

ID	Supertask	PT	ST	Efficiency
1	$\tau_1 + \tau_4$	1.56	1.63	0.41
2	$\tau_1 + \tau_4 + \tau_5 + \tau_{20}$	1.58	1.90	0.48
3	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20}$	1.94	2.08	0.52
4	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32}$	1.93	2.03	0.51
5	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64}$	1.90	2.00	0.50
6	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{128}$	1.74	1.81	0.45
7	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{128} + \tau_{1024}$	1.64	1.73	0.43
8	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{96}$	1.86	1.95	0.49
9	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{96} + \tau_{128}$	1.72	1.78	0.45
10	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{64} + \tau_{96} + \tau_{128} + \tau_{1024}$	1.63	1.68	0.42
11	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{16} + \tau_{20} + \tau_{32} + \tau_{96}$	1.88	1.98	0.50
12	$\tau_1 + \tau_4 + \tau_5 + \tau_8 + \tau_{20}$	1.88	2.19	0.55
13	$\tau_1 + \tau_4 + \tau_8$	1.90	2.25	0.56
14	$\tau_1 + \tau_4 + \tau_8 + \tau_{16}$	1.95	2.18	0.55
15	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32}$	1.93	2.01	0.50
16	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64}$	1.91	2.40	0.60
17	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{128}$	1.70	2.08	0.52
18	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{128} + \tau_{1024}$	1.61	1.88	0.47
19	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{96}$	1.87	2.31	0.58
20	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{96} + \tau_{128}$	1.72	2.02	0.51
21	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{64} + \tau_{96} + \tau_{128} + \tau_{1024}$	1.62	1.85	0.46
22	$\tau_1 + \tau_4 + \tau_8 + \tau_{16} + \tau_{32} + \tau_{96}$	2.01	2.66	0.67
23	$\tau_1 + \tau_5$	1.08	1.13	0.28

Table 12: Intra-task speed-up for supertasks using 4 cores

5.2.4 Summary

In this section we presented results achieved with the parMERASA Mapping Tool as approach to extract parallelism at Runnable level of AUTOSAR tasks. The tool distributes the Runnables of a task to the available cores such that precedence constraints imposed by data dependencies among Runnables are respected. The observed results were of mixed nature. The highest speed-up observed with this method is 3.3, on the one hand, and the smallest speed-up is 1.0, on the other hand. In order to improve the speed-up we introduced the concept of supertask, which groups Runnables of tasks that are scheduled to the same point in time. On a 2 core processor the efficiency is high for nearly every supertask combination. Generally, the Mapping Tool is well suited for the scheduling of a single task or supertasks on a small number of cores (≤ 4).

5.3 Inter-Task parallelism

In this section, we describe the parMERASA approach to exploit thread-level parallelism of automotive applications.

A fundamental requirement for the migration of a legacy application from a SCE to a multi-core ECU (MCE) is the *preservation of the data flow*, i.e. the order in which tasks communicate. The reason is, this eases the validation and testing on the new platform, as a similar functional behaviour is expected. Unfortunately, the static Runnable-to-task mapping of an AUTOSAR application imposes data dependencies at task level. Hence, the achievable degree of thread-level parallelism is limited by the communication between tasks. Deliverable D2.4 introduced the classification of such dependencies as weak or strong. Using this method, we have created groups of tasks with strong dependencies and fixed time-relation. In this section, we introduce and evaluate the concept of *timed implicit communication (TIC)*. We will propose a conversion that allows to classify all dependencies as weak to generate a maximum nominal speed up) for extraction of inter-task parallelism. Explicit communication is transformed into implicit communication. Produced data elements from tasks are stored in a buffer and a publication timestamp is attached. To that end, we propose that the tasks do not publish produced data until the end of the current task period. The motivation for using task periods is because this characteristic of the application is the same for all target platforms. Reception of data elements is defined by the scheduling of task invocations on the SCE. The publication timestamps allows the MCE consumer task to read from the same invocation of the producer task as on the SCE. This is independently of the scheduling on the MCE. This guarantees that tasks read from the same task instance on the SCE and MCE task scheduling. Hence, the data flow remains the same in both ECU scheduling.

The approach is compliant with AUTOSAR, i.e. integration in a standard conform application is possible. TIC is integrated into (tiny automotive RTE). Decoupling producer and consumer task as well as using a publication timestamp makes the MCE task scheduling independent from the data flow, while full advantage is taken from parallel execution. Finally, the Runnable-to-task mapping of the application can remain unchanged, which guarantees a correct data flow inside a task.

5.3.1 Determining the data flow

As prerequisite the SCE configuration of the application must be known. For simplicity, we assume SCE to be non-preemptive. The migration of an AUTOSAR legacy application is performed in two steps. In a first step, the SCE configuration is transformed by changing the communication mode from explicit to implicit. The reason is implicit communication allows producer and consumer to be decoupled in order to execute in parallel. However, this approach does not guarantee that the same

data flow is maintained. For this reason, in a second step we propose to attach a *publication timestamp* to produced data elements to identify input data from its *history*. This approach requires storing produced data elements in a buffer data structure, which is provided by the tiny automotive RTE. Filling the buffer is done by invoking every task once at the application start.

After the application has been scheduled on the MCE (for example with the parMERASA Mapping Tool) the data flow between task invocations is determined based on the scheduling on the SCE. Determining the communication is done by defining a date for reading from the history of a variable, i.e. the corresponding publication timestamp. For every consuming task τ_c^n a set of read dates $A_c^n = \{t_p \mid \forall p \in \text{prod}(c)\}$ is defined. $\text{prod}(c)$ is a set of tasks that produce input data for c . At first, a task τ_c reads a default input value from task τ_p until $t = \max(c, p)$. Afterwards, the SCE scheduling is used to identify the date for input data, see Figure 25.

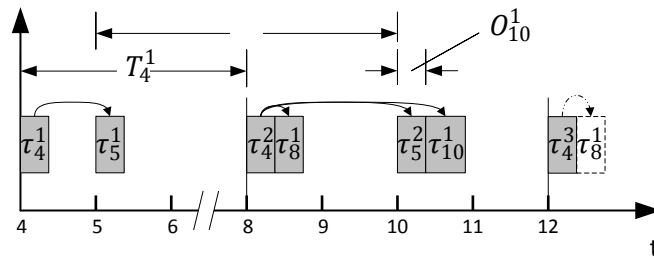


Figure 25: The scheduling and data flow on a SCE. Arrows represent data flow from producer to consumer task. Dashed lines indicate an alternative schedule and data flow for τ_8^1 .

The scheduling is transformed by shifting the reception of data by one producer period τ_p^n to τ_p^{n-1} . Given the parameter t_{SC} as time when the consumer task was scheduled on the SCE, the read date for data (t_p) from task p is calculated with:

$$t_p = \left\lfloor \frac{t_{SC}}{T_p} \right\rfloor \cdot T_p$$

Where T_p is the period of producer task p . The floor function guarantees that tasks always consume input data at the beginning of a new period. In order to identify the producing instance on the SCE t_{SC} is determined in a subsequent step. Given the schedule time of the consumer task on the MCE as t_{MC} the value of t_{SC} can be determined with:

$$t_{SC} = \left\lfloor \frac{t_{MC}}{T_c} \right\rfloor \cdot T_c + O_c^n$$

O_c^n represents the offset from the beginning of the n -th period in the event the receiver task has not been scheduled at the beginning of its period, see Figure 25. Task τ_4 will publish its data at times 4ms, 8ms, 12ms, etc. The data element produced by τ_4^1 must be consumed by τ_8^1 , τ_5^2 , and τ_{10}^1 as defined in the SCE scheduling shown in Figure 25. This is described by the arrows in the Figure. Hence, the date of the publication timestamp is in this case $t_p = 8$ for the value produced at $t = 4$ for τ_4^1 . Afterwards, the receiver τ_5^2 and τ_{10}^1 consume this value at $t = 10$ and τ_8^1 consumed the value at $t = 12$, respectively. For example, the read date for τ_8^1 scheduled at $t = 12$ on the MCE can be determined as:

$$t_{sc} = \left\lfloor \frac{12}{T_8} \right\rfloor \cdot T_8 + O_8^1 = \left\lfloor \frac{12}{8} \right\rfloor \cdot 8 + 0 = 8$$

$$t_4 = \left\lfloor \frac{8}{T_4} \right\rfloor \cdot T_4 = \left\lfloor \frac{8}{4} \right\rfloor \cdot 4 = 8$$

That means, τ_8^1 will read data from τ_4 with publication timestamp $t_4 = 8$, which was produced by τ_4^1 executed at $t = 4$, instead of reading data from τ_4^2 which is the task instance that has been executed immediately before τ_8^1 at cycle 8.

Note that if another scheduler places τ_8^1 at a later point in time, e.g. at $t = 14$ which could consume from task instance τ_4^3 , still the value of τ_4 published at $t = 8$ is consumed. This guarantees that tasks read from the same task instance on the MCE, if they did this also on the SCE. Consequently, the data flow is the same as on the SCE for explicit and implicit communication. Moreover, tasks can execute in parallel.

The buffer has been implemented by the project partner UAU. It resides in the tiny automotive RTE. See Deliverable 4.10 for details.

We evaluate the approach in two steps. First, the performance is analysed quantitatively with a diesel EMS as example based on OTAWA WCET estimates. Second, a prototypical implementation for the parMERASA processor on top of the tiny automotive RTE is investigated.

5.3.2 Quantitative analysis

In this section the performance of TIC is investigated. The WCET of a task is determined by static analysis with OTAWA.

5.3.2.1 Implementation

In the EMS 756 times Sender-Receiver (SR) communication, 77 times Inter-Runnable Variable (IRV) communication, and 33 *Server-Runnables* provide services to client Runnables. Client-Server (CS) communication is performed through a function call. Server-Runnables typically maintain an internal state, which is changed by an invocation from a client Runnable. Thus, memory coherency must be guaranteed. This is out of the scope of our approach and in order to perform performance evaluation we propose to enclose calls to a server in *ticket-locks* [4]. This blocks other tasks (Runnables) until the execution of the Server-Runnable has finished. These locks can be integrated within the RTE and are transparent to the client and the server. Thus, no changes of the application are required and data consistency is guaranteed as well as data coherency. In the examined application 4 Server-Runnables must be enclosed in ticket-locks.

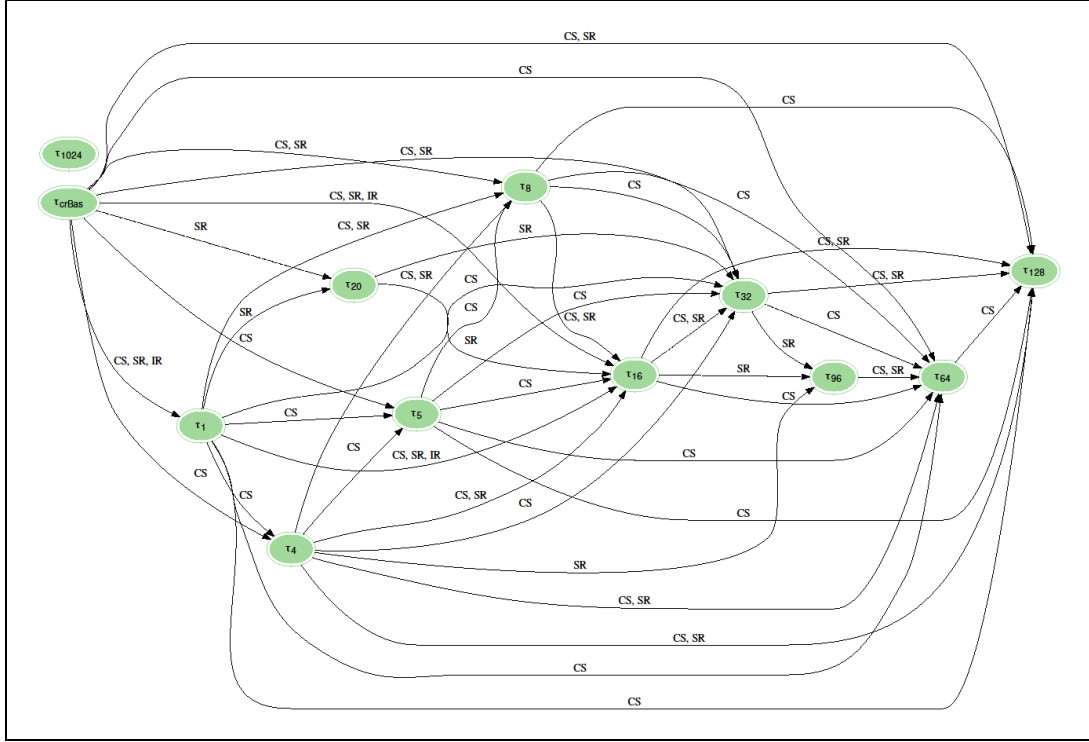


Figure 26: Graph representation of the data flow in a diesel EMS. Vertices represent tasks and edges represent data flow. Labels indicate the source of the flow: SR (Sender-Receiver), IR (Inter-Runnable Variable), CS (Client-Server).

5.3.2.2 Scheduling

Figure 26 illustrates the data dependencies in the examined EMS. Using a buffer with timestamps between tasks makes it possible to schedule tasks as if they were independent. That means, none of the edges in Figure 26 must be respected during scheduling. This is an advantage compared to other approaches as a higher degree of parallelism is achieved.

The Mapping Tool is used to create a static non-preemptive schedule for the worst-case scenario, which is the situation when all tasks have to be scheduled in the smallest period. As explained before, a special characteristic of an EMS are drifting timescales. Therefore, the crank-angle task is placed in a separate core to guarantee that interrupts can always be handled with low latency. For periodic tasks a static schedule is generated on the remaining cores is generated.

5.3.2.3 Setup and Metrics

The parameters for the experimental setup are listed in Table 13. We consider a 2-, 4-, and 8-core parMERASA processor with tree NoC. Each core has a private instruction scratchpad and a data cache.

We investigate the *performance* and the *overhead* of the approach. Attaching a timestamp (in a write operation) requires additional execution time as well as searching for a data element with a specified timestamp (in a read operation). The overheads for an application with task set T is calculated as the sum of the extra cycles for buffer write operations o_W , for buffer read operations o_R , and for ticket-locks around critical sections o_L :

$$o_T = \sum_{\tau \in T} o_R(\tau) + o_W(\tau) + o_L(\tau)$$

Extra cycles for buffer operations are calculated as $o_R(\tau) = |\tau|_{read} \cdot o_B$ and $o_W(\tau) = |\tau|_{write} \cdot o_B$, receptively. Here $|\tau|_{read}$ is the number of read operations in τ and o_B is the worst-case overhead for a single buffer operation. Because o_B is implementation specific, we consider different values in the evaluation.

Parameter	2-cores	4-core	8-core
Data cache size	256 KB		
Cache latency	1 cycle		
On-chip Memory latency	11 cycles	32 cycles	73 cycles
WCET_{lock}	211 cycles	337 cycles	538 cycles
o_B	0, 50, 100, 150, 200, 250, 300, 350 cycles		

Table 13: The experimental setup for the evaluation of inter-task parallelism with buffered communication

Server-Runnables R are critical section:

$$o_L(\tau) = \sum_{r \in R} \sum_{\tau' \in T_r \setminus \tau} |\tau'|_r \cdot (WCET_r + WCET_{lock}),$$

where T_r is the set of concurrent executing tasks (containing R), $|\tau'|_r$ is the number of calls from τ' to r , and $WCET_{lock}$ is the platform specific WCET to acquire and release one ticket-lock.

For a setup with two cores no scheduling is required to calculate the speed-up, because the first core exclusively executes the crank-angle task and the second core executes the periodic tasks in sequential order. Thus, the speed-up is calculated as:

$$s = \frac{WCET_{sequential}}{\max(WCET_{crBas}, \sum_{i \in T \setminus \tau_{crBas}} WCET_i)}$$

For setups with more than two cores the speed-up is calculated as:

$$s = \frac{WCET_{seq}}{\max(WCET_{crBas}, makespan(T \setminus \tau_{crBas}, p - 1))}$$

Here, $makespan(T, c)$ is the length of the schedule for the task set T on c cores, which is defined by the scheduling with the parMERASA Mapping Tool.

5.3.2.4 Overhead

Figure 27 shows the ratio of overhead to the total application WCET. For all setups we can observe the overhead increases with o_B . On the dual-core processor the overhead is with an optimal buffer very high (18.5%). The values increase up to 69.1% as o_B increases to 350 cycles.

The overhead on a 4-core processor (9.3%) is smaller compared to the 2-core processor for $o_B = 0$ cycles. The reason is, more cores cause more interference, which results in a higher WCET and thus the buffer overhead has a smaller impact on the overall execution time. Nevertheless, the values increase up to 58.5% for $o_B = 350$ cycles.

On the 8-core processor we can observe the same behaviour. For $o_B = 0$ cycles the overhead is with 0.4% negligible small, but it increases to 43.9% for $o_B=350$ cycles.

In summary, the ratio of overhead increases with o_B . As the number of cores increases the overhead decreases, because the interferences in the NoC influence the WCET more significantly.

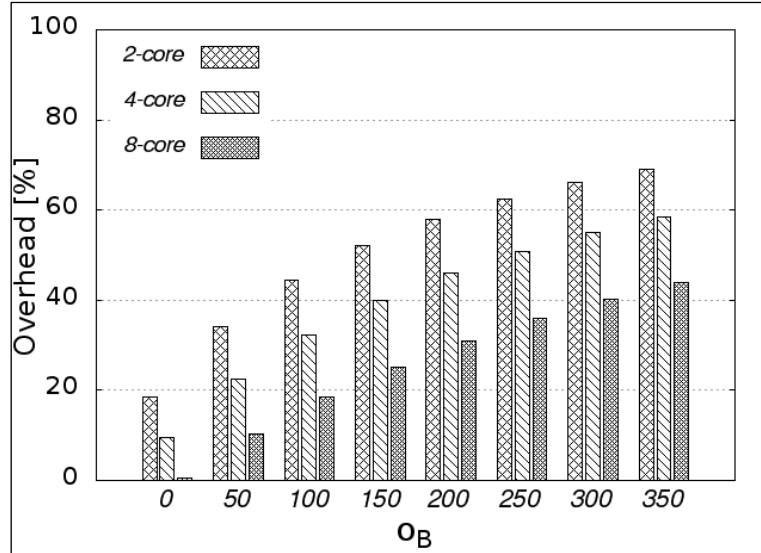


Figure 27: The ratio of overhead to the total application WCET depending on o_B .

5.3.2.5 Performance

Figure 28 shows the speed-up for the worst-case scenario of the EMS on different MCEs. The 8-core processor provides the highest speed-up, which is 4.5 for $o_B = 0$ cycles. The values for the speed-up decrease down to 2.1 as o_B increases to 350 cycles.

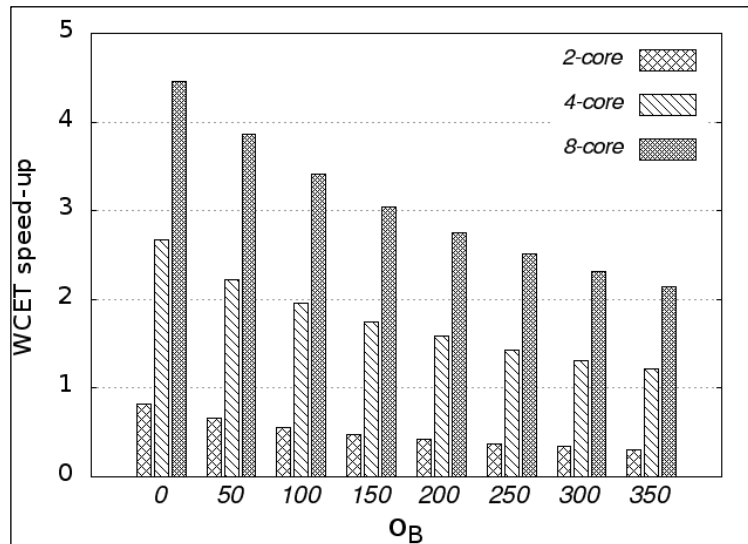


Figure 28: The results for speed-up.

On a 4-core MCE a maximum speed-up of 2.7 is achieved for $o_B = 0$ cycles. The speed-up decreases down to 1.2 as o_B increases to 350 cycles. The speed-up on the 2-core MCE is lower than 1 for all values of o_B . The reason is, the first core only executes the crank-angle task and all other tasks have to execute in sequential order on the second core. Moreover, the buffer operations and locks introduce additional overhead, which exceed the benefit of having two cores.

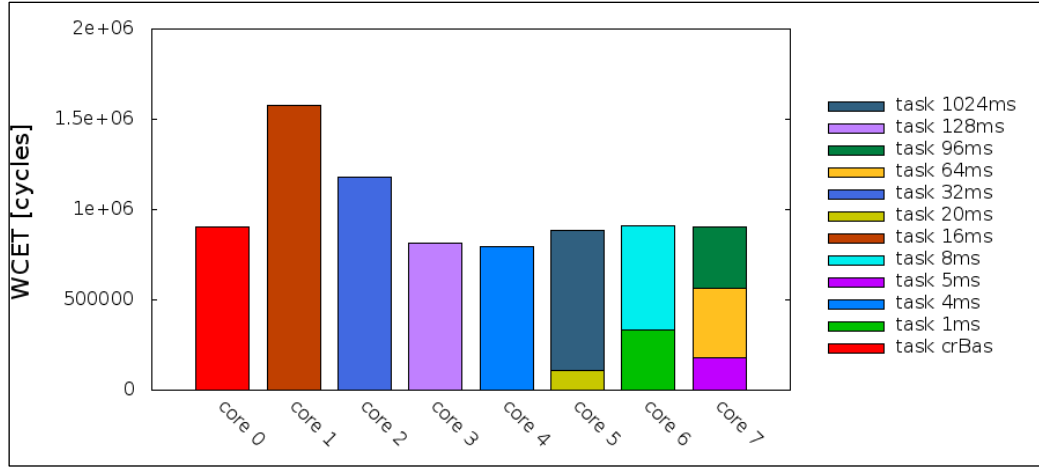


Figure 29: The schedule on the 8-core processor shows τ_{16} has the highest WCET of all periodic tasks.

The highest efficiency is achieved with the 4-core MCE (0.675). On the 8-core processor the efficiency is 0.56. Looking at the scheduling on this platform shown in Figure 29 reveals why the speed-up could not be increased above 4.5. The task τ_{16} has the highest WCET of all periodic tasks and is therefore scheduled to core 1. Consequently, the speed-up of the EMS is limited by the WCET of this task. For this reason, the 4-core MCE is the preferable target platform for the parallelized application, because the efficiency is higher. Nevertheless, improvements of the 8-core implementation could make this also an attractive target platform. We discuss in Section 5.4 a combined approach in which the Runnables of τ_{16} are distributed over 2 cores using the Mapping Tool to reduce the WCET and the speed-up is increased.

Summarizing, the highest speed-up could be achieved with the 8-core MCE (4.5). Although, the limiting factor on this platform is τ_{16} . Using multiple cores for the execution of the crank-angle task is a possible improvement.

5.3.3 Prototype Evaluation

For demonstration purpose a prototype has been implemented and simulated with the parMERASA simulator. The execution times have been measured with RapiTime PARMERASA, which is described in Deliverable 3.12.

The EMS is executed on the parMERASA software stack, i.e., an execution platform comprising the parMERASA simulator and the tiny automotive RTE. The latter one represents the domain specific runtime environment (see Deliverable D4.4) for the automotive domain. A buffer data structure has been integrated in the tiny automotive RTE provided by UAU. It satisfied the following properties:

1. The read and write access is never blocked (wait-free).
2. The write operation $write(x, v, t)$ stores the value v for the shared variable x in the database with the publication time t_p .
3. The read operation $read(x, t)$ returns the value of the variable x at time t .
4. Multiple producers can write to the same shared variable at the same time without blocking each other.

The prototype configuration uses a single cluster with 3 cores. The crank-angle task (τ_{crBas}) is scheduled on core 0. The execution of τ_{crBas} is triggered by a trace file. On core 1 a 2ms³ task is executed and on core 2 the 8ms task is executed. The periodic tasks are executed in parallel, synchronous to the ticks of a global clock with a period of 1ms.

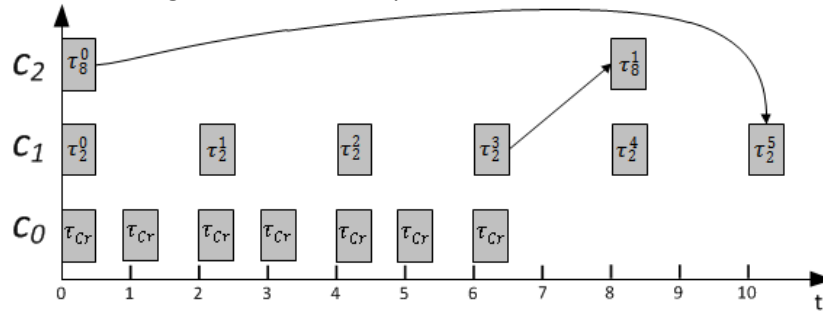


Figure 30: Prototype implementation of 3 tasks using TIC. Arrows represent the data flow between the task invocations.

Figure 30 shows the scheduling of the prototype implementation. At the start τ_2 , τ_8 , and τ_{crBas} are executed simultaneously to fill the buffer with a single value. They execute always at the beginning of a new period afterwards, that means τ_2 executes at $t = 2, 4, 6, 8, 10$ etc. τ_8 receives data from the previous instance compared to the single-core configuration. This is τ_2^3 instead of τ_2^4 in this example.

5.3.3.1 Implementation

The executable prototype is described in Prototype Deliverable 4.10. The code of the EMS must be adapted to allow access to the buffer. In order to be compliant with AUTOSAR, API-calls are defined in accordance with the *Rte_IRead* and *Rte_IWrite* function macros defined in the AUTOSAR RTE [5]:

- a) `Rte_IWrite_<re>_<dt>_<var>(value, publication_time)`
- b) `Rte_IRead_<re>_<dt>_<var>(time)`

Here, *<re>* is the name of the Runnable, *<dt>* is the data type of that variable, *<var>* is the variable name. The call is expanded to read/write task that will be further used for communication.

Figure 31 shows the implementation of the 2ms task on core 1. In the initialisation phase all parameters are set and the *Rte_Buffer_Init()* function allocates memory for the variables on this core. The makro *TASK_1MS_WRITE* defines the publication timestamp, which is used in *Rte_IWrite*-calls. The value is updated every period. Inside *Runnable1* the publication timestamp is used to write to variable the global variable *VAR*. Moreover, a read date is defined. In this case the only producer task is τ_8 . The variable *TASK_1MS_READ_FROM_TASK_8MS* contains the read date of the current period and is updated accordingly.

³ This is the former 1ms task. Only a subset of Runnables was implemented to illustrate the applicability. As a result, execution takes place every 2 milliseconds.

```

void Task_Core_1() {
    // Init section
    U1 r_esr_2m = (U1) 1;
    uint32_t b = 0;
    uint32_t t = (uint32_t) ADV_ABSOLUTE_START_APPLICATION;
    uint16_t ms = 1;
    Rte_Buffer_Init();
    TASK_1MS_WRITE = t;
    TASK_1MS_READ_FROM_TASK_8MS = (uint32_t) ADV_ABSOLUTE_START_APPLICATION;
    while (1) {
        TASK_1MS_WRITE += (uint32_t) ADV_CYCLES_ONE_MILLISECOND;
        advance(t);
        b = (uint32_t) get_time_base();
        r_esr_2m++;
        if (r_esr_2m >= (U1) 2) {
            r_esr_2m = (U1) 0;
            Runnable1();
        }
        t += (uint32_t) ADV_CYCLES_ONE_MILLISECOND;
        if (ms % 8 == 0)
            TASK_1MS_READ_FROM_TASK_8MS += 8 * (uint32_t) ADV_CYCLES_ONE_MILLISECOND;
        ++ms;
    }
}

void Runnable1() {
    float temp1, temp2;
    // implementation
    Rte_IWrite_Runnable1_float_VAR(f4LIB_Min(temp1, temp2), TASK_1MS_WRITE);
}

```

Figure 31: Example implementation of the task on core 1 that executes Runnable1.

5.3.3.2 Results

The prototype has been instrumented with RapiTime PARMERASA in order to measure the execution times, described in parMERASA Deliverable 3.12.

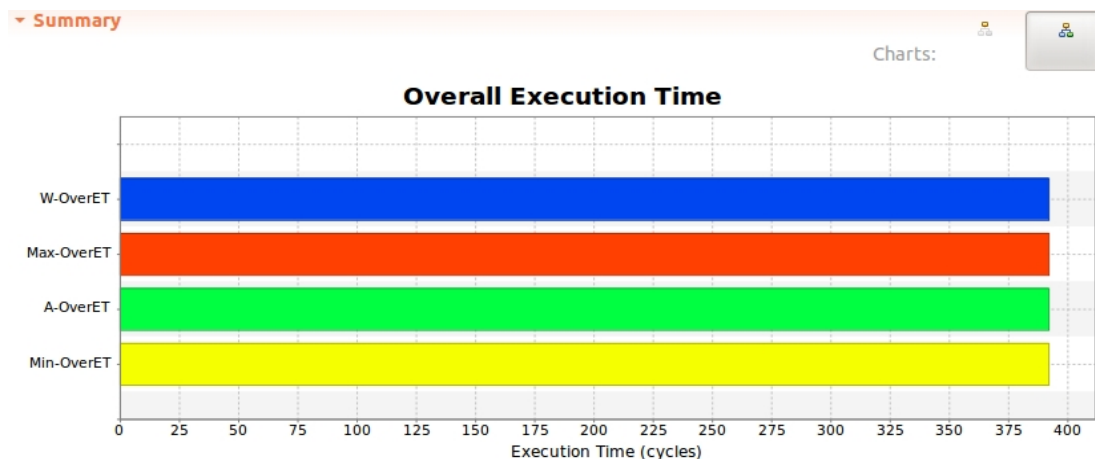


Figure 32: Screenshot from RapiTime parMERASA shows the WCET of a buffer write operations.

Figure 32 shows the observed execution time for the above mentioned Rte_IWrite-function. *W-OverET* is the Worst-Overall Execution Time, *Max-OverET* is the Maximum-Overall Execution Time, *A-OverET* is the Average-Overall Execution Time and *Min-OverET* is the Minimum-Overall Execution Time. These execution times are derived from the time spent between entry lpoint(the starting point from where the testing of code started until the point where it ended) of the called function referenced in the table row and any of its observed exit lpoints inside the called function. This block of code has been executed 3 times during testing and gives the *W-overET*, *Max-OverET*, *A-OverET*

and *Min-OverET* equals to 392 cycles. Similarly we performed the read operation through buffer and got the following results that are shown below in Figure 33.

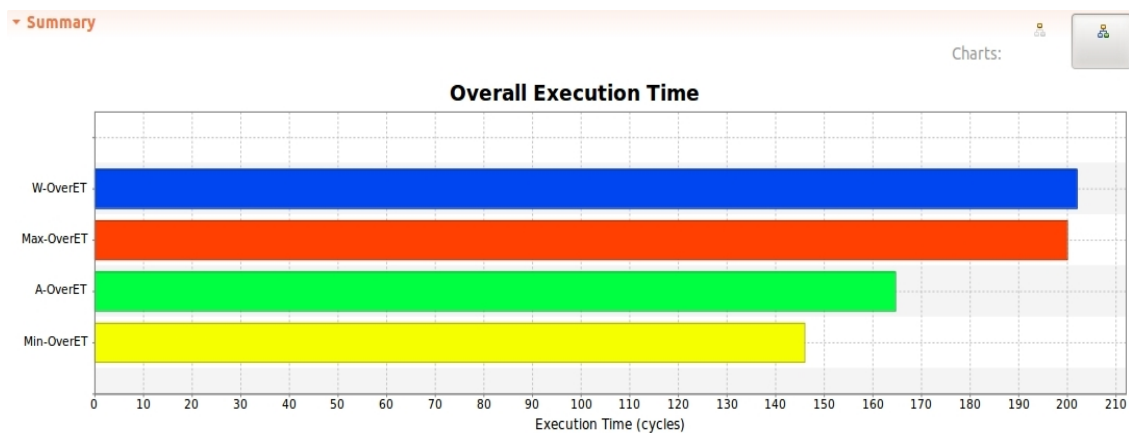


Figure 33: Screenshot from RapiTime parMERASA shows the WCET of a buffer read operations.

From the graphical picture of the read operation we observed W-overET is 202 cycles, Max-OverET is 200 cycles, A-OverET is 165 cycles, Min-OverET is 146 cycles. Total of 3 tests were performed.

5.3.4 Summary

In this section we presented timed implicit communication as approach to extract parallelism at task level of automotive applications. TIC uses a buffer and attaches a timestamp to data elements to decouple tasks. A prototype has been implemented to illustrate the applicability of the approach. For the examined EMS the approach has shown to provide a speed-up for the worst-case scenario of 4.5 on 8-cores. This approach is well suited for the scheduling of multiple tasks on a larger number of cores (>4). In order to improve the speed-up a combined approach is presented in Section 5.5.

5.4 Intra-Runnable Parallelism

Runnables in the EMS are typically very small. However, some Runnables feature a code size that is large enough to investigate its parallelization. Even if the outcome saves only a small amount of execution time, it can cause large benefits for the application in case of a frequently invocation of the Runnable on the critical path. Moreover, we envision in future that algorithms for online physical simulation or automatic parameter optimization might be incorporated in the automotive software. Runnables with such capabilities might have higher execution times compared to actually applied Runnables in automotive sector. Therefore, they limit the possibilities of Inter-Runnable parallelization. The obvious solution of further splitting the Runnable into smaller Runnables is not feasible for code readability. The common practice of assigning one algorithm to one Runnable should be preserved. As an alternative solution, the possibility to parallelize a single Runnable is investigated here.

Initially, the portability of the intended solution to real systems in the automotive domain is examined. To execute parallelized Runnables on automotive embedded systems support from the operation system (OS) is needed. Amongst other things, AUTOSAR specifies the requirements of operating systems in the automotive domain. This holds true for the tiny automotive RTE, which is a subset of AUTOSAR and applied for executing the application. The operating system of the tiny automotive RTE utilizes a time-triggered execution model which provides benefits for Intra-Runnable

parallelization. Mainly two aspects of the execution model lead to these benefits. They are listed below:

- The start of a thread execution can be forced to a fixed timestamp.
- The model is non-preemptive.

The ability of determining the exact timestamp for executing a particular code fragment can be applied for starting a parallelized Runnable at a fixed timestamp synchronously on several cores. Furthermore, due to the non-preemptive character, none of the Runnable's execution threads can be interrupted while the other threads remain executing. These features offer the possibility of executing a single Runnable distributed on multiple cores without interference to execution of other Runnables.

Several Runnables are analysed by UAU in cooperation with DNDE to demonstrate feasibility of Intra-Runnable parallelization. They are taken from the engine management system application by DNDE in order to ensure significance of the analysed code. Runnables with high load (meaning the number of invocations multiplied by the WCET) and Runnables with high WCET which stay in the critical path of a task schedule promise the greatest benefits from parallelizing. Therefore, these criteria are applied to select four particular Runnables. Two of them (subsequently named *R1* and *R2*) are selected by maximum load and the others (named *R3* and *R4*) have the largest WCETs of all Runnables in the critical path.

The appearance of code is nearly the same for all analysed Runnables. It mainly consists of conditional code and sequences of function calls. It is also characterized by nested function calls. The Runnables rarely include loops and the ones available possess only a small number of iterations to execute. Furthermore, the code is characterized by an almost complete absence of pointers. One special function (subsequently named *FN_func*) that is invoked repeatedly in three of the four observed Runnables should be highlighted: This function mainly consists of a sequence of function calls and represents large parts of the code of Runnables.

Section 5.4.1 shows the procedure of parallelizing a Runnable in detail. Section 5.4.2 envisions possible implementations for exposed parallelism and section 5.4.3 evaluates the parallelized Runnables and compares the different implementations.

5.4.1 Parallelization

In general for parallelization of single Runnables the *pattern-supported parallelization approach* introduced in Section 2.3 is applied. It enables the parallelization of sequential source code which is achieved by processing the phases *Reveal Parallelism* and *Optimize Parallelism*.

Here the parallelization is done by hand. By doing so, a fluent transition between both phases proved to be sensible. In concrete it means that observed parallelism (Phase 1) is examined relating to its feasibility (Phase 2) directly afterwards. The decision made in the second phase is an intuitive one. Though there is assistance for that decision (described in Section 5.4.1.1), there is no exact measure.

To determine parallel segments the *Parallel Design Patterns* introduced in parMERASA Deliverable D2.4 are applied. The PDPs describe best-practice solutions for parallelization of code. The PDPs provide us with a structure to the obtained parallelism and can be implemented with the synchronization idioms provided by the Kernel Library (cf. D4.2). Since these idioms are analysable for WCET, the patterns provide support for a close estimation of time bounds and the patterns'

structured parallelism facilitates the analyses of WCET. The actually provided patterns are listed below:

- Task Parallelism
- Periodic Task Parallelism
- Data Parallelism
- Pipelining

In Section 5.4.1.1, we reveal details about the code analyses leading to segments applicable for parallelization. Section 5.4.1.2 presents Parallel Design Patterns that are feasible for parallelizing examined Runnables.

5.4.1.1 Code Analysis

A static code analysis is needed to examine code segments feasible for parallelization. To do this, we applied the tool Understand [6] to gain support for analysing the code.

To expose data dependencies we explore all shared variables. Therefore, all accesses of each variable and their type (read/ write access) must be discovered. This leads to a list of variables and their accesses in code which is taken as basis for further analysis.

The next step is to analyse data dependencies in code by using the observed list of variables. For that, we examine the accesses of each variable in the sequential order of code. For every access a all subsequent accesses b of this variable are surveyed. Now the dependency between both accesses a and b can be determined.

The possible dependencies are presented below:

- a is read access:
 - b is read access: no dependency
 - b is write access: anti dependency
- a is write access:
 - b is read access: flow dependency
 - b is write access: output dependency

After determining data dependencies it is possible to identify independent segments in code. To do this the code is segmented and afterwards all defined code segments have to be examined. The segmentation of code follows three rules listed below:

- Define control structures (conditional blocks, loops) as own segments
- Define function calls as own segments
- Define consecutive instructions as own segments if they are dependent to each other

The segmentation has to be done recursively inside control structures and called functions. The rules listed here are not strict: In some situations applying the rules is not appropriate. As example for such cases let us assume a sequence of consecutive instructions. If there is one instruction in it which is independent to the other instructions there is no need to define multiple segments (as claimed in third rule). This is because it is inappropriate for parallelization to execute one single instruction as a self-contained thread. The overhead of parallelization would be much higher than the theoretical benefit.

After splitting code, all exposed segments have to be investigated respectively to their dependencies to other code segments. For this purpose, for each segment s all accesses to shared variables will be exposed. If there are nested segments within s , the accesses of these segments also need to be considered. The previously acquired list of variable accesses provides support to this issue. Next all subsequent segments t are analysed related to their variable accesses. For every t accesses of nested segments are considered, too. If there are accesses to the same variable in s and t , data dependencies of these have to be taken into account. If any dependencies exist, segments s and t are dependent, otherwise they are independent.

Now that we know independent segments in code, suitable segments for parallelization can be extracted. For that, if there are nested segments in code, one has to analyse each level by its own. During analysis a possible schedule of code segments for the currently observed level is determined by hand. The basis of this schedule is the exposed dependencies of code segments.

After scheduling, the parallel parts need to be examined respectively to their length in order to decide about their feasibility of parallelization. This is an intuitive decision. Small code segments are not practical for parallelization because the overhead for executing code in parallel is higher than the benefit of parallelization. If parallel execution shall be nested in a thread of an already existing parallelism as shown in Figure 34, it is to mention that this is suitable only if the contemplate segments (in Figure 34 named *Code_part_21* and *Code_part_22*) of code in sequential order have higher execution time than the other threads of the superordinate parallelism (in Figure 34 named *Code_part_1*). Segments where feasibility of parallelization is uncertain should stay in parallel. After implementing and testing (see sections 5.4.2 and 5.4.3) it can be decided if the uncertain parallel code segments are meaningful.

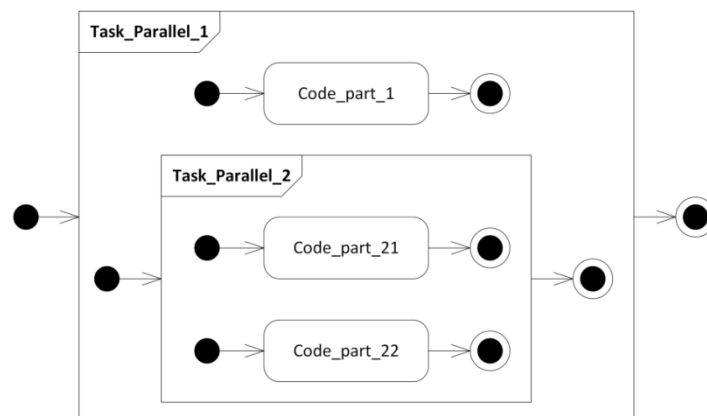


Figure 34: Activity and Pattern Diagram (APD) of nested parallelism; *Task_Parallel_2* is nested in *Task_Parallel_1*

5.4.1.2 Application of Parallel Design Pattern

While analysing code one has to search for structures that allow application of Parallel Design Patterns which are introduced in [1]. The investigation of the selected engine management system Runnables yields to a possible parallelization applying two different patterns: *Task Parallelism* and *Data Parallelism*.

However, a closer investigation shows that *Data Parallelism* is not applicable: This is because of the small number of iterations of the loops suitable for *Data Parallelism*. Even the loop with the highest potential for this pattern possesses only twenty iterations in maximum and there are only about 5 lines of code to execute per iteration.

The part of code with the highest potential for using Task Parallelism is the function *FN_func* characterized at beginning of Section 5.4. As already mentioned, *FN_func* is used in several Runnables. Therefore, parallelizing this function means parallelizing several Runnables. Also, *FN_func* covers large parts of the analysed Runnables. That is because the function is called multiple times within a single Runnable and it includes a huge amount of code to execute compared to the rest of the Runnables' code. The Activity and Pattern Diagram (APD) introduced in Deliverable 2.4 of *FN_func* is shown in Figure 35 and Figure 36.

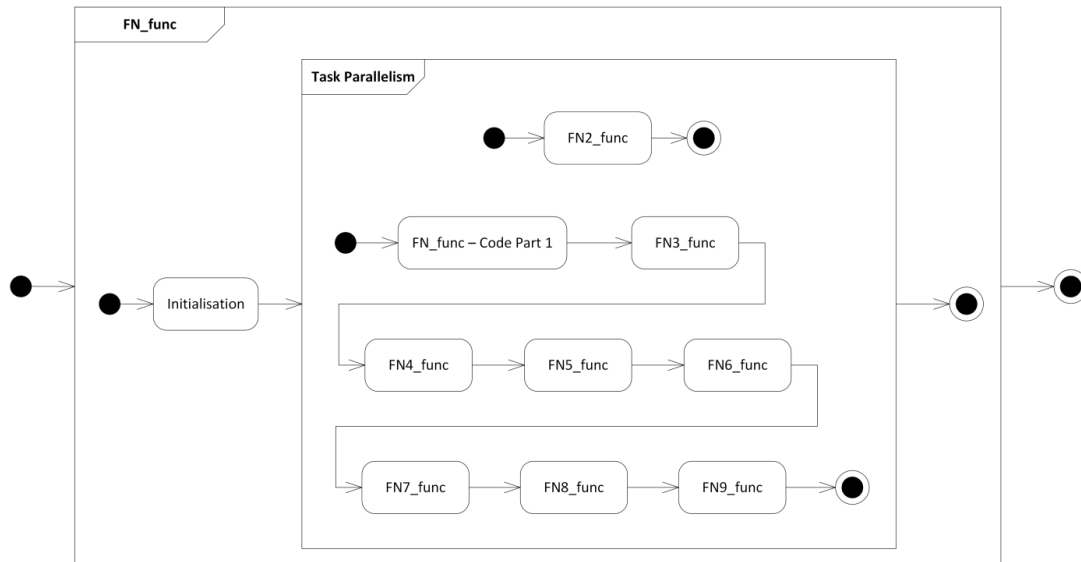


Figure 35: Activity and Pattern Diagram of function *FN_func*

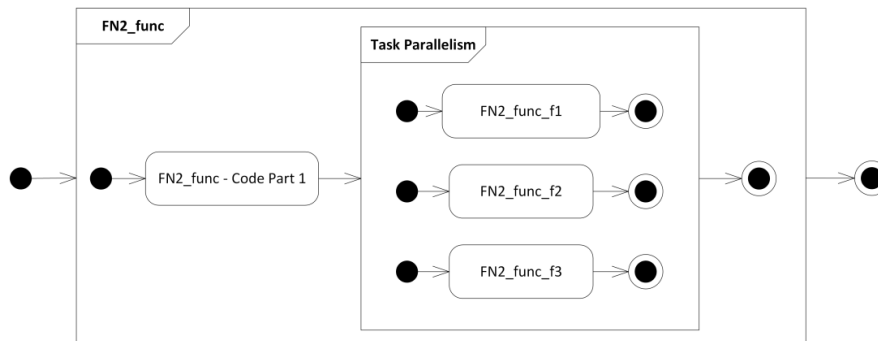


Figure 36: Activity and Pattern Diagram of function *FN2_func* (nested in *FN_func*)

As one can see there is an initialization phase within this function which has to be executed in sequential. The rest of the code can be executed as *Task Parallelism* which splits the code into two threads. One thread encapsulates the invocation of another function (subsequently named *FN2_func*) and the other executes the rest of the code in sequential. Further parallelization of the sequential part would be possible but there is no need to do so. This is because execution of the thread with the single function takes longer than executing the whole code of the other thread. In Figure 36 the APD of *FN2_func* is displayed. Applying *Task Parallelism* is possible for this function, too. The parallelization splits the execution into three threads.

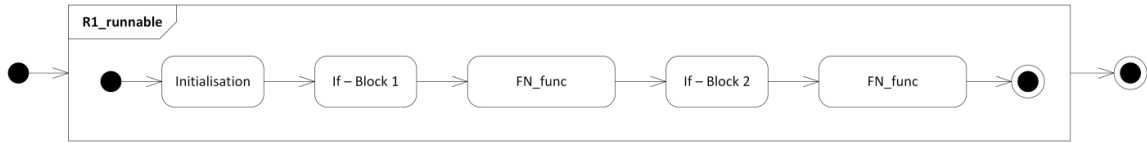


Figure 37: Activity and Pattern Diagram of Runnable R1

As example, the APD of Runnable *R1* is shown in Figure 37. This Runnable consists of code segments segmented during code analysis. It attracts attention that the function *FN_func* is invoked two times. A closer look to the code exposes that the other code segments of this Runnable are quite small. For that reason the Runnable code mainly consists of two calls of function *FN_func*. In another examined Runnable (namely *R3*) there is larger amount of code not belonging to *FN_func* but this function still covers considerable parts of the Runnable. Table 14 displays the observed execution time of sequentially executed Runnables and the execution times covered by *FN_func* to show the impact of this function to the Runnables.

Table 14: Observed execution times of Runnables for sequential execution and the impact of *FN_func* to the measured values (it is to mention, that each of *R1* and *R3* execute *FN_func* two times)

Measured Runnables/ functions	Impact of <i>FN_func</i>	
	Observed ET	Percentage of <i>FN_func</i>
FN_func	3758	100%
R1	7606	98.82%
R3	7860	95.62%

5.4.2 Implementation

We implemented two of the four Runnables (*R1* and *R3*) to evaluate the exposed parallelism. *R2* and *R4* are ignored because no parallelism is detected (in *R4*) respectively the parallelizable part (function *FN_func*) is never invoked (in *R2*).

For implementation we use two settings of parallelization. On one hand, the parallelization is done applying two threads and, on the other hand, a parallelization employing four threads is implemented. Each thread is mapped to a different core. When parallelizing with two threads only the *Task Parallelism* of Figure 35 is implemented in parallel. The code of function *FN2_func* is executed sequentially. Therefore, we need two cores for parallelization. When applying four threads, both *Task Parallelisms* of Figure 35 and Figure 36 are implemented. The parallelism of Figure 35 needs two cores and the nested parallelism of *FN2_func* (Figure 36) needs three cores. The third thread for *Task Parallelism* of *FN2_func* is already represented by one of the threads of Figure 35. Therefore, this complete implementation needs four cores. As mentioned in Section 5.4.1.2, further parallelization would be possible but cannot show positive effects. This is the reason why an implementation with more than 4 threads does not make sense.

The Runnables are implemented in two ways. Each way is implemented once with two threads and once with four threads for parallelization. Furthermore, a sequential implementation is provided as reference implementation. Listing 1 shows an example of sequential code. Subsequently, this code is used as example to show how the parallelization is done. The two parallel implementations apply on one hand Timing Analysable Algorithmic Skeleton (TAS) (introduced in [11]) and on the other hand an adapted implementation with the objective of reducing overhead.

```

// Variables

// Execution
void func() {
    // sequential part
    int a = 1;
    int b = 2;
    if (a<b) {
        b = a;
    }
    else {
        a = b;
    }

    // parallel part 1
    inti;
    for (i=0; i<5; i++) {
        a++;
    }

    // parallel part 2
    int j;
    for (j=0; j<5; j++) {
        b++;
    }
}

```

Listing 1: Example code for sequential execution

For implementing parallel code with *Timing Analysable Algorithmic Skeletons* (TAS), the segments representing one thread of parallel execution have to be encapsulated in a single function. An array is created including function pointers to the parallel functions of Task Parallelism. Afterwards variables which are written in a parallel executed function have to be mapped to shared variables. This does not lead to conflicts, because writing a variable implies that other simultaneously executed threads are not allowed to access this variable. For variables, which are only accessed for reading, it is possible to pass them as parameters to the parallel functions. To do so, one has to create a structure including all parameters of the parallel functions. Pointers to these structures are created and collected in an array. Finally, for execution of parallel segments the arrays with function pointers and pointers to the parameters have to be passed to the skeleton. The example of Listing 1 implemented with TAS is introduced in Listing 2. Within this listing, only shared variables are utilized for passing variables to parallel segments.

```

void par_part_1(void *args);
void par_part_2(void *args);

// Variables
SHARED_VARIABLE(membase_uncached0) volatile int a;
SHARED_VARIABLE(membase_uncached0) volatile int b;

SHARED_VARIABLE(membase_uncached0) volatile
void *_par_args[] = {NULL, NULL};
SHARED_VARIABLE(membase_uncached0) volatile
tas_runnable_tpar_code_parts[] = {
    (tas_runnable_t) par_part_1,
    (tas_runnable_t) par_part_2
}

```

```

};
SHARED_VARIABLE(membase_uncached0) volatile
tas_taskparallel_tpar_code = {
par_code_parts, par_args, 2
};

// Execution
void par_part_1(void *args) {
    // parallel part 1
    inti;
    for (i=0; i<5; i++) {
        a++;
    }
}

void par_part_2(void *args) {
    // parallel part 2
    intj;
    for (j=0; j<5; j++) {
        b++;
    }
}

void func() {
    // sequential part
    a = 1;
    b = 2;
    if (a<b) {
        b = a;
    }
    else {
        a = b;
    }

    // parallel execution (runs in main thread)
    tas_taskparallel_init(&par_code, 2);
    tas_taskparallel_execute(&par_code);
    tas_taskparallel_finalize(&par_code);
}

```

Listing 2: Example code for execution with Timing Analysable Algorithmic Skeletons

For this implementation it is possible to assign the workloads to cores during execution or before execution starts. When assigning code during execution it is possible to use the cores not needed at the moment for additional workload like other Runnables. For allocating cores before execution this is not possible. However, assigning cores at start causes less overhead than during execution.

The main reason for an adapted implementation of the *Task Parallelism* PDP is the objective to reduce parallelization overhead. For that, there is no possibility to assign parallel code during execution time as it is possible with TAS. The parallel parts of code are assigned to cores at design time. Therefore, it is necessary to specify which core is used to execute a specific part of code. Indicating that the code is executed by any additional core is not sufficient. For timing analysable behaviour the beginning and end of parallel segments are marked by barriers. In case of a conditional execution of the parallel segments, the according conditions have to be checked before reaching the barriers. Therefore, an additional barrier is needed before verification takes place to ensure consistence of the outcome across all cores. The barriers are implemented for threads that are involved in executing the parallel part of code. Local variables used in the single core version are

replaced by shared variables, if they have to be accessed from more than one core. Listing 3 shows how parallel code is implemented with this adapted implementation.

Though it is not possible to assign code to different cores during execution time it is possible to use idle cores for other workload. To do so, the additional code must be inserted in existing parallel code by hand at the correct position. Therefore, the WCETs of all parts of the code must be known. Otherwise too much workload could be assigned to one core. This would lead to an unbalanced workload in the parallel executed code section and would cause idle times for other cores waiting at the barriers.

```
//Variables
SHARED_VARIABLE(membase_uncached0) volatile int a;
SHARED_VARIABLE(membase_uncached0) volatile int b;

SHARED_VARIABLE(membase_uncached0) volatile
barrier_tsync_par_start = {.waiting = 0, .count = 2, };
SHARED_VARIABLE(membase_uncached0) volatile
barrier_tsync_par_end = {.waiting = 0, .count = 2, };

// Execution
void func() {
    intcpu = procnum();

    if (cpu==0) {
        // sequential part
        a = 1;
        b = 2;
        if (a<b) {
            b = a;
        }
        else {
            a = b;
        }
    }

    barrier_wait(&sync_par_start);      // ID=bar_start
    if (cpu==0) {
        // parallel part 1
        inti;
        for (i=0; i<5; i++) {
            a++;
        }
    }
    else if (cpu==1) {
        // parallel part 2
        int j;
        for (j=0; j<5; j++) {
            b++;
        }
    }
    barrier_wait(&sync_par_end); //ID=bar_end
}
```

Listing 3: Example code for an adapted parallel implementation

5.4.3 Evaluation

We calculate the WCETs and measure the observed execution times (OETs) using parMERASA Simulator for evaluation purposes. Therefore, the times are measured in the unit of processor cycles. Afterwards, results of different implementations are compared and interpreted. Subject of measurements are the OETs respectively WCETs of Runnables *R1*, *R3*, function *FN_func*, and function *APP_func*. *APP_func* does not belong to the EMS application. It is an additional implementation that reproduces the structure of *FN_func* (cf. Figure 35). The generation of this function is motivated by the objective to compare the effects of parallelized Runnables. The parallelized code of these Runnables is structured the same. They distinguish only in their execution times (ETs). *APP_func* exists in two versions. It is once implemented possessing a low execution time (subsequently named *APP_func_low*) and once featuring a high execution time (namely *APP_func_high*). The increased ETs are reached by adding additional workload equally to sequential and parallel executed code segments. The WCETs are calculated by RapiTime which provides a hybrid approach for analysing code (see Deliverable 3.12).

As described in Section 5.4.2 there are different implementations examined. The first implementation utilizes *TAS* by assigning the workloads before execution starts (subsequently named static allocated *TAS* or *sTAS*). The second one applies *TAS* by assigning the workloads during execution (subsequently named dynamic allocated *TAS* or *dTAS*) and the last one is the adapted implementation. The WCET of the adapted implementation is calculated by extracting the ETs of all sequential code segments and the ETs of the longest parallel executed paths and summing all up. It is to mention that RapiTime calculates parts of the execution for *TAS*s utilizing four cores as black boxes. Hence, the WCET results for these implementations are provided under reservation.

To get an overview of the analysed Runnables, the results are summarized in the following tables. Table 15 and Table 16 show the execution times of sequential execution. The according Runnables are executed on the same simulated processors as their parallelized versions. Thus, the tables show ETs for the two-core and the four-core processor. Table 17 and Table 18 display the execution times and speed-up values for OET and Table 19 and Table 20 contain the according values for WCET.

Table 15: OETs and WCETs of sequential implemented Runnables executed on 1 core utilizing a 2 respectively 4 core configuration

Sequential Execution	FN_func		R1_runnable		R3_runnable	
	2 Cores	4 Cores	2 Cores	4 Cores	2 Cores	4 Cores
OET	1294	1294	2678	2678	2932	2932
WCET	9714	11616	19586	23390	20346	24148

Table 16: OETs and WCETs of sequential implemented APP_func executed on 1 core utilizing a 2 respectively 4 core configuration

Sequential Execution	APP_func_low		APP_func_high	
	2 Cores	4 Cores	2 Cores	4 Cores
OET	1980	1980	7738	7738
WCET	8884	13384	35216	53216

Table 17: OETs and speed-up values of parallelized Runnables

OET (Speed-Up)	FN_func		R1_runnable		R3_runnable	
	2 Cores	4 Cores	2 Cores	4 Cores	2 Cores	4 Cores

sTAS	1825 (0.71)	1800 (0.72)	3412 (0.78)	3492 (0.77)	3692 (0.79)	3744 (0.79)
dTAS	2505 (0.52)	2902 (0.45)	5099 (0.53)	5610 (0.48)	5353 (0.55)	5860 (0.50)
Adapted	1389 (0.93)	1408 (0.92)	2690 (1.00)	2834 (0.94)	3070 (0.96)	3203 (0.92)

Table 18: OETs and speed-up values of parallelized *APP_func*

OET (Speed-Up)	APP_func_low		APP_func_high	
	2 Cores	4 Cores	2 Cores	4 Cores
sTAS	2516 (0.79)	1941 (1.02)	7966 (0.97)	3831 (2.02)
dTAS	3151 (0.63)	4734 (0.42)	8605 (0.90)	6584 (1.18)
Adapted	2137 (0.93)	1559 (1.27)	7585 (1.02)	3410 (2.27)

Table 19: WCETs and speed-up values of parallelized *Runnables* calculated by *RapiTime*

WCET RapiTime (Speed-Up)	FN_func		R1_runnable		R3_runnable	
	2 Cores	4 Cores	2 Cores	4 Cores	2 Cores	4 Cores
sTAS	8947 (1.09)	6806 (1.71)	18440 (1.06)	13730 (1.70)	19200 (1.06)	14488 (1.67)
dTAS	11476 (0.85)	11882 (0.98)	23506 (0.83)	23890 (0.98)	24028 (0.85)	24648 (0.98)
Adapted	9335 (1.04)	5039 (2.31)	18828 (1.04)	10236 (2.29)	19683 (1.03)	11019 (2.19)

Table 20: WCETs and speed-up values of parallelized *APP_func*

WCET RapiTime (Speed-Up)	APP_func_low		APP_func_high	
	2 Cores	4 Cores	2 Cores	4 Cores
sTAS	9338 (0.95)	7174 (1.87)	35092 (1.00)	22554 (2.36)
dTAS	11894 (0.75)	12494 (1.07)	37626 (0.94)	26976 (1.97)
Adapted	8767 (1.01)	7972 (1.68)	34477 (1.02)	25094 (2.12)

The analysis of Table 15 to Table 20 exposes multiple times higher WCETs than OETs. The parallel implementations discover comparable differences between OET and WCET. In addition, the measurement of OET reveals no significant performance benefits. Moreover, the values exhibit an increased OET for nearly all parallel implementations and some of the *Runnables'* OETs are even doubled. During investigating the reasons for this behaviour, it turned out that significant parts of *FN_func* are not executed when measuring the OETs. In concrete, the function *FN2_func* (see Figure 36) is not executed because of a conditional execution. This is an important reason for the big difference between the values of OET and WCET. For an improved investigation of parallelism, we ensured the execution of the conditional executed part by adapting one line of code. The according OETs are displayed in Table 21 and Table 22.

Table 21: OETs of sequential implementation including the execution of *FN2_func* *Runnables* executed on 1 core utilizing a 2 respectively 4 core configuration

Sequential Execution	FN_func		R1_runnable		R3_runnable	
	2 Cores	4 Cores	2 Cores	4 Cores	2 Cores	4 Cores
OET	3758	3758	7606	7606	7860	7860

Table 22: OETs and speed-up values of parallelized *Runnables* including the execution of *FN2_func*

OET (Speed-Up)	FN_func		R1_runnable		R3_runnable	
	2 Cores	4 Cores	2 Cores	4 Cores	2 Cores	4 Cores
sTAS	3411 (1.10)	2720 (1.38)	6182 (1.23)	4510 (1.69)	6816 (1.15)	5116 (1.54)
dTAS	3894 (0.97)	5177 (0.73)	7875 (0.97)	10258 (0.74)	8478 (0.93)	10867 (0.72)

Adapted	3131 (1.20)	2290 (1.64)	5610 (1.36)	3516 (2.16)	6338 (1.24)	4233 (1.27)
----------------	-------------	-------------	-------------	-------------	-------------	-------------

The new OETs of Table 21 and Table 22 exhibit about 2,5 times higher ETs than the ones in Table 15 and Table 17. However, there is still a large gap between the OETs and the according WCETs. As shown in Figure 38, the new measured speed-up values show a better performance than the values of execution without *FN2_func*. This observation can be explained with the large percentage of parallelized workload that is covered by *FN2_func*. Thus, we want to mention that the performance in the case of OETs is strongly dependent of the actually taken execution paths. Subsequently, when talking about OETs the ETs of the implementations with execution of *FN2_func* are meant.

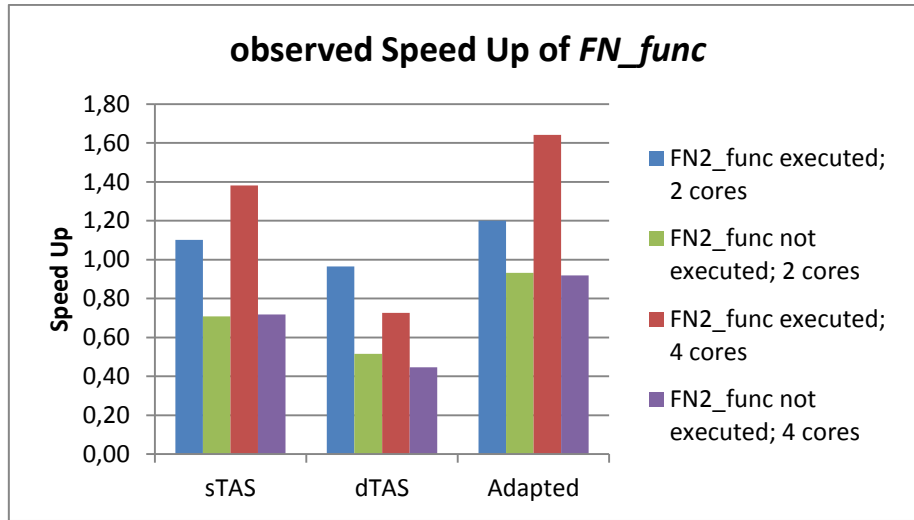


Figure 38: Speed-up values of OETs executing and not executing *FN2_func*; here, the implementations of *FN_func* is representatively displayed

There are few Runnables that show noticeable speed-up values for OETs. Mostly, a speed-up of lower than 1.5 is achieved. Overall, the OET speed-up values range from 0.72 to 2.16. The investigation of WCETs exhibits a similar performance for the most cases. However, there are also some differences between OETs and WCETs. This concerns especially the speed-up values of the versions utilizing four cores, which exhibit for WCETs higher speed-up values than for OETs. The lowest calculated WCET speed-up is 0.75 for *APP_func_low*. No Runnable provides a higher speed-up than *APP_func_high*, which reaches a value of 2.36. The values of Runnables range from 0.83 to 2.31.

A closer look at the results discovers a dependency of speed-up values regarding to the chosen parallelization technique. All implementations utilizing *sTAS* provide better speed-up values than the corresponding *dTAS* implementations. The maximum difference in speed-up between these two implementations is 0.26 for two cores and 0.95 for four cores. The minimum speed-up margin between *sTAS* and *dTAS* is 0.06 for two cores and 0.39 for four cores. When comparing the adapted implementation with the TAS implementations, the speed-up values show similar values as *sTAS*. Only the WCETs of execution with four cores provide noticeable speed-up margins for the Runnables. These Runnables provide margins up to 0.60. However, the speed-up values of the adapted implementation is slightly better than the ones of *sTAS* for most Runnables.

After investigating the speed-up values, it turned out that *dTAS* provides the poorest speed-up of all observed parallelization techniques. Since the margin of speed-up values between this implementation and the others is up to 1.42, one can say that this is a significant shortcoming of this

parallelization technique. Analyzing *sTAS* and the adapted implementation, we already mentioned that the adapted implementation shows better speed-up values than *sTAS*. However, with regard to the small margins between their speed-up values, it should be investigated carefully which implementation to apply. This is also relevant because of the easier implementation of *sTAS* compared to the adapted implementation, when parallel executed segments are located in conditionally executed code branches.

The different speed-up values observed for various implementations is mainly caused by the parallelization overhead. This can be seen especially at the comparison of *sTAS* and *dTAS*. Like already mentioned, the main difference of both variants is the time when assignment of workload takes place (before respectively during execution). Assigning it during execution (*dTAS*) causes additional workload. Thus, the parallelization overhead is increased, which leads to a poorer performance than for the parallelization without this overhead during execution (*sTAS*).

Due to the similarities in code construction the functions *APP_func_low* and *APP_func_high* are appropriate to investigate the effects of parallelization of Runnables. The Runnables are characterized by different execution times. It appears that *APP_func_high* gains a higher speed-up than *APP_func_low* for all investigated parallelization techniques. Furthermore, the better performance of *APP_func_high* compared to *APP_func_low* exhibit that the ET of the parallelization overhead does not grow as fast as the ET when increasing the workload of function *APP_func* (from *APP_func_low* to *APP_func_high*). In contrast, the overhead causes reduced speed-up values when the ET is decreased. At a certain point, the speed-up drops below 1 and parallelization causes no benefits anymore. This point can be seen as break-even-point at which parallelization becomes profitable and vice versa. It is not possible to identify an ET for a specific Runnable that acts as break-even-point for all applicable parallelization techniques. This can be exposed by comparing the speed-up values and ETs from *APP_func_low* or *APP_func_high* for diverse analysed parallel implementations.

When we observe the speed-up values for applying various numbers of cores, the results show a different behaviour for the implementation of *dTAS*, compared to the other implementations. The utilization of *sTAS* or adapted implementation reveals significantly higher speed-up values of the version with 4 cores than the one with 2 cores. For Runnables, this is due to the parallelization of *FN2_func*, which provides a large potential for parallelization. The implementation of *dTAS* obtains a loss of performance for the 4 cores version compared to the 2 cores version for OETs. This can be explained by the additional parallelization overhead. While there is no increase in overhead for *sTAS* and adapted implementation, the overhead of *dTAS* is nearly doubled. Thus, the overall workload is increased and leads to a performance loss shown as decreased speed-up values. The WCETs provide slightly higher speed-up values for 2 core version than for 4 core version. Thus, it seems that the consideration of the worst case reduces the effect of the additional overhead. The speed-up values of 2 core versions are up to 1.36 and ones of 4 core versions are up to 2.36. It is to mention that the maximum speed-up of 4 core version is reached by *APP_func_high*. The maximum for Runnables is 2.31.

For a closer evaluation of the performance, the efficiency values of all speed-up values are calculated and displayed in Table 23 and Table 24. The efficiency represents the speed-up value per core for a particular parallelization technique. These values reach from 0.18 (*dTAS* of *R1*) to 0.68 (adapted implementation of *R1*) for Runnables. The efficiencies of *APP_func* are located in this range, too. Only

one value of *APP_func_low* obtains a smaller efficiency. The effects on the efficiency due to different implementations and different numbers of applied cores can be seen in Figure 39. It provides representatively for the Runnables the efficiency values of *FN_func*. The efficiency of both implementations of TAS is worse for applying 4 cores compared to the application of 2 cores. In contrast the adapted implementation causes a higher efficiency here. When comparing the implementations, one can see that *sTAS* obtains higher values than the adapted implementation for two core versions and vice versa for 4 core versions. The worst efficiency for both versions provides *dTAS*.

Table 23: Efficiency values of Runnables

Efficiency WCET (OET)	FN_func		R1_runnable		R3_runnable	
	2 Cores	4 Cores	2 Cores	4 Cores	2 Cores	4 Cores
sTAS	0.54 (0.55)	0.43 (0.35)	0.53 (0.62)	0.43 (0.42)	0.53 (0.58)	0.42 (0.38)
dTAS	0.42 (0.48)	0.24 (0.18)	0.42 (0.48)	0.24 (0.19)	0.42 (0.46)	0.24 (0.18)
Adapted	0.52 (0.60)	0.58 (0.41)	0.52 (0.68)	0.57 (0.54)	0.52 (0.62)	0.55 (0.46)

Table 24: Efficiency values of APP_func

Efficiency WCET (OET)	APP_func_low		APP_func_high	
	2 Cores	4 Cores	2 Cores	4 Cores
sTAS	0.48 (0.39)	0.47 (0.26)	0.50 (0.49)	0.59 (0.51)
dTAS	0.37 (0.31)	0.27 (0.10)	0.47 (0.45)	0.49 (0.29)
Adapted	0.51 (0.46)	0.42 (0.32)	0.51 (0.51)	0.53 (0.57)

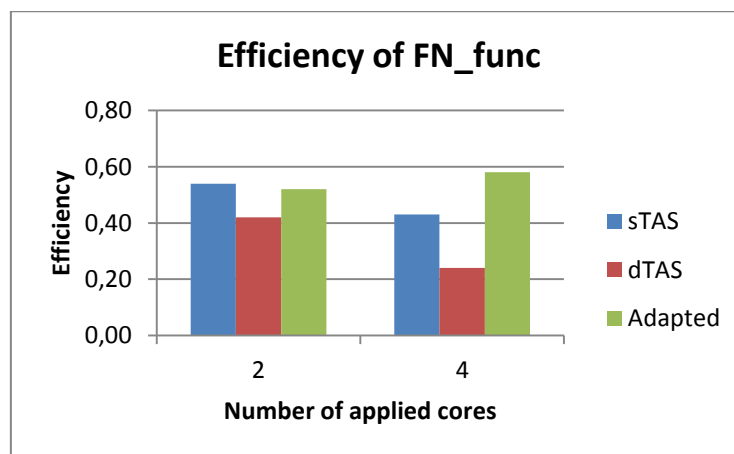


Figure 39: Efficiency values for different implementations and numbers of cores; here the implementations of *FN_func* is representatively displayed

5.4.4 Conclusion

In this section investigated the intra-Runnable parallelization. For this, a selection of the most promising Runnables and two additional functions were examined. We analysed the code of all Runnables to disclose the data dependencies and afterwards applied the Parallel Design Pattern to reveal parallelism. Though, we used the *pattern-supported parallelization approach* which provides two phases for parallelization, the parallelization was done in a fluent transition of both phases. Afterwards two of the Runnables were identified as suitable for parallelization and the Parallel Design Pattern were applied to them. Three different implementations were utilized, each in two versions; one applying two cores and one applying four cores. According to the different versions and implementations the parallelization obtained different speed-up values. The provided values were in

a range from 0.72 to 2.31 for Runnables and from 0.42 to 2.36 for the additional functions. The analysis of efficiency obtained a minimum of 0.18 for Runnables and 0.10 for the additional functions. The maximum is 0.62 and 0.57 for the additional functions.

5.5 Combined Approach

This section combines the previously discussed approaches to improve the speed-up of the worst-case scenario, which is the point when all tasks of the EMS have to be scheduled in the smallest period.

As stated in Section 5.3.2.5 the speed-up with 8 cores for the worst-case scenario is 4.5 times. The limiting factor is τ_{16} , because it has the highest WCET of periodic tasks and allocates on core completely. Therefore, the application is split in three parts, see Table 25. The first part is formed by the crank-angle task, which resides on core 0 to allow handling of interrupts with low delay. The task τ_{16} is distributed over the cores 1 and 2 using the parMERASA Mapping Tool in order to reduce the WCET of this limiting task. The evaluation in Section 5.2.3.2 has shown that the Mapping Tool is very well suited for the parallelization of Runnables of a task over two cores. The third group comprises all remaining tasks. They are scheduled over the five cores (3-7) using the Mapping Tool.

Group	Core	Tasks
1	0	τ_{crBas}
2	1 + 2	τ_{16}
3	3 – 7	$T = \{\tau_1, \tau_4, \tau_5, \tau_8, \tau_{20}, \tau_{32}, \tau_{64}, \tau_{96}, \tau_{128}, \tau_{1024}\}$

Table 25: Distribution of tasks over cores with the combined parallelization approach

5.5.1 Experimental results

Figure 40 shows the resulting allocated WCET of tasks on the 8-core parMERASA processor. The WCET of the τ_{16} could be reduced as desired from 1577084 cycles down to 962557 cycles. This results in a speed-up of 1.6 for τ_{16} . The red line indicates the WCET of τ_{16} before parallelization on two cores. Both values include overhead for locks around server-calls. The length of the schedule for the other periodic tasks on cores 3 to 5 is given with 1179217 (τ_{32} on core 3).

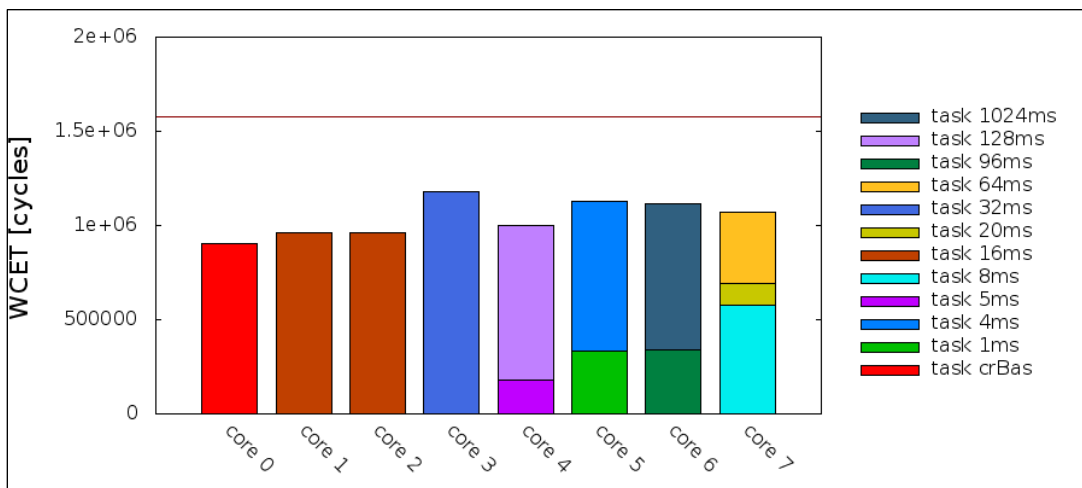


Figure 40: Resulting allocation of tasks to cores in the combined parallelization approach.

Consequently, the speed-up for this setup is calculated as:

$$s = \frac{WCET_{\text{sequential}}}{\max(\tau_{crBas}, \tau_{16} + o_{16}, T)} = \frac{7038533}{\max(900683, 962557, 1179217)} = 5.97$$

This results in an efficiency of 0.75 on the 8-core parMERASA processor.

5.5.2 Summary

This section presented combined approach in which inter-task and intra-task parallelism are exploited. All tasks use timed implicit communication as described in Section 5.3. The task limiting the parallelism was distributed on 2 cores using the parMERASA Mapping. The combined approach for the examined EMS application results in a WCET speed-up of 5.97 times.

6 CONSTRUCTION MACHINERY

6.1 Single core application on the parMERASA platform

The application prototype provided by BAUER in the parMERASA project is software running on electronic control unit of a foundation crane. It is worth to mention that BAUER application represents a complete control program and just part of it, which is good for parallelization. The software executes BAUER Dynamic Compaction (BDC) – for more details, please refer to parMERASA Deliverable D2.1, Section 5.2.1 - 5.2.3.

In order to have comparable data with the simulator, a single-core version was created. Its structure can be seen in Figure 41. In the original electronic control unit, the execution of the periodic tasks interrupts the execution of the main control loop several times (see Figure 43). In the parMERASA version for the simulator, the periodic tasks are executed after each main loop pass. Periodic tasks are independent of the extent of the main loop. As can be seen in Figure 42, the timing is not substantially affected. The time of the current execution of the main loop is measured. Then it is determined how often the periodic tasks have to be called. After the first pass, the time is determined by the duration of the main-loop and the last call of the periodic tasks. Therefore, the total time increases from the second pass and converges from run to run to a realistic execution time, as it would be achieved with a scheduler.

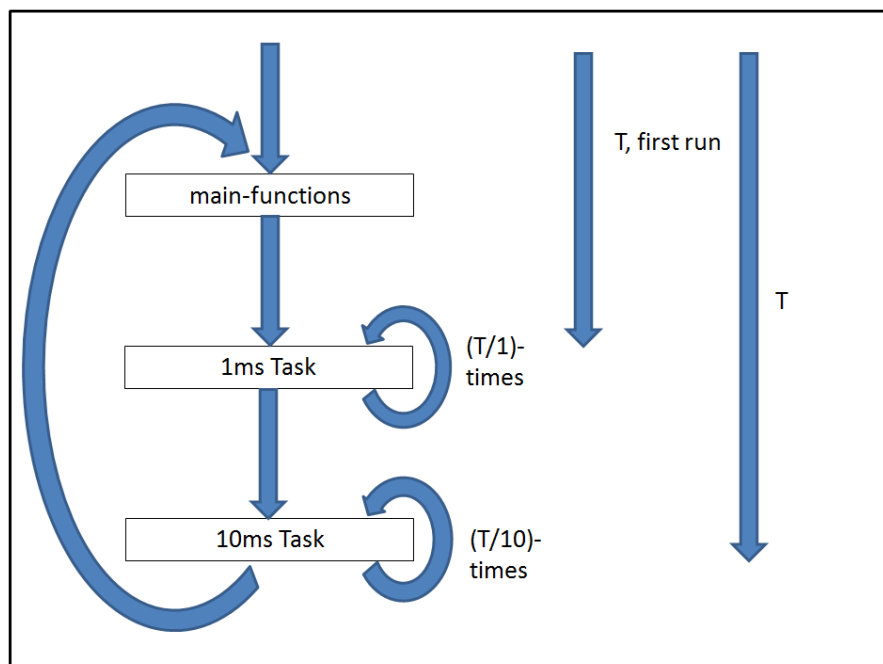


Figure 41 Structure of the single core version of the BAUER application on parMERASA simulator

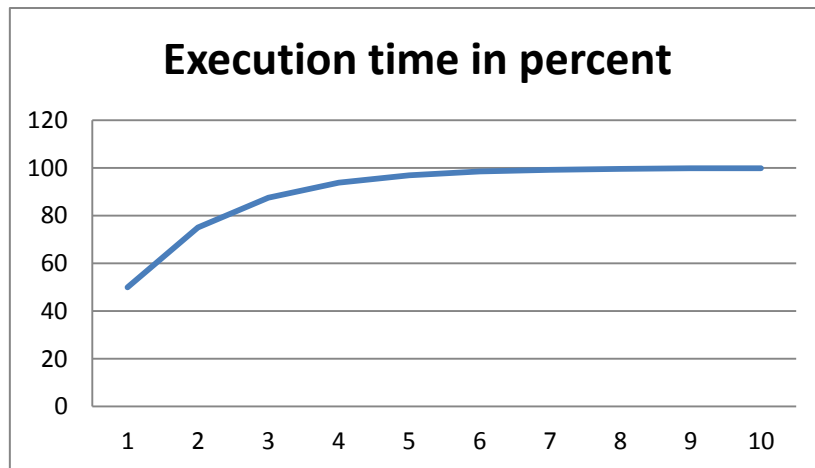


Figure 42 Execution time is near the real execution time after a few runs

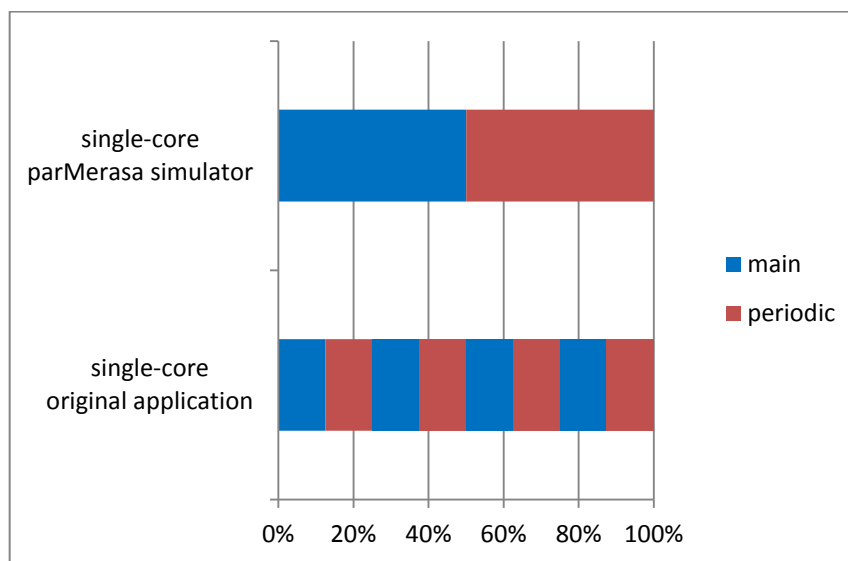


Figure 43 Distribution of the execution time of the main loop and the periodic tasks in the original application and in the parMERASA simulator

6.2 Parallel applications using timing analysable algorithmic skeletons

For the application parallelization, we apply the parMERASA parallelization approach from UAU. In the application analysis phase several possibilities for parallelism have been identified: On 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 Parallel Design Patterns and the Timing-analysable Algorithmic Skeletons (TAS) developed by UAU. It contains functions for controlling different parts of the machine, e.g., for rotating or moving the foundation crane or the subprogram running BDC.

Results of the program analysis are numerous activity and pattern diagrams. In their graphical representation, they do not contain shared resources (e.g., global variables); otherwise, they would be too complex. The important activity and pattern diagram *vBetrieb* is presented below.

Figure 44 shows a detailed illustration of *vBetrieb*, which is an instance of the Task Parallelism Parallel Design Pattern. 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

contrast the four activities vLogo_allg to vBohrverfahren contain many possibilities to exploit parallelism.

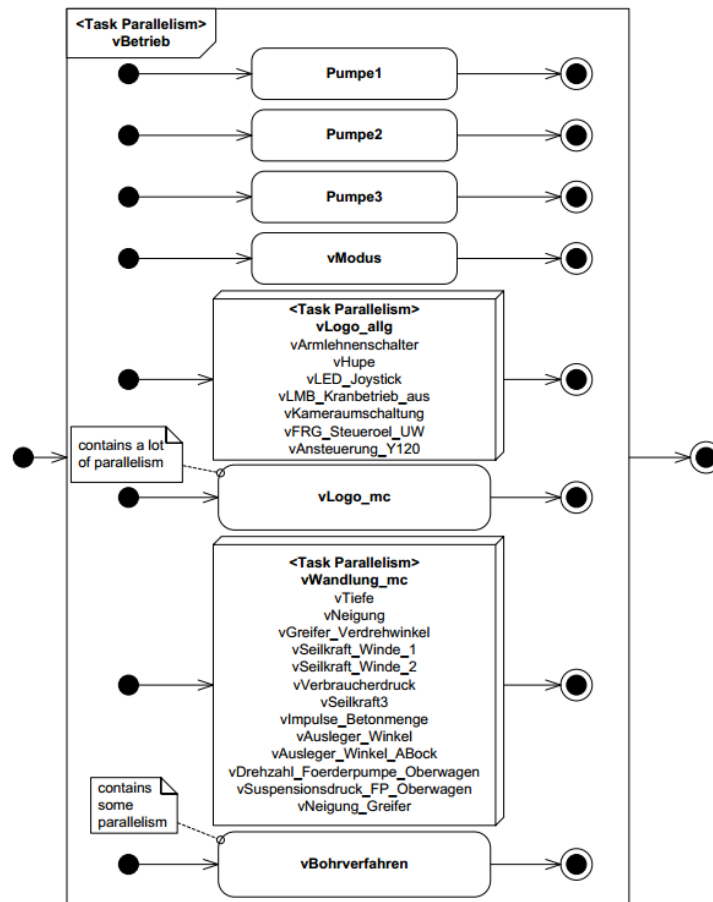


Figure 44 APD of vBetrieb: Eight activities, whereof the half parallelizes as well, execute in parallel

One of the further results of the detailed analysis (see also parMERASA Deliverable 2.4) was, that the periodic tasks should run on dedicated cores. In the original system, they were operated by a scheduler that interrupted the main control loop. For the parallel version, the idea was to give them one dedicated core. But, since the clock frequency of the parMERASA simulator is lower than in the original system (100 MHz vs. 150 MHz), the periodic tasks should run on two cores. This means that there are always two cores for periodic tasks and a variable number of cores for the main-loop depending on configuration.

The University of Augsburg calculated how the main loop can optimally be distributed over the cores. For this purpose the WCET of individual sections of the code and the synchronization overhead were taken into account. In Figure 45, every 'X' is one calculated execution time for one configuration. For a fixed number of cores, there are thus configurations with different execution times. It is assumed that one thread is always mapped to one core.

In parMERASA, we implemented the pareto-optimal configurations, i.e. those where we could expect a high speedup.

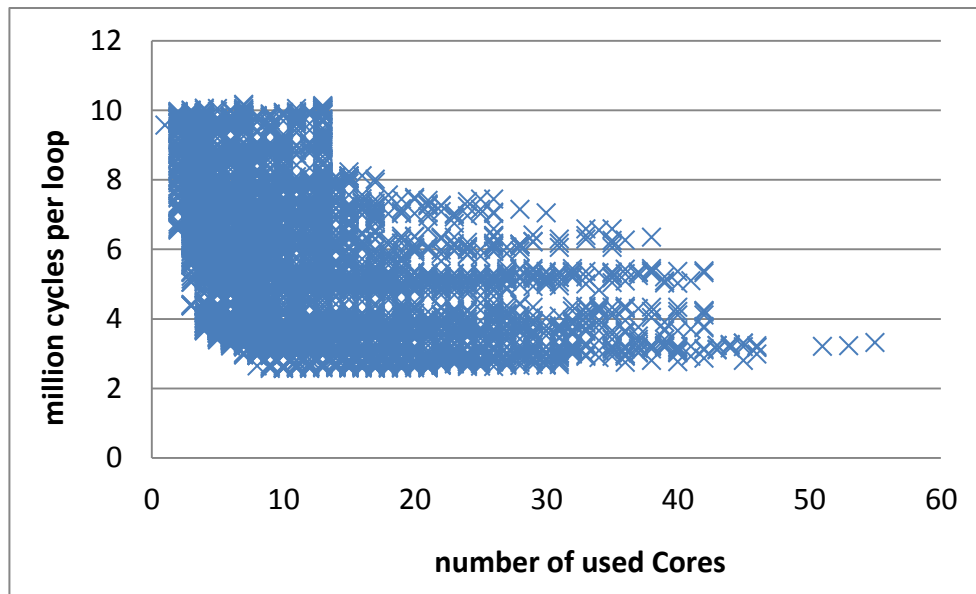


Figure 45 Mapping of the main-loop. Every "X" is a possible mapping

The parallel execution of the main loop is done using the Timing-analysable Algorithmic Skeletons (TAS). They are employed in the main loop several times, even in a nested way. This means that one core that runs as a worker, can also run a new instance of TAS and gets a worker for this instance. In Figure 46, the distribution of the 8-core version can be seen.

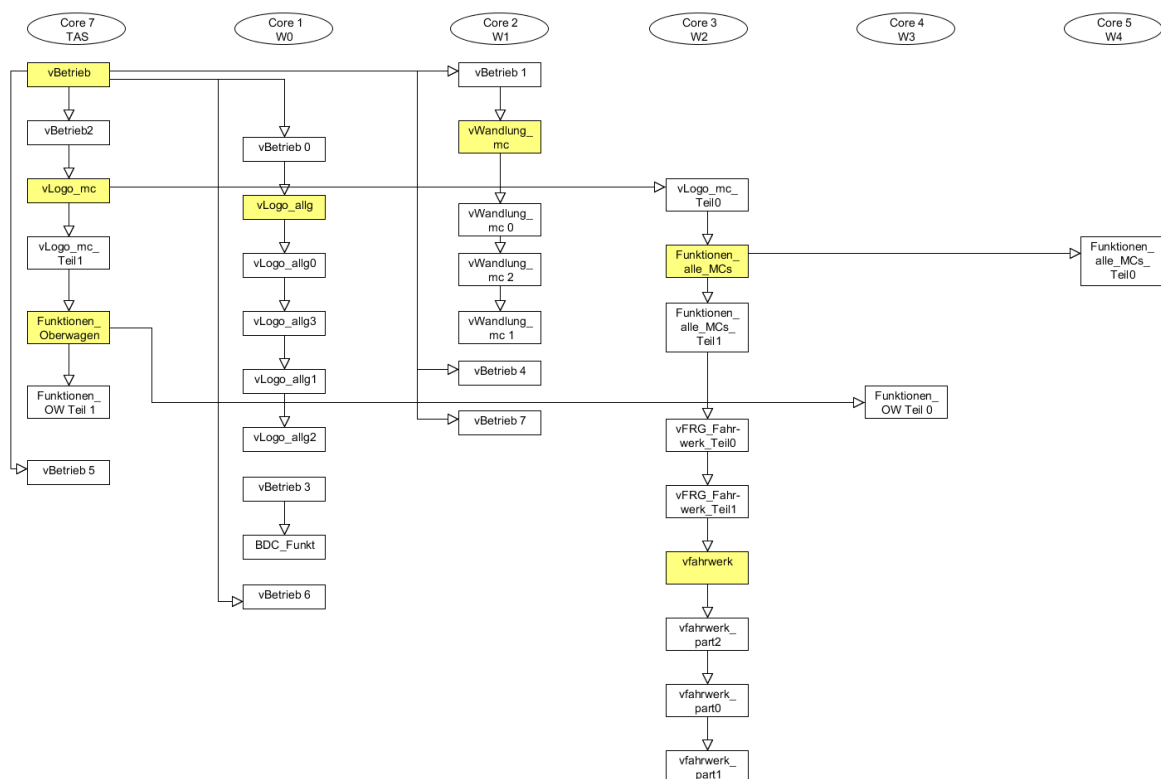


Figure 46 Configuration for 8-Cores (6 cores main-loop + 2 cores for periodic tasks). The yellow squares are the TAS-roots.

The TAS run the code, depending on the number of available cores, sequentially or in parallel. In the example of "vBetrieb" you can clearly see that Core 7 distributes the work on Core 1 and Core 2 and

does one part of the work on its own. In the example of "vLogo_allg" on Core 1, you can see that the pieces of code can also be executed sequentially. This has the consequence that a run of the same software with a different number of cores requires only a change in the TAS configuration file.

In the 3-core version, all TAS instances run sequentially. This version uses two cores for the periodic tasks. The main loop is executed on one core. Since the same software is to be used for more cores, the TAS run without workers. In the 4-core version, the work is performed with one TAS worker. However, if there a TAS instance is called by another TAS instance, then the original TAS instance cannot be executed in parallel since no more workers are available.

6.3 Simulation of parallel Code

To support the application programs, it was task of work package 4 to develop common domain independent system architectures for a many-core processor and domain specific runtime environments. These components have to serve as base for the application. For the construction machinery domain there is no standard which the new RTE has to implement. Therefore, the main function of the BIOS is to link the application software to the system software respectively the parMERASA simulator. The BIOS is designed in that way that the application software does not need to call different functions than in the original system.

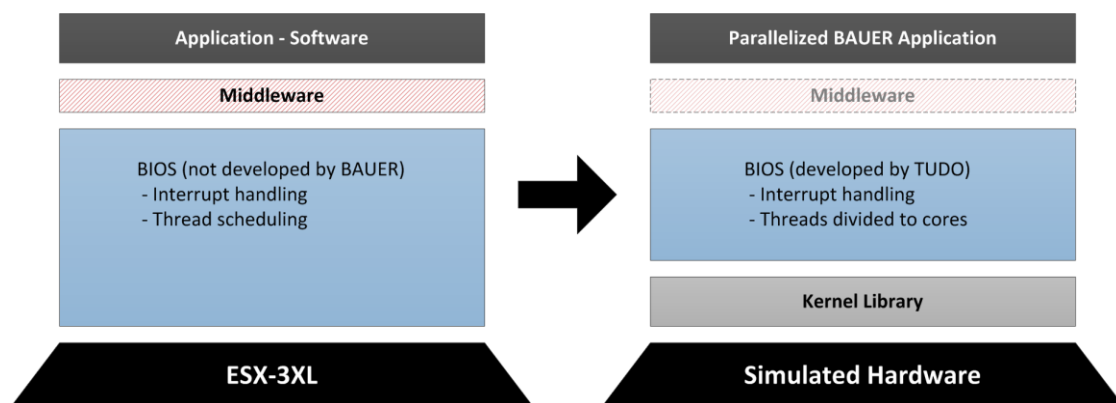


Figure 47 Changes from original single target system to parMERASA multi-core system

As Figure 47 shows there is now the additional layer of the kernel library between the BIOS and the simulator hardware in the parMERASA version of the application. All OS functionality of the original BIOS is now integrated in kernel library and therefore not part of the new BIOS.

The software has been prepared to run on the simulator. By remembering the system time at the beginning of an iteration of the main control loop and comparing it with the system time at the end, the cycle time of this iteration can be calculated.. By running the main loop more often, and recording the largest cycle time, the measurement-based WCET can be determined.

6.4 WCET analysis

Interpreting the theoretical results shown in Figure 45, several configurations have been chosen to implement. There are several ways to determine the WCET of these: The first way is to observe the application. For this purpose, the time of a main-loop run is measured. After several such measurements the largest measured time is taken as Observed WCET. But since it is here crucial to choose the longest paths, a computed WCET is more significant. For this purpose, the software can be analysed with OTAWA and RapiTime.

However, the analysis tools need to know which core runs which part of code. To guarantee this, the TAS were adapted by the University of Augsburg to be able to determine which core runs which code before running the application. In order to achieve an optimal distribution, the behaviour in the simulator without fixed cores was observed and then the TAS were configured to use these cores in a fixed way. Now the behaviour is the same, but it is clear and guaranteed which cores are used. The basis for the calculation of the speedup is the single core version running on the simulator.

The results of the OTAWA analysis can be seen in Table 26. The more cores there are, the more the results look unrealistic. For example for the 12 Core-Version, the WCET is 300 times bigger, than the results with RapiTime. There are some possible reasons for this. Maybe the Input for the OTAWA-Tool should contain more information than it is currently being specified. OTAWA needs some information about the TAS, shared variables, locks and barriers. Maybe these are not yet enough to get good results. One other possible reason is, that the way how to calculate the WCET, is different in OTAWA and RapiTime. The BAUER application is a huge application and very complex. The application was too big to analyse it in one step. Some functions had to be analysed separately. This inevitably added pessimism to the global analysis. The application is not easy to analyse, also in the single-core version. Analysis of a parallel version is new for the analysis-tools, there is a lack of experience. At the end of parMERASA there was no time left in order to better optimize the WCET analysis. We did not have time to do a refined analysis that includes an initial state of the memory. This results in a data cache analysis predicting a lot of misses.

Table 26 Results of the WCET-Analyses with OTAWA

Number of Cores	WCET in cycles for the main-loop
1 Core	11,989,192
4 Cores	8,172,637
8 Cores	203,366,331
12 Cores	1,199,706,222

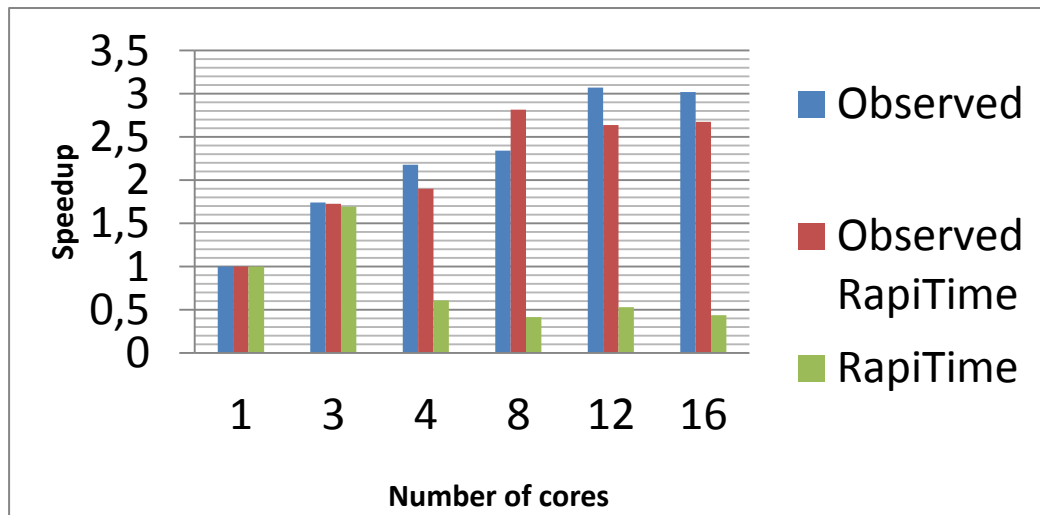


Figure 48 WCET Speedup for multiple application configurations

In Figure 48, you can see the speedup for five multicore-versions compared with a single-core version. All these configurations were analysed manually and by RapiTime. RapiTime delivers two kinds of information, the maximum observed time and the calculated WCET. The manually observed WCET is nearby the maximum observed time by RapiTime. So it seems to be a realistic way. The green bar is the calculated WCET. These numbers get higher with more cores (lower speedup). One explanation, which is justified by the analysis, is the synchronization-overhead. For example, there is one big array (8192 bytes), which is a copy of the EEPROM in the code. This array is used by nearly all cores, but it is only protected by one lock-variable. If one core accesses this array, all the other cores have to wait to get access to the array. Most of the cores do not use the same section of the array. One possible idea to optimize would be to use more locks to be able to lock only a section of this array.

In Figure 49, you can see a comparison between the theoretical execution time, the observed execution time and the maximum observed Time be RapiTime. The observed Times are nearby the theoretical execution time. Therefore, it can be seen that the best configurations were chosen and the estimation was quite good..

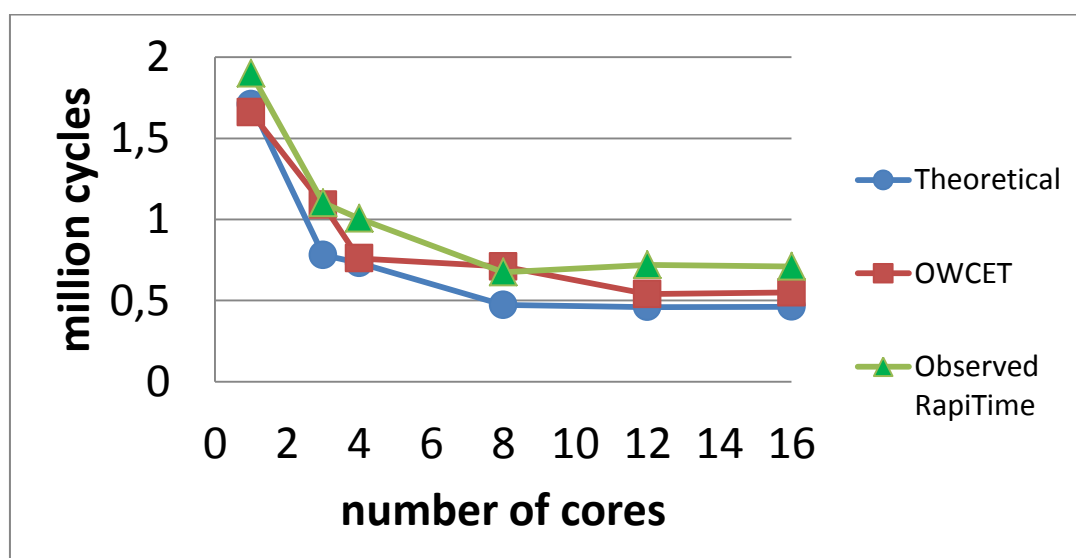


Figure 49 Result of the WCET analyses with observed, RapiTime

6.5 Conclusions and outlook

The WCET analysis was a hard task due to the size and complexity of the foundation crane program. Therefore, the resulting numbers should not be seen as final as there is still potential to improve them. The WCET tools might get better when getting more experience and with their feedback the foundation crane program could be optimized to result in a better WCET.

The current results show that it is possible to parallelize a hard-real time industrial control program and to get a WCET speedup. This currently applies especially to the "small" parallelization configurations, i.e. those employing only a few cores. When using more than four cores, there is still a speedup when looking at the observed times, but the calculated WCET shows a slowdown.

Therefore, the insights of parMERASA can be used to move our code to a small multicore processor, as soon as available in an electronic control unit by our hardware supplier Sensortechnik Wiedemann GmbH. The usage of "big" multicore processors still has to be further investigated.

Nevertheless, if more cores are available, these could be used for extensive and exact computations which are currently not possible due to the restricted computation power. This concept of "slack time" has already been explained in chapter 2.5.

To make the foundation crane control program better analysable, it is necessary to treat it as a mixed-criticality system. It has to be separated into hard real-time, soft real-time and those parts which do not have to fulfil any real-time requirements. Currently, all parts of the program are treated as hard real-time. Also, the TAS are made for hard-real-time systems, where all program parts are of the same priority. But, for example to show production-data in the operating screen, no hard real-time requirement is necessary. One possible way for BAUER would be, to execute the hard-real-time parts of the code on dedicated cores and the other (soft real-time) parts on other cores. In this case, it would be an idea to give the hard-real-time cores prevailed access to the memory to minimize the synchronisation-overhead.

BAUER wants to use the results of parMERASA to prepare the source code of its machines for better parallelizing. To do this in an effective way, an in-depth analysis of our specific parallelization needs has to be done. Like discussed above, the BMA software has parts which must be handled as hard-real-time systems, but also parts where no hard real-time mechanisms are needed. Then, parallelizing the code should be easier and more effective.

Also, the dependency analysis of the program helps us to check dependencies and find solutions to minimize contention and synchronization overhead. The WCET analysis of the parallel program gives us an informative basis where we have to improve program parts to be able to make an efficient parallel control program when the necessary hardware is available.

7 CONCLUSIONS

In this section, we summarize the learnt lessons for the WCET-aware parallelisation and analysis on a multi-core system for each industrial prototype (see Section 7.1). Furthermore, we provide a progress report with respect of each one of the parMERASA SMART Objectives in Section 7.2.

7.1 Summary on the WCET-aware parallelization on a multi-core system

In parMERASA WP2, **UAU** introduced the structured parallelism through the parallel design patterns. Furthermore, a set of synchronization idioms has been analysed and different implementations of the barrier synchronization has been explored. The experimental results advocate the benefits of applying structured parallelism and the usage of barrier synchronization for parallelised industrial applications.

In the avionics domains, **HON** contribute with two applications – 3DPP and SN, which were WCET-aware parallelised and analysed. For the **3DPP** application, we concluded that the WCET-aware parallelization can achieve up to 4 times WCET speedup for 16 threads application configuration (see Figure 9). Since the WCET efficiency was relatively low with high number of threads, we considered the optimal application configurations to be up to 4/8 threads, where the WCET Speedup is close to the theoretical WCET speedup. Furthermore, our analysis suggested that the parallelization with 16 cores is not WCET efficient, which was the inflection point where the synchronization time (and memory access delay in parMERASA platform) becomes the dominant factor in the execution time of each core. The low WCET efficiency with high number of threads can be compensated by applying a preemptive operating system, which schedules other threads on the same processors. Note that all presented WCET estimations on the parMERASA were based on simple mapping scheme, where 1 thread runs on 1 core. Furthermore, our WCET analysis suggests that application applying barrier synchronization primitive results in higher WCET speedup compared to conditional variables primitives. In summary, we concluded that the overall WCET analysis for the 3DPP application is strongly influenced by i) the ratio between computation time and synchronization, ii) the granularity and load-balancing of the code decomposition, and iii) the size of the local cache.

The computational complexity of **SN** exceeded application multiple times computational complexity of 3DPP application. Such high SN computational complexity leveraged a full advantage of the multi-threading by exploiting data and tasks (i.e., pipelining) parallelism. Our WCET experimental results suggested that no improvement in SN end-to-end WCET was observed when a simple task-level pipeline was exploited in 7T application configuration, because all SN threads are still sequentially executed. On the other hand, an improvement of the end-to-end WCET was feasible by increasing the parallelization of SN application (e.g., with 16_CAM SN application configuration). However, our analysis also reveals that there was always a trade-off between the end-to-end and the pipelining-stage WCET improvements. Therefore, for significant application end-to-end WCET speedup optimization, we recommend parallelization on processing stages, which have a major contribution to the end-to-end application WCET and operates under the sweet spot.

In the automotive domain, **DNDE** achieved up to 6 times speedup exploring various levels of parallelization – intra-Runnable, inter-Runnable, and introducing supertasks. The concept of timed implicit communication is presented as approach to extract parallelism at task level. Application data is redirected to a buffer to decouple tasks. For the examined EMS prototype, the buffer approach has proven to provide a speed-up for the worst-case scenario of 4.5 on 8-cores.

The parMERASA Mapping Tool is presented as approach to extract parallelism at Runnable level of AUTOSAR tasks. The Mapping Tool distributes Runnables of a task to the available cores such that precedence constraints among Runnables are respected. The observed results were of mixed nature with varying WCET speed-up from 1 times up to 3.3 times. The WCET speedup has been further improved with a supertask concept.

The pattern-supported parallelization approach has been applied to extract intra-Runnable parallelism. Therefore, a selection of the most promising Runnables and two additional frequently used functions were examined. We analysed the code of all Runnables to disclose the data dependencies and afterwards applied the Parallel Design Pattern to reveal parallelism. Afterwards two of the Runnables were identified as suitable for. Three different implementations were utilized for 2-core and 4-core multi-core platform instances. The observed WCET speed-ups were in a range from 0.72 to 2.31 for Runnables and from 0.42 to 2.36 for the additional functions.

Eventually, a combined approach has been presented. Here, the Mapping Tool and TIC were combined. The task limiting the parallelism was distributed on 2 cores using the parMERASA Mapping. With this approach a speed-up of 5.97 for the worst-case scenario was observed. TIC is well suited for the scheduling of multiple tasks on a larger number of cores (>4). The Mapping Tool is well suited for the scheduling of a single task or supertasks on a small number of cores (≤ 4). Intra-Runnable parallelism is most suitable for parallelization of frequently used Runnables or Runnables on the critical path.

A transfer of the methods to COTS multi core processors without strict real time support is required. The inter-task communication dependencies will be further investigated. Inter-task communication schemes have to be evaluated with respect to their effect on the physical behaviour of a diesel engine.

In parMERASA, the construction machinery domain was presented by **BMA**. The WCET-aware parallelization of their application suggested that multi-core platforms are applicable and WCET speedup achievable for hard-real time domain of construction machinery. The presented analysis in this deliverable advocates that a reasonable WCET speedup was achievable even with low number of cores. Although there was still WCET speedup with high number of cores compared to sequential execution, the achieved WCET speedup was lower than the obtained with low number of cores.

Our analysis concluded that efficient WCET-aware parallelization is possible in case the foundation crane control program parallelism was categorized and split among hard real-time, soft real-time and no real-time domains. A feasible way to execute mixed-critical workloads would be to perform threads mapping to cores with respect to the criticality domain, i.e., only threads from the same criticality domain can be mapped on the same core. In this case, it would be an idea to give the hard-real-time cores prevailed access to the memory to minimize the effects of the synchronisation-overhead. Overall, the WCET analysis of the parallel program and the RapiTime dependency tool provided an informative basis for potential future application improvements.

7.2 WP2 progress with respect of the parMERASA SMART Objectives

In this section, we list the status of the WP2-relevant **parMERASA Smart Objectives**. Note that the progress on the Smart Objectives was performed during the entire project and it is not limited to the timeline of parMERASA Deliverable D2.6.

O1: Develop a **software engineering approach targeting WCET-aware parallelization techniques** and parallel execution patterns that favour WCET analysis. The software engineering approach should **define a development path leading from sequential legacy programs to parallel programs** and maximise design artefacts reuse. The project will develop **at least four patterns that are analysable**. These patterns should be applicable to both commercial off-the-shelf (COTS) multi-cores and the parMERASA platform and will be demonstrated during the case studies.

The main contributor to O1 objective was UAU. The parallelization approach start with single-core software was developed and is called *pattern-supported parallelization approach* [8, 7]. It is suitable for hard real-time embedded systems if it is applied with the parMERASA Pattern Catalogue. The parMERASA Pattern Catalogue [3, 9] describes four timing-analysable Parallel Design Patterns. As conclusion, all respective targets of Smart Objective 1 were reached by UAU.

The presented parallelization approach was directly applied by BMA and HON. For the parallelization of automotive software it was only possible to apply the approach to a small subset of Runnables. This is due to the different structure of AUTOSAR compliant automotive code with hierarchical and fine-grained organization. The parallelization of the software by the application partners was supported by UAU as defined in the description of work.

O2: **Achieve parallelization** of the industrial case studies by applying the software engineering approach and suitable parallel execution patterns. **Deliver higher performance than single core processors and achieve at least an eightfold WCET speedup** (WCET of the sequential program divided by the WCET of the parallel program). This metric will be demonstrated by applying the parMERASA WCET analysis tools on the avionics, automotive, and construction machinery case studies.

All industrial prototypes have been successfully parallelized with WCET speedup > 1 compared to the original sequential application configuration. Furthermore, all application prototypes have been successfully analysed by the parMERASA WCET tools – RapiTime and OTAWA. The achieved WCET speedup goes upto six times. Therefore, we consider that O2 is successfully achieved with a minor deviation that the achieved speedup was lower than the originally projected.

O3: Develop on target verification tools, in particular WCET analysis tools with less than 25% pessimism on WCET estimates of parallel programs. Further tools to be developed concern code coverage and memory analysis as well as parallel program profiling and visualisation. This objective will be demonstrated by review of evidence that can support a certification argument and by applying the tools to the case studies.

The improved WCET tools – RapiTime and OTAWA were successful in the code analysis with each one of the application prototypes. Further details on the tools performance and performed optimizations are presented in parMERASA deliverable D3.12.

O4, O5, O6 - the system architecture, system-level SW, and standardization contributions were developed in WP4 and their correct execution was verified with the applications from WP2.

O7: Contribute to Open Source software. The software developed at the universities, i.e. the static WCET tool OTAWA, the developed system software and the parMERASA simulator, will be made publicly available under an Open Source license at the end of the parMERASA project.

The timing analysable algorithmic skeletons developed by UAU were released as open-source software under LGPLv3 license, available for download at <https://github.com/parmerasa-uau/>. By this open source contribution, we consider O7 to be successfully met by WP2.

8 REFERENCES

- [1] 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.
- [2] 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.
- [3] 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.
- [4] R. Jahr, A. Stegmeier, R. Kiefhaber, M. Frieb, T. Ungerer. User Manual for the Optimization and WCET Analysis of Software with Timing Analyzable Algorithmic Skeletons. Technical Report 2014-05, Fakultät für Angewandte Informatik der Universität Augsburg, 2014.
- [5] 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.
- [6] R. Jahr, M. Frieb, M. Gerdes, T. Ungerer, A. Hugl, and H. Regler. Paving the way for multi-cores in industrial hard real-time control applications. In *9th IEEE International Symposium on Industrial Embedded Systems (SIES), WiP Session, 2014*, June 2014. (Accepted and presented but not available online yet.)
- [7] 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.
- [8] 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.
- [9] R. Jahr, M. Gerdes, T. Ungerer, H. Ozaktas, C. Rochange, and P. G. Zaykov. Effects of structured parallelism by parallel design patterns on embedded hard real-time systems. In *20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014*, August 2014. (Accepted and presented but not available online yet.)
- [10] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3):202–210, 2005.