Int J Parallel Prog (2016) 44:1296–1336
DOI 10.1007/s10766-016-0432-7

CrossMark

A Parallelization Approach for Hard Real-Time Systems and Its Application on Two Industrial Programs

Strategy and Two Case Studies for the Parallelization of Hard Real-Time Systems

Martin Frieb[1] · Ralf Jahr[1] · Haluk Ozaktas[2] · Andreas Hugl[3] · Hans Regler[3] · Theo Ungerer[1]

Received: 30 November 2014 / Accepted: 25 April 2016 / Published online: 3 May 2016
© Springer Science+Business Media New York 2016

Abstract Applications in industry often have grown and improved over many years. Since their performance demands increase, they also need to benefit from the availability of multi-core processors. However, a reimplementation from scratch and even a restructuring of these industrial applications is very expensive, often due to high certification efforts. Therefore, a strategy for a systematic parallelization of legacy code is needed. We present a parallelization approach for hard real-time systems, which ensures a high reusage of legacy code and preserves timing analysability. To show its applicability, we apply it on the core algorithm of an avionics application as well as on the control program of a large construction machine. We create models of the legacy programs showing the potential of parallelism, optimize them and change the source codes accordingly. The parallelized applications are placed on a predictable multi-core processor with up to 18 cores. For evaluation, we compare the worst case execution times and their speedups. Furthermore, we analyse limitations coming up at the parallelization process.

Keywords Parallelization · Parallelization approach · Model-based · Parallel design patterns · Algorithmic skeletons · Real-time · Embedded · Control code · Case study

✉ Martin Frieb
martin.frieb@informatik.uni-augsburg.de

[1] Department of Computer Science, University of Augsburg, 86135 Augsburg, Germany

[2] Université Toulouse III - Paul Sabatier, 118 route de Narbonne, 31062 Toulouse, France

[3] BAUER Maschinen GmbH, BAUER-Str. 1, 86529 Schrobenhausen, Germany

1 Introduction

Industrial embedded applications often have code bases which have grown and improved over many years. Of course, they should also benefit from the advantages of multi-core processors. However, the parallelization of legacy single-core software is challenging [38]. It is even harder for *embedded hard real-time software* because timing constraints and other non-functional requirements have to be met. Therefore, usually a timing analysis takes place before these applications can be employed. To support timing analysis, it is necessary to follow implementation rules, e. g. no dynamic memory allocation, no pointers, etc.; see details e. g. in [5,41]. Our focus is how to parallelize legacy embedded hard real-time applications. Their sequential implementation is already built in a timing-analysable way and we want to get a parallel version which is timing-analysable, too.

The common parallellization approaches like the PCAM approach by Foster [15] or the pattern-based parallelization approach by Mattson et al. [37] do not respect non-functional requirements, making a timing analysis almost impossible. Another attempt is automatic parallelization as proposed by Cordes and Marwedel [11], Cordes et al. [12] or Kempf et al. [32]. This also leads to parallel code not suitable for timing analysis. Since these established approaches do not work for applications requiring a timing analysis, our *parallelization approach for hard real-time systems* closes this gap. It combines the benefits of the parallelization approaches of Foster [15] and Mattson et al. [37] while introducing parallelism only in a way facilitating timing analysis.

In this article, we first describe our parallelization approach for hard real-time systems, then we show the complete process of parallelization of two industrial applications: one from the avionics domain and one from the automation domain. The presented approach is not limited to these examples, but may be applied to other industrial applications, too. Our process starts from the sequential legacy code and results in a multi-core implementation and a static WCET analysis.

The structure of our article is as follows: we first give some backgrounds in the following Sect. 1.1, then an overview over related work in Sect. 2. Afterwards, we describe our parallelization approach for hard real-time systems in Sect. 3. An avionics application and its parallelization are depicted in Sect. 4. In the Sects. 5, 6, 7, 8 and 9 the control program of a foundation crane is described (Sect. 5), parallelized following our parallelization approach (Sects. 6, 7 and 8) and evaluated (Sect. 9). Finally, we conclude with lessons learned in Sect. 10 and summarize the results in Sect. 11.

1.1 Background Information

Real-time means that the system has to provide results in due time, i. e. within a given deadline [59]. We distinguish between firm and hard real-time systems: in firm real-time systems the result of a computation looses its relevance when the deadline is missed (e. g. a GPS position of a moving car), while in hard real-time systems the miss may result in harm or damage (e. g. an airbag not responding properly). To guarantee that the deadline holds, the worst case execution time (WCET) is estimated [59]. This

means an upper bound of the longest path which may be taken during the execution of the program has to be found and estimated to ensure execution is finished before the deadline is over.

There are two families of approaches for performing the WCET analysis of applications: static WCET analysis and measurement-based WCET analysis. Each approach is complementary to the other. Static WCET analysis techniques are based on a timing model of the hardware architecture and reach upper bounds of the real but not computable WCET. Measurement-based approaches derive timings of small blocks of code by direct observation of the execution of the program on the target platform. Thus, they avoid the effort of building an exact timing model of the processor, but there is no guarantee to catch the real WCET.

Our parallelization approach is an extension of a pattern-supported parallelization approach for embedded real-time systems [27,28], which was developed in the parMERASA project [56]. We describe how it works and apply it on the two industrial applications. Its basic idea is to create a model of the application, optimize it for a parallel execution and then change the source code accordingly. In the model, parallelism can only be introduced with parallel design patterns (PDPs). Design Patterns are a textual description of best practice solutions for recurring problems [37]. PDPs focus on how to introduce parallelism in different situations.

Currently, our applications are build in a timing-analyzable way and we want them to stay analyzable. Therefore, we collected PDPs respecting timing constraints in the parMERASA Pattern Catalogue [18]. It contains the four PDPs *Task Parallelism, Periodic Task Parallelism, Data Parallelism* and *Pipeline Parallelism*:

- *Task Parallelism* A number of tasks are executed concurrently and the further execution of the program is suspended until they are all completed. The WCET is mainly defined by the longest WCET of the subtasks. To be timing-analysable, the tasks have to be scheduled and mapped statically.
- *Periodic Task Parallelism* Several tasks are executed periodically with equal or different periods. When the parallel WCETs are determined, it has to be checked if the periods and deadlines still hold or could be missed in the worst case.[1] In the latter case, additional cores might be utilized.
- *Data Parallelism* Computations on a data structure are performed, which can be decomposed into concurrently computable chunks. Therefore, the same algorithm can be applied simultaneously to several parts of the data structure. Since the same computations take place for different input data, the WCET should be similar for all computed chunks.
- *Pipeline Parallelism* The executed computations on input data can be divided into several stages. After data has been processed in one stage, it is handed over to the next one. Afterwards, the finished stage can process the next set of data. Hence, the data is processed in a chain of producers and consumers. Ideally, the stages are load-balanced, i. e. their workloads obtain similar WCETs. To achieve this, stages

[1] Due to e. g. synchronization overheads, some parts of the program might take longer. Therefore, it is important to keep an eye on periods and deadlines and to check if everything still works fine.

might be joined or split, or further PDPs may be applied within single stages. For the WCET analysis, each stage and the data exchange between stages have to be analyzed.

We measure the success of our parallel implementations by determining the WCET speedup. It is defined by dividing the WCET of the sequential implementation through the WCET of the parallel implementation.

Our case studies are the core algorithm of an avionics application and the control program of a large construction machine:

The avionics application gets sensor inputs, processes them via fast Fourier transformation (FFT), matrix multiplication, summarizing a set of matrices and computing an inverse FFT as output. Unfortunately, we cannot go into further details what its purpose is.

The construction machine is the BAUER foundation crane MC 128 (engine power 709 kW, max. boom length 54.4 m, weight of the base unit 170 t, max. capacity 200 t [6]). It can be used for different drilling or milling techniques as well as BAUER dynamic compaction (BDC), which is an improved "implementation" of dynamic soil compaction (see e. g. [35]). This technique is performed in the control program we analyze and parallelize. The job operated by the machine is to uplift a pounder with a weight of around 20 t to a height of around 30 m and afterwards drop it in free fall. Then, the process is repeated until the ground is patted to be able to build a large building on it.

Because the machine has to react in due time the application is hard real-time. When the brakes are put on at the wrong moment (while the pounder is in free fall), it could mean damage to the machine. The control program consists of a main control loop with additional interrupts interwoven in a complex way to meet the timing requirements. It also has to do a lot of other work concurrently, like checking many sensors, setting actuators, reacting on inputs from and informing the driver, etc.

Therefore and because its structure is widely used in automation domains, the application is typical for industrial applications. Because of its structure and its future performance needs (more features, especially for safety and security are expected), it would be desireable to run it on a multi-core platform. However, changes to the software require a lot of tests on the real machine to be sure that everything works properly. This is a quite expensive task, especially when something unexpected happens. Thus, a systematic parallelization approach should be applied.

2 Related Work

We give an overview on the parallelization approaches our approach is based on in Sect. 2.1. Since we employ PDPs, related work about them is topic of Sect. 2.2, while we deal with their representation on code level—algorithmic skeletons—in Sect. 2.3. Finally, in Sect. 2.4 we deal with existing researches focussing on the parallelization of hard real-time software.

2.1 Parallelization Approaches

So far, there are mainly two well-known development approaches for the transformation of sequential software into parallel software, both are model-based and were defined for the high-performance domain, i.e. the *PCAM-Approach by Foster* [15] (divided into the four phases Partitioning, Communication, Agglomeration and Mapping) and the *parallelization approach by Mattson, Sanders, and Massingill* [37] (which is utilizing PDPs). Both approaches do not target embedded systems and the observance of timing constraints.

Inspired by above mentioned approaches, Jahr et al. introduced a new approach for the embedded systems domain targeting multi-core processors and respecting timing constraints [27,28]. To our knowledge, it is the only one focussing on embedded real-time systems. It introduces parallelism only by PDPs. Our *parallelization approach for hard real-time systems* presented in this article extends the approach by Jahr et al. by a systematic WCET analyzable implementation of algorithmic skeletons as well as a refinement loop, see details in Sects. 2.3 and 3.

All these parallelization approaches have to be done manually or semi-automatically with tool-support. Instead, automatic parallelization could also be applied. Cordes and Marwedel [11], Cordes et al. [12], for example, developed an approach with a *hierarchical task graph* as model which can be optimized automatically, too. Another approach with automatic parallelization is followed by Kempf et al. [32], who focusses on industrial applications of a specific type and comes to similar speedups like us. However, automatic parallelization leads to unstructured parallelism making later timing analysis much harder; structured parallelism is very beneficial for this (cf. [29,41]).

Moreover, the structure of an industrial control code with its timer-based interrupt service routines and without loops over large data structures is not well suitable for automatic parallelization.

2.2 Pattern Catalogues for Parallel Design Patterns

All PDPs to be used in the parallelization process are taken from a Pattern Catalogue, which can be more general, but also domain or application specific. To describe a pattern, a common structure called *meta-pattern* is used.[2] Well-known design patterns observed in parallel software have already been collected and described by Mattson et al. [36,37] and arranged in the classification system *our pattern language* (OPL, [33]) consisting of multiple levels.[3]

In the parMERASA project [56], we developed our own pattern catalogue from the existing ones and an analysis of industrial applications. It is called the *parMERASA pattern catalogue* [18] and contains four PDPs and two synchronization idioms. These synchronization idioms are, in contrast to the PDPs, platform dependent. They define

[2] For example, the meta-pattern in OPL (see http://parlab.eecs.berkeley.edu/wiki/patterns/patterntemplate) requires the specification of name, problem, context, forces, solution, invariants, an example, known uses, related patterns, references, and author.

[3] See online version: http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.

the timing-analyzable synchronization prototypes provided by a target platform. Also, we extended the meta-patterns by *real-time prerequisites*, *synchronization idioms*, and *WCET hints* denoting the requirements and consequences for WCET analysis.

The parMERASA pattern catalogue is utilized for the parallization both of the avionics application and the control code of the foundation crane.

2.3 Algorithmic Skeletons

Algorithmic skeletons are software libraries providing parallelization concepts for applications and can be understood as equivalent of PDPs on code level. Originally, they were introduced by Murray Cole and the concepts were further improved [7,10]. Because algorithmic skeletons are code, they are platform and programming language dependent. A detailed overview on existing skeleton libraries can be found in [21].

There are only a few algorithmic skeleton libraries which try to address real-time requirements: *A Parallel Skeleton Library for Embedded Multicores* [34], QUAFF [14] and SkiPPER [53]. The first one is written for C++ and uses templates and inheritance. Since our industrial applications are written in C and the target platform is also built for C code, we cannot employ it. QUAFF and SkiPPER only focus on firm real-time requirements just like image processing, not on hard real-time requirements for embedded systems. Furthermore, SkiPPER is domain-specific and only provides limited nesting capabilities.

To fill the gap, we developed the *Timing-analysable Algorithmic skeletons* (TAS) [30,54] in the parMERASA project und utilize them in our parallelization approach for the parallization of our industrial applications.

2.4 Parallelization of Hard Real-Time Software

This article presents a part of the work from the parMERASA project [56]. There, three other industrial hard real-time applications were parallelized by applying the original pattern-supported parallelization approach by Jahr et al. [27,28]. These applications are a 3D path planning algorithm and an algorithm for stereo navigation from the avionics domain and a diesel engine motor injection algorithm from the automotive domain. For results, see [55,57].

Hereby, Kehr et al. [31] focus on the parallelization of the diesel engine management system (EMS). Analysing their code with OTAWA [5], they reach WCET speedups up to 4.5 on 12 cores (assuming no worst case buffer overhead). They describe their application as an “an ideal use case” for parallelization—therefore, it might be a good reference for comparison how big the parallelization potential is.

Gerdes et al. [20] modified the control application of a large drilling machine for a timing-analyzable multi-core processor with two or four cores, which is a different construction machinery series by BAUER. A WCET speedup of 1.93 was reached on four cores; RapiTime [47] was used for the measurement-based WCET analysis. However, the parallelization was based on knowledge of the application (i. e. domain knowledge), whereas the approach followed here is more systematic and model-based.

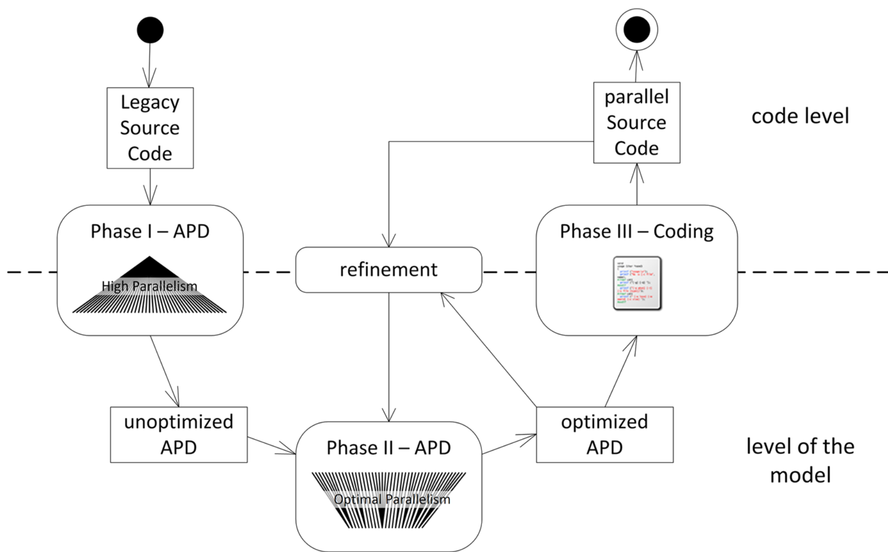

Fig. 1 The three phases of the pattern- and skeleton-supported parallelization approach are I. Revealing Parallelism, II. Optimizing Parallelism and III. Implementation. From the timing analysis of the parallel code, the optimization may be refined iteratively to improve the results

3 Parallelization Approach for Hard Real-Time Systems

We enhance the model-based *pattern-supported parallelization approach*, which was introduced by Jahr et al. [28]. Its applicability for embedded hard real-time systems is discussed in [27]. In this section, we describe the concepts of the original approach and our extensions.

The goals of the parallelization approach are to provide a way to parallelize legacy single-core software of embedded systems, while keeping the development effort low and the software timing-analysable. Its main idea is to introduce parallelism only through PDPs in a model. They are taken from a pattern catalogue. On code level, PDPs are represented by algorithmic skeletons. This way, analysis of timing behaviour is eased since parallelism is applied only in a structured way (cf. [29,41]).

Throughout the parallelization process an extended version of the UML2 activity diagram (AD) [40] called *activity and pattern diagram* (*APD*) is created and worked on. This allows rapid definition and refinement of situations exhibiting chances for parallelism. It is further described in Sect. 3.1.

Unfortunately, there is currently no tool to find situations in code where parallelism could be applied. Hence, an experienced software developer or engineer has to find them manually. However, there are lots of tools for UML diagrams and some to check dependencies on code level, e. g. CScope.[4]

As illustrated in Fig. 1, the parallelization approach for hard real-time systems is organized in three phases:

[4] Homepage: www.cscope.sf.net.

– Phase I: Revealing Parallelism
– Phase II: Optimizing Parallelism
– Phase III: Implementation

In the original approach [27,28] there are only the phases I and II, but we added a Phase III, because of the platform specific implementation, which is further supported by timing-analyzable algorithmic skeletons and code generation. Additionally, refinement is added to iteratively improve the results. The phases are described in Sects. 3.2, 3.3 and 3.4.

3.1 Activity and Pattern Diagram (APD)

Modelling is done using a so-called activity and pattern diagram (APD), which gives a formal model of the structure of the application and can be utilized for documentation and maintainability. Each activity represents a code block of the original program, e. g. a function, several functions or a part of a function.

In addition to the elements of the basic UML2 AD, a second kind of activity node is added to model the PDPs (cf. Fig. 2). It can be used and behaves exactly the same way as the (sequential) activity nodes but can encapsulate not only a single but multiple functionalities modeled as separate APDs. As can be seen in Fig. 3, the usage of the bold "fork" and "join" bars existing in the basic AD to model creating and joining threads is not allowed anymore.

It is important to model dependencies (e. g. shared variables) because they restrict parallelism. Furthermore, dependencies have to be respected at scheduling. Therefore, activities and patterns should be annotated with *input* (read global data), *output* (write

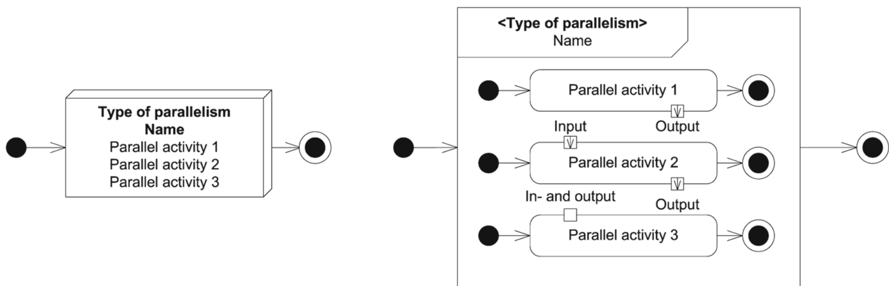

Fig. 2 New activity node representing a PDP: on the *left side* in compact shape, on the *right side* detailed. The pins represent shared variables and data structures

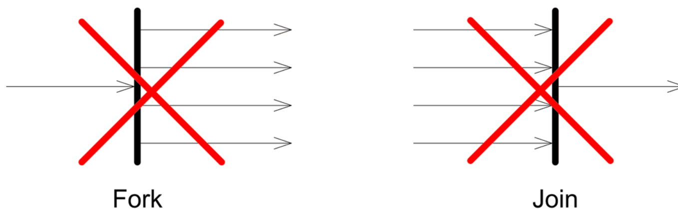

Fig. 3 In the APD, the use of "fork" and "join" is not allowed anymore

global data), and *resource pins* (read and write global data) exposing the usage of data structures and shared resources like special devices or interfaces. Also, interrupts can be annotated in a similar way. The APD may be further extended to model more functional and non-functional aspects.

APDs can describe a program in a less or more detailed way and therefore form a hierarchy, which is illustrated in Fig. 4: with more details, the activity *Program* can be decomposed into the activities *Initialization* and *Execution*. The latter consists of one sequential activity and a PDP, which contains a sequential activity and another PDP. The APD resulting in Phase I and being optimized in Phase II comprises all hierarchy levels and contains all details. Most APDs for the foundation crane only show few details or a small part from the bottom of the hierarchy. Due to confidentiality reasons and because it is too large, the foundation crane's APD is never shown completely.[5]

3.2 Phase I: Revealing Parallelism

The goal of Phase I is to expose a high degree of parallelism in an APD, which can be optimized in Phase II for the target multi-core architecture. Thus, not only an APD has to be created, but also all dependencies have to be found and WCETs of the sequential activities have to be determined. Following the suggestions of the PCAM-Approach [15] "a high degree" means that there should be more possibilities for parallelism than are feasible for the target platform. Hence, reordering of independent parts and optimizing can be done in Phase II.

In the top of Fig. 4, the starting point can be seen: a sequential software represented by a single activity. Two operations are to be applied to increase the degree of details in the diagram:

1. *Replacing* an activity with a PDP. The PDP is composed of several activities, which may be executed in parallel. This operation should always be preferred over splitting because it increases parallelism.
2. *Splitting* an activity into several (sequential) activities to be executed one after another. This does not increase parallelism, but more details are added for this program part.

These operations should be repeated until it is clear that the degree of parallelism is high enough. Throughout the process, the decomposition of an activity should have the same semantic meaning as the activity itself.

Dependencies between the activities are to be determined by a dependency tool, (e. g. CScope[6] or the Rapita Dependency Tool[7]) and also entered into the diagram using pins.[8]

---

[5] However, the XML example input file of our speedup approximation and parameter optimization tool gives a clue of the complete APD. It can be found at https://github.com/parmerasa-uau/parallelism-optimization/tree/master/ParallelismAnalysisJMetal.

[6] Open-source, Homepage: www.cscope.sf.net.

[7] Part of the Rapita Verification Suite (RVS), Homepage: www.rapitasystems.com.

[8] Besides Figs. 2 and 13 these pins are never shown to keep the diagrams compact.

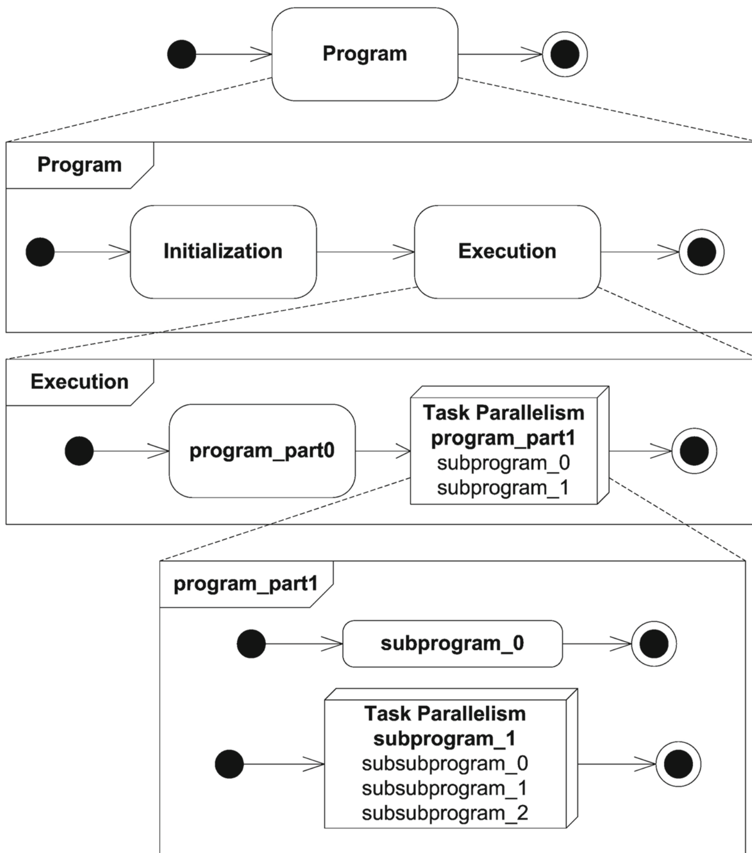

Fig. 4 APD of the program including several hierarchy levels: the analysis starts in the top with a single activity, below there is one level more of details exposing two sequential activities. The second one of them called *Execution* is composed of a sequential activity and a PDP. This PDP contains one sequential activity and another PDP

Additionally, the WCET of all activities has to be determined to see how much each activity contributes to the overall WCET. This allows better optimization in Phase II. Furthermore, the analysis in Phase I should focus on computation intensive activities and avoid making a highly detailed analysis of insignificant program parts.

The result of Phase I is a platform independent APD with a high degree of parallelism, exposing dependencies and WCETs for the activities.

3.3 Phase II: Optimizing Parallelism

The goal of Phase II is to optimize the degree of parallelism for a good WCET. Its result is an optimized APD together with a list which variables have to be synchronized.

When implementing the APD resulting from Phase I directly, the cost for communication and synchronization would be very high. There would be several cores having

only a little bit of work and being idle most of the time. Therefore, in Phase II, some activities have to be joined to reduce the cost for communication and synchronization and to get a higher workload for the different cores. But not too much has to be joined as the overall result should still be a parallel program. Thus, a good trade-off between distributing the program over several cores and keeping the communication overhead low has to be found.

This can be seen as optimization problem. It has to take platform dependent parameters into account as well as the WCETs determined in Phase I and which global variables have to be synchronized when executing activities in parallel. Some examples are access times to core-local or global shared memory or the clock frequency and resulting execution time for program parts. We developed the *speedup approximation and parameter optimization tool*, employing a genetic algorithm to find an optimal configuration[9] with a minimized amount of shared variables to be synchronized and a good WCET[10] [30].

At the end of Phase II, the mapping of the activities to threads and cores must be tackled. Both mapping and placement are already well-researched, see e. g. [58] for a machine learning based method, [51] for mapping in combination with fault tolerance, and [42] for a mapping tool developed in the parMERASA project. On the parMERASA platform, we always map one thread to one core for better timing analyzability [44].

In the original pattern-supported parallelization approach [27,28], the result of Phase II is the parallel source code, but in our parallelization approach for hard real-time systems we decided to make the implementation in the new Phase III. Therefore, the result of Phase II is an optimized APD together with information which activities have to be mapped to which threads and cores and a list which shared variables have to be synchronized.

3.4 Phase III: Implementation

Contrary to the original approach by Jahr et al. [27,28], we add an additional Phase III for the implementation. In this phase, parallelism has to be implemented and synchronization has to be realized. For the former, the location of the corresponding PDPs has to be found in the source code and the sequential function calls must be rewritten to be executed in parallel. This can be facilitated with algorithmic skeletons, as described in the next paragraph. The synchronization of shared resources can be implemented by placing locks or rewriting code e. g. to utilize nonblocking data structures [23,24]. WCETs of the activities only played a role in Phase II to find an ideal configuration. New WCETs may be estimated after Phase III and refinement may be done to improve them.

[9] A configuration specifies which program parts should be executed in parallel, how many cores are utilized and which variables have to be synchronized.

[10] The tool is open source and can be downloaded at https://github.com/parmerasa-uau/parallelism-optimization.

Algorithmic skeletons are representing the same concept as PDPs. Patterns work on level of the model, while skeletons work on code level. An overview on existing skeleton libraries is given in [21]. We implemented a new, timing-analyzable skeleton library,[11] which is presented in [54], while a manual can be found in [30]. The usage of algorithmic skeleton libraries also enables reusing code for parallel code structures leading to a reduction of development and debugging efforts.

Additionally, we suggest generating a kind of program scaffolding. There, all calls to algorithmic skeletons are already placed in function bodies and the software engineer only has to insert his code based on the sequential implementation and the APD. Furthermore, code for the synchronization of global variables might be automatically generated, i. e. mutator functions (`get/set`) with locks around the variable accesses. The main remaining work would be to place mutator functions in the code when they were not utilized until now. This can be done quite fast with search-and-replace, but should be done carefully when e. g. a variable is used throughout a function and it is assumed that it does not change meanwhile. Then, it should be loaded at the beginning and stored at the end of the function.

With these methods, the implementation can be done semi-automatically. The final result is parallel code that can be executed on the target platform. When the timing analysis of the parallel program indicates that further refinement is necessary (e. g. because of differing synchronization overheads), then the parameters may be adapted and Phase II and III be done again like illustrated in Fig. 1.

4 Parallelization of a Signal Processing Application

In this section, we apply the parallelization approach for hard real-time systems on the core algorithm of a signal processing application from the avionics domain. Thereby, we want to make clear why WCET optimization is a complex issue and what the encountering problems are.

This section is structured as follows: first, we present the signal processing application in Sect. 4.1. Section 3.2, 3.3 and 3.4 correspond to the phases I to III, refinement takes place in Sect. 4.5. Finally, we analyzse the results in Sect. 4.6.

4.1 The Signal Processing Application

Our first parallelization takes a signal processing application, which is a core algorithm of an avionics application. We already presented some parallelization results for this application in [54]. Algorithms like this can often be found in embedded real-time systems for processing data which is captured with sensors. The structure of the application is illustrated in Fig. 5. It takes two sets of matrices as input (subsequently named `a` and `B`). They are characterized by the same size and number of included matrices and used for calculation of one output matrix. The program is executed iteratively, processing ongoing input sets. The first step of the computation is a FFT of

[11] Our Timing-analyzable Algorithmic Skeleton (TAS) library is open source (LGPLv3 licence) and can be downloaded at https://github.com/parmerasa-uau/tas.

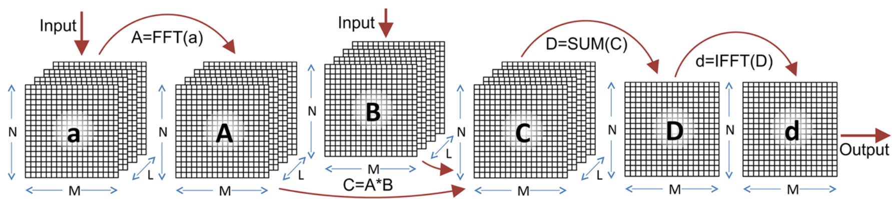

Fig. 5 Signal processing application

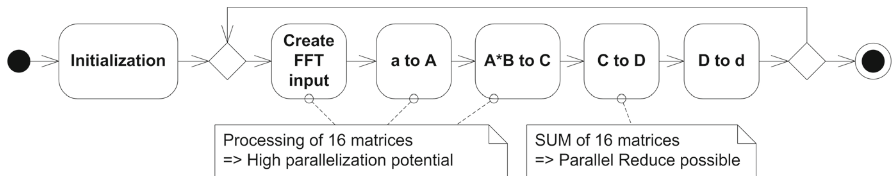

Fig. 6 AD of the sequential signal processing application

each matrix of `a`. The result (named `A`) is multiplied with `B` element by element. Afterwards, the calculated set of matrices `C` is summarized to one matrix `D` by computing the sum element by element. At last, the output matrix `d` is generated by calculating the inverse FFT.

In concrete, the matrix sets are instantiated with 16 matrices, where each consists of 128 rows and 128 columns.

4.2 Applying Phase I on the Signal Processing Application

The goal of the Phase I is to create an APD showing a high degree of parallelism, estimating the WCETs of its activities and determining dependencies between them.

Figure 6 illustrates an APD of the sequential application. It is already annotated where we see potential for parallel execution. Since the `Initialization` is only run once we do not further investigate it. The activities `create FFT input`, `a to A` and `A*B to C` process 16 matrices in an independent way. This may be done in parallel with up to 16 threads, each processing one matrix. `C to D` is a SUM operation over all matrices. It may be parallelized with a REDUCE operation.[12] `D to d` builds the inverse FFT of the resulting matrix of `C to D`, which cannot be parallelized.

Dependencies are only found between the activities: `a to A` takes the results of `create FFT input`, while `A*B to C` needs the results of `a to A` and so on. Therefore, dependencies are not an issue here.

We determined the WCET of these activities utilizing the static WCET analysis tool OTAWA [5]. It does an analysis of the binary and source code of the program without executing it. Thereby, it relies on a hardware model. In our case, the hardware is the

[12] Utilizing our PDPs, this can be realized with the Data Parallelism PDP, e.g. two threads, each doing SUM on half of the matrices. Finally, one thread would have to do the final SUM of the two resulting matrices.

Table 1 WCETs of the sequential activities

Activity	WCET (cycles)
Initialization	654,747,089
Create FFT input	4,187,336,815
a to A	47,853,059,229
A*B to C	109,620,927
C to D	98,621,517
D to d	2,983,266,489

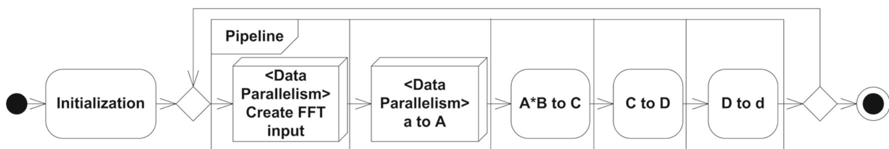

Fig. 7 APD of the sequential signal processing application with high degree of parallelism

parMERASA platform [43], which is an experimental multicore processor developed in the parMERASA project [56]. It is implemented in a cycle-accurate simulator based on SoCLib[13] [16]. The parMERASA platform is built in a way that facilitates WCET analysability, e. g. by ensuring deterministic behaviour and avoiding speculative components (since in the worst case it always has to be assumed that speculation fails). For our experiments, we have one core at the sequential version and up to 18 cores in the parallel versions. We assume the memory access latency to be 18 cycles.[14] Instruction cache works in a perfect way, while data caches are disabled for better analysability.

The WCETs from the sequential analysis are listed in Table 1: It can be seen that the highest contribution comes from `a to A`. `A*B to C` and `C to D` are very small—therefore, a parallelization will not bring any benefit.

From the sequential APD and following the WCET numbers, the APD in Fig. 7 was created. It differs from the proposals in Fig. 6 in the way that `A*B to C` and `C to D` are modeled as sequential activities and not as PDPs since they are only very small activities. Furthermore, a Pipeline PDP is placed over the five processing activities to increase the throughput. It enables parallel processing of data in five stages: while one set of input data is being processed in `a to A`, the next set of input data is being processed in `Create FFT input`. When the stages are finished, data is handed over to the next stage and processing of the next set of input data can start while the old one is still being processed in the subsequent stages. With the Pipeline PDP, processed results become available more often. Attention has to be paid on interferences: while

[13] Original Homepage: http://www.soclib.fr parMERASA simulator (open source under BSD licence): http://www.parmerasa.eu/files/open_source/soclib_parmerasa.zip.

[14] For more than 1 core, it always has to be assumed that all memory requests of all other cores are processed by the memory controller before the own request is handled. This results in *worst case memory access times* of 54 cycles for 4 cores, 96 cycles for 8 cores and 138 cycles for 12 cores.

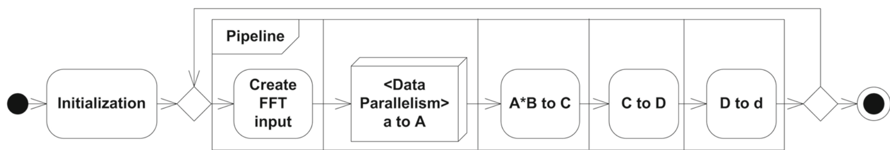

Fig. 8 APD of the optimized signal processing application with five pipeline stages and one data parallelism PDP at `a to A`

one stage is writing data, the next one might want to read it. Thus, double buffers should be employed.

4.3 Applying Phase II on the Signal Processing Application

Taking the APD from Phase I, there are three PDPs and up to 35 threads may be employed (16 in each of the stages `create FFT input` and `a to A` and three more for each of the remaining pipeline stages). However, due to overhead for thread creation, management and synchronization the best parallelization result will not be achieved by just executing everything in parallel. For our first parallelization szenario, we therefore only utilize the Pipeline PDP and the Data Parallelism PDP on `a to A`. Since `create FFT input` is only around one third larger than the largest non-divisible activity in the pipeline (`D to d`), we keep it sequential. Since we assume a high overhead when `a to A` is executed by 16 threads, we only utilize 4 threads here. The resulting APD is illustrated in Fig. 8.

Altogether, there are eight parallel threads which are placed on eight cores (we always apply a 1:1 mapping). This parallelization scenario was already implemented in [54]—however, there only observed execution times (OETs) and their speedups were determined, no WCET analysis took place. The OET speedups were 4.6 for a matrix set of $16 \times 16 \times 4$ (rows $\times$ columns $\times$ matrices) and 5.2 for one with $32 \times 32 \times 8$ matrices. They show that a high parallel performance is possible.

4.4 Applying Phase III on the Signal Processing Application

The implementation is a simple task due to the small program, simple dependencies and the availability of skeletons. Details on the parallelization implementation utilizing algorithmic skeletons can be found in Sect. 8 and in [30,54]. The results of the static WCET analysis done with OTAWA can be seen in Table 2.

For easier comparison, we added the WCETs of the sequential program from Table 1 as well as WCET speedups. Though, all speedups are slowdowns, even at those activities which are executed in a *sequential* way. This is caused by the worst case memory access times which have to be assumed when several cores share one global memory. At a memory intensive application like our signal processing application they lead to a high overestimation (the WCET gets higher than what is realistic). It is a current research field how to do and improve WCET estimation for parallel applications, see e. g. [22,41,49].

Table 2 WCETs of the different activities

Activity	Par. WCET (cycles)	Seq. WCET (cycles)	WCET speedup
Initialization	2,752,714,141	654,747,089	0.23
Create FFT input	19,299,113,419	4,187,336,815	0.22
a to A	60,509,156,473	47,853,059,229	0.79
A*B to C	357,235,281	109,620,927	0.31
C to D	319,769,223	98,621,517	0.31
D to d	15,093,890,823	2,983,266,489	0.20

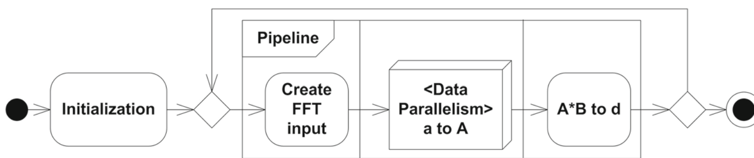

Fig. 9 APD of the optimized signal processing application with three stage pipeline and one data parallelism

Since the first parallelization szenario brings no improvement at all, we have to do some refinement.

4.5 Applying Refinement on the Signal Processing Application

Worst case memory access times depend on the number of utilized cores. Therefore, we focus on reducing the number of cores in the second parallelization scenario: `A*B to C`, `C to D` and `D to d` are grouped together to form one single activity `A*B to d`, the pipeline now has three stages. Altogether, six cores are utilized: four for the data parallelism `a to A` and two for the remaining pipeline stages. The corresponding APD is illustrated in Fig. 9, the results of the WCET analysis can be seen in the second column of Table 3.

Now it takes 47.9 billion cycles to get new results out of the pipeline,[15] which is a small improvement compared to the sequential version with 55.5 billion cycles.[16] Since the pipeline is unbalanced and there is more parallelization potential in its largest activity `a to A`, we try to distribute this activity on 8 and 16 cores. The WCET results are also shown in Table 3.

Comparing these results, the 10 core version (8 cores for the data parallel execution of `a to A`) reaches the best result with generating one result matrix every 37.0 billion cycles (WCET speedup of 1.5). In the 18 core version, `a to A`'s WCET is even lower, but the sequential activity `Create FFT input` slows down and dominates

[15] At the pipeline PDP, data is moved to the next stage when all stages have finished their work. Therefore, every time the largest stage is finished, one result matrix comes out of the pipeline.

[16] In the sequential version, all activities have to be processed to get one result matrix. Then the next set of input data can be processed.

Table 3 WCETs of the parallel implementations with 3 stage pipeline

Activity	WCET 6 cores	WCET 10 cores	WCET 18 cores
Initialization	2,214,774,261	3,344,455,009	5,550,036,165
Create FFT input	15,424,241,353	23,561,415,750	39,448,178,979
a to A	47,905,183,017	36,967,526,805	31,190,970,443
A*B to d	13,216,052,757	20,354,093,164	34,292,417,690

the pipeline's WCET. Therefore, a new result matrix becomes available every 39.5 billion cycles (speedup of 1.4).

4.6 Analysis of the Results

A WCET speedup of 1.5 on 10 cores is low and we have to analyzse the reasons. In [54], an OET speedup of up to 5.2 was reached with 8 cores. When only evaluating the FFT computation `a to A`, an OET speedup of 3.9 (4 cores), 7.8 (8 cores) and 15.6 (16 cores) is possible, while the WCET speedup is only 1.0 (4 cores), 1.3 (8 cores) and 1.5 (16 cores).

But where is the problem of achieving WCET speedups with static WCET analysis? It lies mainly at the worst case memory access times as well as synchronization. The latter are already a challenge on single-core systems, e. g. when interrupts or scheduling has to be respected. But on multi-core architectures everything is more complex with several cores sharing resources like memory and having to assume that all other cores will have access before the own access is processed. At the development of WCET aware software, a lot of restrictions apply also, see e. g. [9,17].

At the sequential parts of our program, the OETs mostly stay the same, but the WCETs rise when utilizing more cores. Figure 10 illustrates how the WCETs of sequential and parallel program parts change with different core numbers.

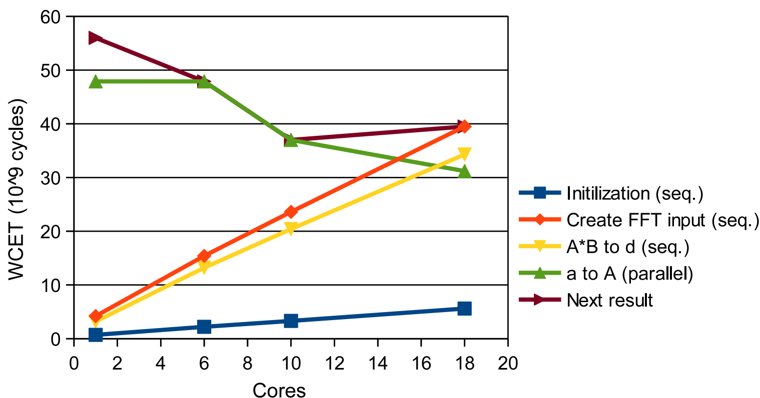

Fig. 10 Relation of WCETs of sequential and parallel activities with rising core numbers. Sequential WCETs go up linearly, the WCET of the parallel activity `a to A` goes down slowly (Color figure online)

We draw three conclusions from the figure:

1. The WCET of sequential functions increases with the number of cores. This is a specific problem of worst case estimations, not present in general-purpose parallelizations. As long as memory interferences have to be assumed, this is a strongly limiting factor for WCET-aware parallelization.
2. We added a curve `Next result`. It illustrates how long it takes until a new result matrix becomes available. In the sequential version, sensor inputs have to go through all activities before the next input set can be processed. Therefore, a new result matrix is available after 55.5 billion cycles. In the parallel versions applying a 3 stage pipeline PDP and the data parallel PDP for `a to A` activity (cf. Fig. 9), the pipeline needs a few iterations to get running and then produces new result matrices every time the data moves on in the pipeline. Since data is moved on every time all stages are finished, this time is determined by the maximum of all pipeline stages, i. e. new result matrices become available after 37,0 billion cycles in the 10 core version and after 39.5 billion cycles in the 18 core version.
3. Although the parallelization of `a to A` leads to a better WCET speedup for this activity when utilizing 18 instead of 10 cores, the 10 core version has a better overall WCET speedup. This is due to the activity `Create FFT input` which becomes the dominating activity in the pipeline at the 18 core version. It remains sequential, but its WCET increases over that of `a to A` due to the worst case memory access times.

In conclusion, a parallelization for WCET-aware real-time applications not only has to estimate which program parts may get faster when executing them in parallel, but also has to take into account that additional cores lead to a higher WCET in sequential program parts. Therefore, a good trade-off between sequential and parallel program parts has to be found. Parallelization is only reasonable when its effects are stronger than the slowdown caused by synchronization and worst case memory access times of additional cores.

5 Control Program of BAUER MC 128

The second industrial application that we apply our parallelization approach for hard real-time systems on is the control program of the foundation crane BAUER MC 128. In this section, we give a short overview on how the program is integrated in the machine: first, Sect. 5.1 describes the hardware in the machine, then the software architecture of the single-core implementation is explained in Sect. 5.2, while Sect. 5.3 characterizes the execution behaviour of the application.

5.1 Hardware Overview

The driver of the machine controls it primarily through a screen, two joysticks and some control keypads. In addition, there are several pedals for moving the machine. Also, there are security devices, such as the so-called “armrest switch” that must be

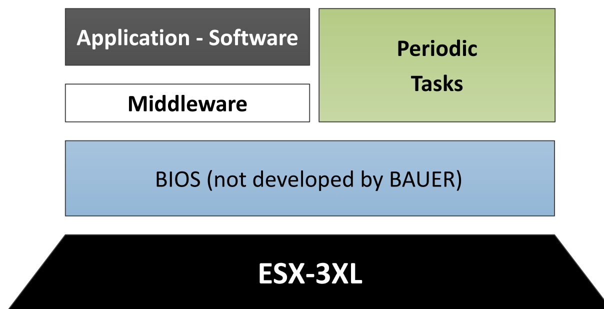

Fig. 11 The architecture of the ECU of the MC 128: at the bottom there is the hardware (ESX-3XL) together with the BIOS, which is the basic software provided by the ECU manufacturer. Hereon the middleware, which is developed by BAUER, is placed. It contains generic functions for all machines from BAUER together with periodic tasks e. g. for CAN communication. On top there is the application program specific for the machine together with its periodic tasks

closed when the driver is sitting in the cab, thus indicating that the crane is to be operated.[17]

In the MC Series, an embedded real-time-capable electronic control unit (ECU), ESX-3XL by Sensor-Technik Wiedemann [52], is deployed. It features a single-core Infineon TriCore processor, 4 CAN interfaces, and a high number of input and output ports. Functions provided by the software running on this ECU are, e. g. the manual and automatic control of winches, the ability to turn and move the machine, and also control of multiple pumps since the foundation crane is operated hydraulically.

5.2 Single-Core Software Architecture

The software on the ECU comprises three layers, which can be seen in Fig. 11 and are built on top of the hardware (ESX-3XL). APIs are defined between them:

At the lowest level is the so-called *BIOS* (closed-source by the ECU manufacturer). It is equipped with a real-time scheduler, which can execute periodic tasks at different priorities and frequencies interrupting the application program. These tasks are implemented in other layers of the software and e. g. send and receive CAN bus messages, read input values from sensors and set output values to actuators.

The *middleware* by BAUER is re-used over all the companies' machines and provides shared functionalities abstracting from the BIOS. This includes drivers for the interfaces and sensors in the foundation crane (e. g. keyboards and joysticks, inclinometers for boom orientation). Also, the middleware implements multiple periodic tasks, e. g. for the CAN busses.

The actual control of the machine takes place in the *application program*. It is written specifically for one type of machine (here: foundation cranes). In the source code of the crane, all possible procedures (e. g. drilling, milling, BDC, etc.) are implemented, but only unlocked upon acquisition of the necessary equipment. The application program

[17] Many components check the armrest switch because for security reasons they are not allowed to run when there is no driver sitting in the cab.

consists of initialization code, a main control loop (in the remainder: main loop) and a number of periodic tasks. There are two types of the latter: a part of the periodic tasks is executed by the scheduler in the BIOS, which interrupts the main loop for their execution. Some more periodic tasks are located directly in the main loop, where they are polling how much time has passed since the last call and are executed only if enough time has passed.

5.3 Execution Cycle

The control program is basically executed in two phases: (i) the initialization code is executed once after power-up. All outputs are assigned valid values and the configuration is loaded from the EEPROM. (ii) Then, the main loop is executed concurrently to the scheduler invoking the periodic tasks. Hence, the scheduler interrupts the main loop from time to time (in the single-core implementation).[18] Of course this can be prohibited for short periods for the sake of data consistency by disabling interrupts.

6 Phase I: Revealing Parallelism at the Foundation Crane

As described in Sect. 3.2, we start with the single-core legacy program and get an APD showing a high degree of parallelism in this phase. Furthermore, we estimate WCETs of the sequential activities and determine shared ressources, i. e. variables. First, we describe the methodology how we analyze the legacy program (Sect. 6.1), then the concepts found in the source code (Sect. 6.2). Section 6.3 shows the results of the analysis together with a sample APD, while Sect. 6.4 handles periodic tasks.

6.1 Methodology of the Code Analysis

The foundation crane control application is analyzed top-down by a software engineer without domain knowledge.

In the beginning, we start with the legacy program (`main` at code level), represented by a single activity in the APD as can be seen in the top of Fig. 12. By increasing the level of details at the activity `Foundation Crane` and applying the "Task Parallelism" PDP at the `main_loop`, the bottom part of Fig. 12 can be constructed. `main_loop` contains nine activities which are independent[19] from each other since each of them addresses a different part of the machine.

The search for situations where patterns could be applied had to be done manually because there are currently no tools available. However, for collecting dependencies, several tools can be employed: we used CScope,[20] an open-source static code analysis

18 Interrupts take place every 1 ms because this is the smallest period of periodic tasks.

19 They have a high degree of independence–however, the control application of the foundation crane contains no components which are completely independent of all others since they all share the same data structures, e. g. for accessing sensors and actuators.

20 Homepage: www.cscope.sf.net.

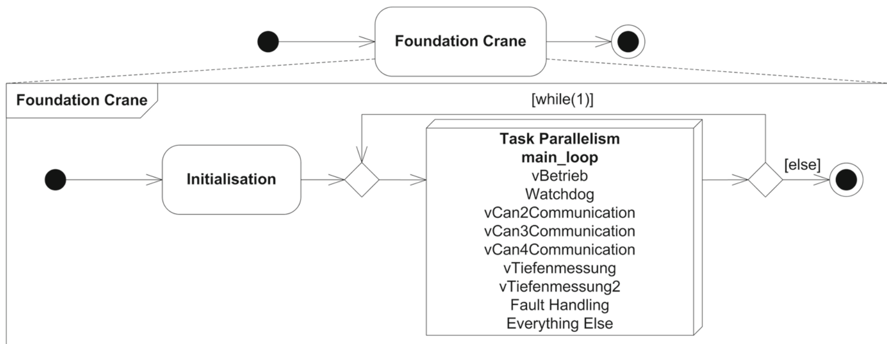

Fig. 12 The first APD of the analysis: one activity with one level of more details. There is an initialization and a `main_loop` which is executed forever. Pins are ommited to keep the diagram compact

tool and the Rapita Dependency Tool [48]. Additionally, we implemented a custom tool to make lists of all shared variables used by a function and all functions called by it. This enables us to see all shared variables used by an activity. In Phase II, we employ this information to minimize the number of variables to be synchronized.

An activity with all of its shared variables can be seen in Fig. 13. Blue/italic names denote shared variables that were directly found in the function called `vfahrwerk_links`, while the black ones are found in its subfunctions, i. e. functions called by `vfahrwerk_links` or their subfunctions. Beyond Fig. 13, pins representing shared resources are always ommited to keep the APDs compact.

The analysis takes place in an iterative way: each time the level of details is rised or patterns are applied, we get a new APD. For the new activites, a measurement-based timing analysis is done with RapiTime [47], which is a commercial tool and part of the Rapita Verification Suite (RVS).[21] We use RapiTime on the original single-core platform to estimate WCETs, because we did not have a detailed timing model of the TriCore processor for a static timing analysis. Each WCET is noted down and for the further analysis we focus on activities with a high WCET. Hence, we do not waste time analysing insignificant parts of the program. The smallest resulting activities have a size of around 20,000 cycles. For larger activities, we go on with the analysis. We stop examining an activity, when it is small enough, we reach long `if-then-else` branches, state machines, library calls or other code which seems not to be apportionable anymore.

6.2 Concepts Found in the Source Code

The software of the foundation crane originates from another BAUER machine. Therefore, its oldest parts are almost 20 years old. It has grown over time just like most industrial applications do.

The following main coding principles were discovered in the software during the analysis of the source code:

[21] Homepage: www.rapitasystems.com.

Fig. 13 The activity `vfahrwerk_links` together with all shared variables found. The *blue* ones are called directly by the function `vfahrwerk_links`, the *black* ones by functions called by it. On the input pin, all variables which are only read are listed, on the output pin all variables which are only written to and on the resource pin the variables which are read from and written to appear (Color figure online)

– Many options (e. g. special procedures of the machine like drilling, built-in sensors) are switched on/off by `#define` tags. Thus, already at compile time, a lot of source code not relevant for the target machine is omitted reducing the binary size. On the other hand, this makes it harder to understand the code and find parallelization candidates.
– Functions typically control defined parts of the machine or provide functionality of a certain scope.
– The code is control intensive and not data intensive; there are no large loops but many conditions on parameters like the series of the machine, configuration parameters, or sensor values.
– Software communicates via shared memory, e. g. sensor values are written by one function to a shared global data structure and other functions read data from this structure instead of directly accessing sensors. This is applied for sending/receiving CAN bus messages, too.
– More recently added code parts employ mutator functions (like `get/set`) to access shared global variables, older parts access them directly.
– State machines can be found, e. g. for the dynamic soil compaction application: the weight can (a) be moved manually, it is (b) lifted automatically, or it is (c) in free-fall.

- Time slicing is sometimes used, i. e. time-consuming tasks are not processed in ‘one shot’ but with each call only a part is executed. This is done because all parts together would delay other code parts for too long.
- Several periodic tasks are not invoked by the scheduler but are implemented in the main loop and monitor system time waiting for a fixed time interval to have passed since the last execution. Since the main loop is interrupted for the execution of other tasks by the scheduler this might result in higher jitter for these “periodic” tasks implemented in the main loop.

6.3 Results of the Code Analysis

After several iterations, we stopped the analysis. There were PDPs for Task Parallelism, Periodic Task Parallelism, Data Parallelism and Pipelining available in the parMERASA pattern catalogue (see Sect. 1.1 or [18]). In the control application of the foundation crane, only the PDPs Task Parallelism and Periodic Task Parallelism could be applied.

We found periodic tasks from the scheduler and some in the main loop. For them, the parallelization approach is not applied. They are put on dedicated cores, see details in Sect. 6.4. We also leave away the initialization in the parallelization approach because it is only run one time for a few seconds, while the machine is then operating for several hours.

Some parts of the control program seemed like candidates for data parallelism (e. g. there are two winches that are always steered the same way), but their implementations differ in some details and there are two control levers connected physically. Therefore, it is better to leave these as separate activities and group them with a Task Parallelism PDP.

In the remaining control program, the PDP Task Parallelism could be found 12 times and a maximum of 61 activities could be executed in parallel. The smallest ones have a size of around 20,000 cycles, while the largest activity needs around 650,000 cycles. We analyzsed it, but breaking it into several activities seemed to be difficult, because it contains a big state machine and there are lots of case distinctions. Also, a lot of sensor values are evaluated and the machine reacts on them by regulating actuators.

For execution on dedicated cores, activities should have a size of at least 150,000 cycles because of the arising overhead for organizing the execution on another core and waiting at barriers etc. Therefore, the smallest activities may be grouped together to be executed on another core as one large activity. To find an efficient trade-off between runtime and size of activities and resulting overheads is task of Phase II.

Different parts of the code control different parts of the machine and are therefore quite independent from each other. However, some resources are shared although machine parts are independent from each other, e. g. the interface for sensors and actuators. There are also some security features which check whether the machine is allowed to operate and therefore result in shared variables and data structures. Altogether, there were around 650 global variables. They limit parallelism since there always remain dependencies. The potential will be investigated in Phase II.

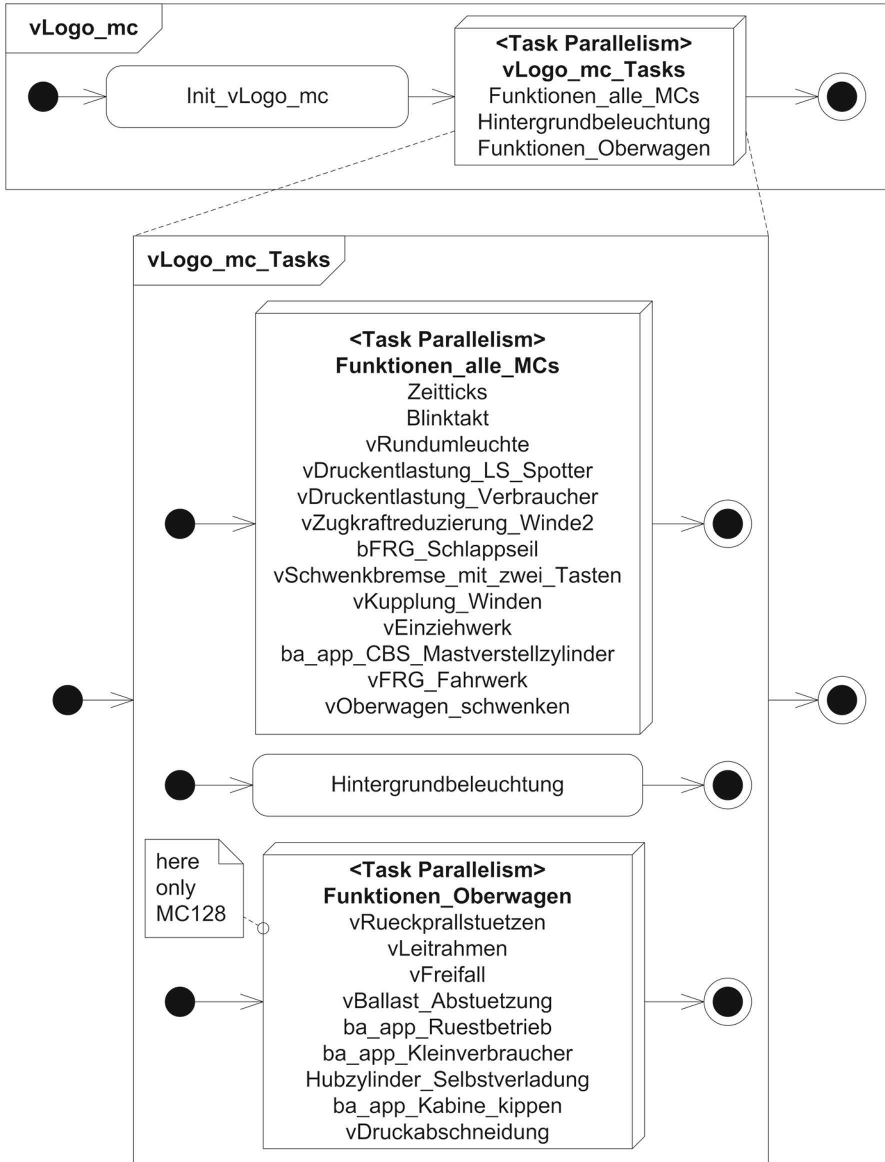

Fig. 14 APD of one part of the control program: there are a lot of functions that may be executed in parallel. Each of them is controlling one component of the foundation crane

One of the APDs resulting in Phase I can be seen in Fig. 14. The activity `vLogo_mc` contains an initialization activity and functions for machine control: `Funktionen_alle_MCs` represents code for all machines of the MC series (foundation cranes), `Hintergrundbeleuchtung` works for all machines (not only foundation cranes, but also e. g. drilling rigs) and `Funktionen_Oberwagen` only

for the MC 128. The first and last ones contain activities that can also run in parallel. Each of these activities is controlling one component of the machine. Most components operate independently from each other. Therefore, it is possible to execute them in parallel. The WCETs are not shown, but most of these functions need around 40,000 to 120,000 cycles. Hence, it might not be efficient to run all of them in parallel, because the arising overhead is too large. In Phase II, the parallelization result for employing two cores to run `vLogo_mc` can be seen (see Fig. 16 there).

6.4 Scheduling of Periodic Tasks

In the single-core version of the control code, periodic tasks are executed by a scheduler provided by the ECU supplier. The execution of periodic tasks interrupts the execution of the main loop and can also interrupt periodic tasks of lower priority. This leads to jitter in the execution times of periodic tasks and the execution time necessary for an iteration of the main loop. To overcome this and to leverage timing analysis, dedicated cores are reserved for the execution of periodic tasks in a simple *static cyclic schedule* [4].

Table 4 shows an accumulated overview over 23 periodic tasks found in the software; 8 were previously defined for the scheduler and 15 are carved out from the main loop. Besides the period, which is between 1 and 1000 ms, we measured the maximum OETs of the tasks on the TriCore platform,[22] which are the basis for further considerations from a conservative point of view.[23]

The most extreme load will be every 3000 ms, when all tasks have to be executed. Executing all tasks together results in an accumulated OET of 0.978 ms to be completed in a 1 ms interval (then the next round of tasks with 1 ms period has to be executed). Because additional latencies should be foreseen for the scheduler itself and calling tasks as well as synchronization costs, it seems appropriate to dedicate two cores for the execution of periodic tasks: one core can execute tasks with 1 ms period, the second core executes all other tasks.

With a more sophisticated scheduling algorithm (see e. g. [50]), periodic tasks could also comprise multi-threaded parallel code. Because this is not necessary for the evaluated software of the foundation crane we do not go into details here.

7 Phase II: Optimizing Parallelism at the Foundation Crane

The result of Phase I are APDs, together with WCETs and dependencies. Goal of Phase II are optimized APDs, which are to be realized in code in Phase III. In Sect. 7.1,

[22] Unfortunately, it was not possible to determine WCETs for tasks in the scheduler since RapiTime supports only analyzing functions in the program flow and OTAWA did not work on the TriCore platform because no detailed timing model is available.

[23] Nevertheless, we are aware that the OETs may be different on the target platform. Our results show that two cores are nearly filled by the periodic tasks now because of lower clock frequency and synchronization overheads.

Table 4 Accumulated list of periodic tasks from the scheduler and found in the main loop with periods (ms) and OETs (ms)

	Period (ms)	$\sum$ OET (ms)
5 tasks	1	0.7170
2 tasks	2	0.1540
1 task	3	0.0083
7 tasks	10	0.0091
1 task	20	0.0590
6 tasks	100	0.0159
1 task	1000	0.0149
Overall OET sum	every 3000	0.9782

the methodology how we optimize APDs is described and in Sect. 7.2 the results are presented. Finally, we choose two scenarios to be implemented in Sect. 7.3.

7.1 Methodology to Optimize the APD

The activities should be placed on the cores in an "efficient" way. This means getting a WCET speedup while trying to employ as few cores as necessary. Thus, the synchronization overhead and idle times have to be kept low.

With a look on the APD, it has to be decided which activities should run sequentially and which ones in parallel. Therefore, different *configurations* have to be evaluated. A configuration specifies:

- which patterns should be enabled, i. e. program parts should run in parallel
- how many cores should be used for which pattern
- which and how many variables have to be synchronized

We developed a speedup approximation and parameter optimization tool[24] to find and evaluate different configurations, which is described in [30]. It takes the APD as XML file and a list of shared variables, which was generated by our custom tool in Phase I as input. Then, it applies a genetic algorithm to generate configurations with a minimum number of used cores, shared variables and approximated WCET. However, the tool does not take slowdowns of sequential activities into account when more cores are utilized, since it should also work for applications without real-time requirements.

7.2 Results of the Model-Based Optimization

The results of the optimization with our tool originally appeared in [26] and can be seen in Fig. 15. As already stated in Sect. 6.3, the diagrams only show possible configurations for the main loop and ignore periodic tasks. Figure 15a shows the relation between the number of employed cores and the approximated execution time. Every X represents one configuration, while $\diamond$ denote the optimal configurations at

[24] Download link: https://github.com/parmerasa-uau/parallelism-optimization/.

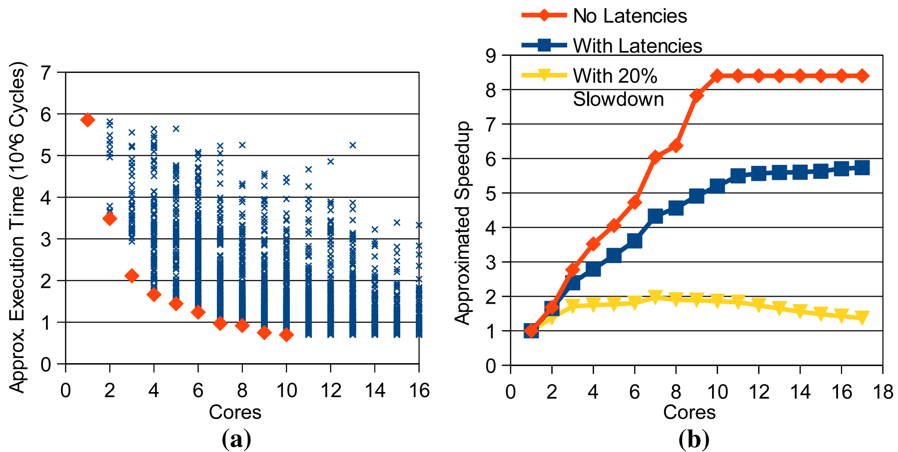

Fig. 15 Evaluation results from multi-objective optimization of the model of the main loop. They were originally published in [26]. In the *right figure*, we added a curve `With 20 % Slowdown` following the results of the parallelization of the avionics application, **a** optimization for minimal approximated execution time and minimal number of cores (cut at 16). *Times* are all evaluated configurations, *diamond* are optimal configurations representing the Pareto front, **b** Approximated optimal speedups for different numbers of cores with/without parallelization overheads and with assuming a slowdown of 20 % for each utilized core

the Pareto front. Configurations with more than ten cores do not reach an additional speedup. This is due to Amdahl's law [3]: the maximum improvement (here: maximum speedup or minimum approximated execution time) is limited by the time needed for the sequential part of the program (in our case: the largest activity).

Another limitation are the parallelization and synchronization overheads. We assume them to be static 25 %. The impact of these overheads can be seen in Fig. 15b: it shows the approximated speedups for the configurations at the Pareto front with and without latencies. While the speedup without latencies is above 8, it gets stuck at around 5.5 when taking them into account.

Furthermore, we added a curve `With 20 % Slowdown`, which was not present in [26]. It respects the worst case assumptions for memory interferences, which are not respected in the latencies for synchronization. Following the results of the parallelization of the avionics application, we know that each additional core utilized slows down the WCET even for sequential activities. Therefore, we assume a *WCET Slowdown Factor* to increase the WCETs by 20 % for each core which is utilized. This means the estimated WCET is multiplied by 1.2 for 2 cores, 1.4 for 3 cores etc. At the avionics application, the WCET slowdown factor was around 50–60 %. Here, we assume it to be lower since the application is control-intensive and not data-intensive. But it illustrates that it might not be efficient to employ more than a few cores.

7.3 Choice of APDs to be Implemented

We chose to implement the configurations with 2 and 6 cores lying on the Pareto front. The first one should provide a reasonable performance improvement while only

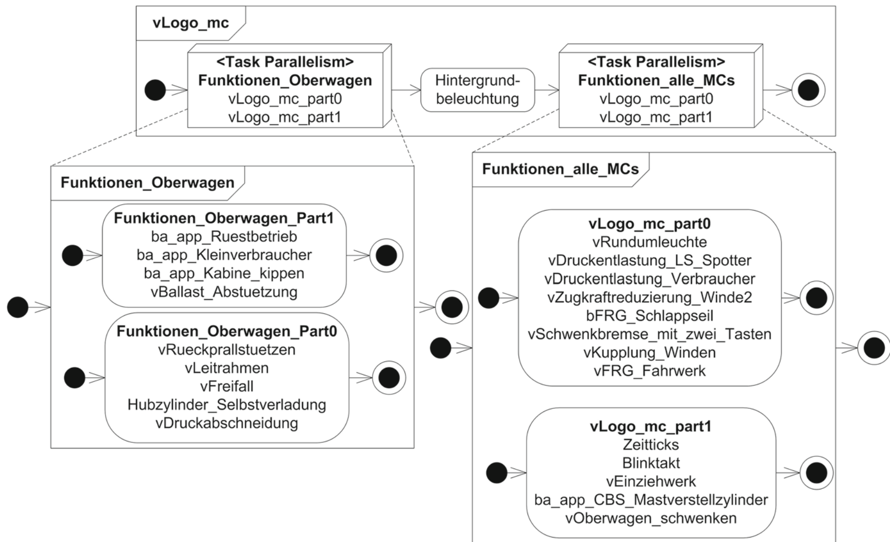

Fig. 16 Optimized version of the APD from Fig. 14. It can be executed on two cores, the "small" activities are grouped together

needing few cores. It should be kept in mind that two additional cores are utilized for the periodic tasks. Thus, the 2 core implementation will need 4 cores altogether.

The 6 core implementation should show the scalability trend. When it reveals that the 20 % slowdown was too pessimistic, implementations with more cores might be beneficial. Otherwise, the 2 core implementation might already be a configuration with a good trade-off.

One part of the optimized APD for 2 cores is illustrated in Fig. 16. It shows the same activity as in Fig. 14, `vLogo_mc`. The execution starts with the activity `Funktionen_Oberwagen`, which is executed by two cores (`part0` by one of them and `part1` by the other one), then one of them executes the activity `Hintergrundbeleuchtung`, while the other one is idle and finally both cores execute the two parts of `Funktionen_alle_MCs` in parallel. The size of the activities in `Funktionen_Oberwagen` (`part0` and `part1`) is quite similar and the same holds for the parts of `Funktionen_alle_MCs`. This way, the best speedup can be achieved, because both cores are working for most of the time.

8 Phase III: Implementation at the Foundation Crane

The result of Phase II is an optimized APD, which now has to be translated into code. The employed time-predictable target platform [56] provides a programming API, where two synchronization primitives as described by Gerdes et al. [18,19] are suitable for process and progress coordination: F&I-barriers[25] and ticket locks. Both

[25] Fetch and increment barriers.

are specifically designed for worst case performance by providing fairness between hard real-time threads.

To realize parallelism as described and optimized in the model of the main loop in Phase II it is necessary to implement the PDPs (Sect. 8.1) and to synchronize accesses to shared resources (Sect. 8.2).

8.1 Implementation of Parallel Design Patterns

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

For the C programming language and with PThreads-like threading no Algorithmic Skeleton Library was available (cf. [21]). Therefore, we developed a new skeleton library called *Timing-analyzable Algorithmic skeletons (TAS)*, which implements patterns from the parMERASA pattern catalogue [18]. It is described in [54], a manual can be found in [30]. The skeleton implementation was done according to known guidelines for timing analyzable code [9,17]. 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 [8] or other kinds of dynamicity.

The main work effort here was to locate the position of PDPs in code and place calls to the skeleton library. Furthermore, a mapping of the tasks to cores had to be done, but this was an easy task since all cores share the same global memory with equal access times and we apply a 1:1 mapping (one thread on one core).

8.2 Synchronization of Accesses to Shared Resources

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

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

To ease the adaption of the source code with respect to synchronizing accesses to shared variables the use of mutator methods (like `get/set`) is preferred wherever possible. With a custom code generator the source code of the mutator functions including synchronization as well as other code fragments are produced for each shared

variable. They can be placed where they are necessary or might be encapsulated as dedicated header file.

It is obvious that mutator methods can only be used for atomic accesses to shared variables. If multiple variables should be updated only in one “transaction” because of a kind of inherent relationship, e. g. a sensor value and the exact time of its measurement, then (unchanged) mutator methods cannot be employed. Also, if lots of operations have to be done on a larger data structure, then atomic locks are also not the appropriate way because of their impact on the performance. Instead, a lock should be acquired before all the operations and released afterwards.[26]

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

In the middleware (see Sect. 5.2) of the foundation crane control code all synchronization was done with directly placing locks into source code. This was appropriate because most variables were only present in a single file, so only one file had to be modified for each variable. If locks have to be nested then the order of locks has to be consistent throughout all code files – else deadlocks can occur easily.

The application was completely instrumented with mutator methods for accessing global shared variables. Situations as described above conflicting with mutator methods were not found. If a variable is read multiple times in a function then it has to be decided whether a local copy should be kept or if always a new value is fetched. Resolving this can be cumbersome and require domain knowledge; neglecting this can lead to race conditions.

9 Evaluation of the Parallelized Foundation Crane Control Code

After following all three phases of the parallelization approach, we have a parallel implementation which has to be evaluated by estimating WCETs and comparing them. In Sect. 9.1, we describe the setting of software and hardware for the evaluation, while the methodology is topic of Sect. 9.2. Section 9.3 presents and analyzes results, which are compared to results of other researches in Sect. 9.4.

9.1 Setup of Software and Hardware

Like in Sect. 4, the target platform is the parMERASA processor [43,56]. We utilize 4 or 8 cores, all cores are connected to a single global shared memory with a latency of 18 cycles. Since it always has to be assumed that all cores interfere when accessing shared memory, worst case memory access times have to be assumed: they are 54 cycles for 4 cores and 96 cycles for 8 cores. Memory accesses are organized hierarchically by utilizing a private first level cache for each core. Thereby, we assume a perfect

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

Table 5 Configurations of the main loop

	1 core	4 cores	8 cores
Cores for main loop	1	2	6
Cores for periodic tasks	1 (same as above)	2	2
Active PDPs	0	5	4
Synchronized variables	0	82	104
Worst case memory access time (cycles)	18	54	96

instruction cache and a data cache with LRU replacement policy for non-shared data, while shared data is not cached.

In the first parallelization scenario, 2 cores are reserved for the execution of periodic tasks, another 2 cores are assigned to the execution of the main loop. Being a time-predictable research processor, there is no dynamic scheduling of threads. Hence, running the main loop with 2 threads means that 2 cores have to be reserved for it (static thread mapping—1 thread is mapped to 1 core).

Additionally, we implemented a version with 8 cores to see if our slowdown assumption from Sect. 7.2 is correct or if there is a better scalability. Again, 2 cores are utilized for the periodic tasks. Here, for the main loop 6 cores are employed.

It does not make sense to compare the sequential version from the original platform with the parallel versions on the parallel platform, because of different architectures (cache sizes, memory latencies, etc.). Therefore, we also had to adapt the sequential version to run on the parallel platform to be able to calculate a speedup.

An overview of the different versions can be seen in Table 5. It also shows how many PDPs (of maximum 12) are enabled, how many variables had to be synchronized and which times have to be assumed for memory accesses.

9.2 Evaluation Methodology

To determine the WCET bounds, we use the static WCET analysis tool OTAWA[27] [5], which analyzes the binary file. The analysis is time consuming since part of it has to be performed manually. Annotations in the source code can ease it by describing interactions between different threads, e. g. which code parts require the same lock or which threads are waiting at a barrier. For parallel parts that are realized with our Timing-analyzable Algorithmic skeletons (TAS) [54], we have written a tool to generate annotations for OTAWA.[28] It is described in [30]. There comes no overhead from the static WCET analysis, however, pessimistic assumptions e. g. for memory accesses lead to a WCET estimation which is higher than the real WCET.[29]

[27] Available as open-source software: http://www.otawa.fr.

[28] Our tool can be downloaded at https://www.github.com/parmerasa-uau/tas2otawa.

[29] The real WCET cannot be estimated, only a safe upper bound, see [59].

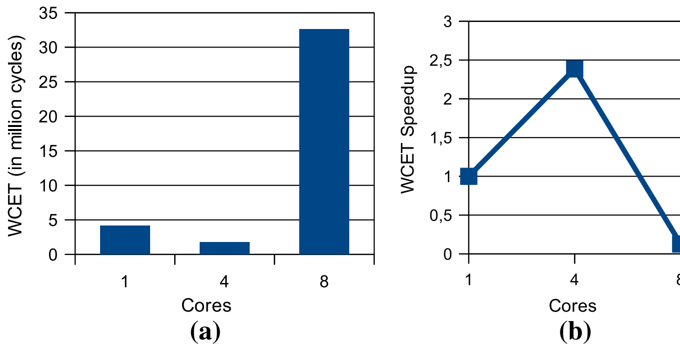

Fig. 17 Evaluation results: static WCETs and WCET speedup, **a** WCET for 1, 4 and 8 cores. At the 4 core version, the WCET falls to around 40 % of the sequential version, while it rises to a multiple at the 8 core version, **b** WCET Speedup for 1, 4 und 8 cores. At 4 cores, the speedup reaches around 2.4, while at 8 cores a slowdown to around 0.15 shows up

We were verifying the correctness of the parallel implementation by comparing the outputs of the program, especially the sent CAN messages, and utilizing RapiCheck [48]. This did not reveal any issues.

9.3 Results and Their Interpretation

Figure 17 shows the estimated WCETs in million cycles and the achieved WCET speedups. As can be seen in Fig. 17a, a WCET of 4.1 million cycles was estimated for the sequential version, while it was around 1.7 million cycles at the 4 core version and 32.2 million cycles at the 8 core version. In Fig. 17b, the according speedups can be seen. The 4 core version shows that our approach works: a WCET speedup of 2.39 is achieved. At the 8 core version, it gets clear that limits are reached: a WCET slowdown of 0.15 occurs. Configurations with more cores lead to results even worse than the 8 core version. Therefore, we analyzed the application to find reasons for the bad scalability and found them in the structure of the application (Sect. 9.3.1) as well as the arising contention (Sect. 9.3.2).

9.3.1 Limits Imposed by the Application Structure

The application structure revealed the following: first, there are several large non-parallelizable parts which suffer from the worst case memory access time slowdown and limit the overall parallelization potential due to Ahmdahl's law [3].

Furthermore, 650 global shared variables in an application with a few 10,000 lines of code are a signal for too many dependencies. Therefore, a refactoring may be necessary to achieve better results. Kempf et al. [32] come to the same conclusion while analyzing the limits of their automatic parallelization. During our parallelization of the foundation crane application, we focused on changes following the parallelization approach and did not do any further refactoring.

The application is control-intensive, no possibilities for data parallelism could be found. Therefore, it is not optimal for parallelization. In Sect. 9.4 we compare our results with parallelization results of other researches, who come to similar results up to four cores and mostly did not investigate more cores. Also, it is hard to get good parallelization results with control-intensive code. It is even harder when parallel performance has to outperform the slowdown caused by higher memory access times when employing additional cores.

Finally, it should be noted that the foundation crane application is not a benchmark or a small part of an application which is well suited for parallelization. It is the complete control program of a construction machine. As such, it is the most complex application we ever made a parallelization for and a WCET analysis of. We see the limits that modules of the application are coupled very closely – there are no completely independent parts. Therefore, synchronization is always needed and attention has to be spent on race conditions and if control code of different machine parts can really be executed at the same time.

9.3.2 Limits Imposed by Contention

This leads to analyzing reasons for contention: Several components of the machine exist more than once and have similar, but not identical code to control them, e. g. there are several hydraulical pumps or several rope winches. Since the machine parts are different, these functions are parallelization candidates, but they share quite a lot of global variables and therefore interfere with each other. Additionally, several data structures are utilized almost everywhere in the application (e. g. those for sensors and actuators) and thus each parallel access adds to the worst case waiting times.

Another issue is a periodic task updating sensor and actuator values. It has a very short period and uses locks during each update. This task might be the central point to reduce contention. However, making changes here requires domain knowledge since data consistency is crucial.

With each additional core, the worst case memory access times increase because it has to be assumed that all other cores access memory before the own request is being processed. This is one of the critical points limiting the scalability. Additionally, the number of shared variables to be synchronized increases with the number of cores.[30] Each variable to be synchronized gets two additional memory accesses for the `lock` and `unlock` functions protecting it. Therefore, a variable access taking one memory access without synchronization takes three memory accesses with synchronization. Furthermore, the accesses to non-shared data are cached, while those to shared data are not cached. These effects add to the worst case memory access times.

We assume that an application like this cannot be distributed over many cores without refactoring. For this, domain knowledge is needed again. However, even for this very large and complex application, our approach is able to achieve a WCET speedup for a few cores. This enables to start utilizing multi-core technology with low effort.

[30] Configurations with more cores usually have more parallel parts requiring more shared variables to be synchronized.

9.4 Comparision of the Results with Other Researches

Rochange et al. [49] already observed the slowdown of sequential code when utilizing more cores at the parallel implementation and WCET estimation of a parallel 3D multigrid solver: their implementation with a single thread has a WCET of 54 million cycles with one core and 71 million cycles with 9 cores present (but still only 1 thread being executed). For the WCET analysis, OTAWA was utilized. A WCET speedup was only possible when achieving a high parallelism, otherwise it was devoured by the high worst case memory access times.

Gerdes et al. [20] parallelized the control code of a drilling rig, which is most close to our foundation crane since it is another construction machinery series by BAUER. Their parallel implementation was based on domain knowledge, the WCET analysis took place with RapiTime. They achieved a WCET speedup of 1.08 for 2 cores and 1.93 for 4 cores. More cores were not investigated.

Kehr et al. [31] parallelized a diesel engine management system (EMS) for 2, 4 and 8 cores. They applied the original pattern-supported parallelization approach by Jahr et al. [27,28]. Their parallel EMS reaches a WCET speedup between 2.1 and 4.5 on 12 cores depending on the worst case overhead for data exchange. OTAWA was used for the WCET analysis. The highest efficiency is achieved with a WCET speedup between 1.2 and 2.7 on 4 cores. On two cores, only a WCET slowdown is realized. The authors describe that communication follows a repetitive pattern and the EMS is therefore “an ideal use case” for parallelization [31]. Their results might give a clue what is possible when an application is structured in a way facilitating parallelization.

The research of Cordes and Marwedel [11], Cordes et al. [12] focusses on automatic parallelization. As explained in the beginning, this makes timing analysis very hard. However, we make a comparison with their results since this might be the preferred way when parallelizing code without timing constraints. Cordes et al. [12] present speedups of the automatic parallelization of several applications and reach speedups of up to 3.7 with four cores. Their industrial application is part of the INTRACOM TELECOM’s Wimax system. It reaches a speedup of 2.2 on four cores, more cores are not investigated. A timing analysis does not take place because it is only a firm real-time application. Therefore, these results are only observed times, not including assumptions about worst case memory access times.

Another approach of automatic parallelization is presented by Kempf et al. [32]. They apply their parallelizing compiler on 44 real-world industrial applications and reach speedups of up to 2.0.[31] Again, these numbers are not WCETs, but only observed execution times.

Up to four cores, the results of the foundation crane control code parallelization are at the same level as other approaches—whereas our approach ensures timing analysability. Given the reasons why the foundation crane application scales bad, a comparison of the 8 core results does not make sense. Furthermore, there are only very few researches providing parallelization results for more than four cores.

[31] There is also one speedup of 5.0—Kempf et al. describe that this is a benchmark testing different components of the system. The parallelized version tests all components simultanously instead of successively.

10 Lessons Learned

While applying our parallelization approach on the two industrial applications, we got an insight on the importance of multi-core platforms for industrial needs, but also on the challenges occuring at the parallelization process. In Sect. 10.1, we summarize what we have learned from the parallelization of the industrial applications, in Sect. 10.2 we conclude what this means for legacy code parallelization in general and in Sect. 10.3 there are conclusions for the WCET analysis of parallel applications.

10.1 Lessons Learned from Parallelization of the Industrial Applications

Big opportunities for improvement arise from the move towards a multi-core platform. The main chance could be the higher performance which such platforms can provide. In the foundation crane example, this can help implementing more sophisticated safety measures and better control algorithms (closed loop instead of open loop). Also, because the system is not running at its performance limits anymore, the development effort could be reduced a bit because of lower need to find the most efficient algorithms at any price. At the signal processing application, more processing power can help to get results more often and therefore lead to better reactions on changes of the environment.

The most labor-intensive task was the analysis of the single-core source code. This could be improved by novel tools to detect data and control dependencies. Because of now better tool support the optimization towards a WCET speedup can be accelerated for future projects. The implementation is easy if algorithmic skeletons are applied for PDPs; for the synchronization the described code generator can assist. However, this synchronization work can be error-prone. The division of the original source code into already relatively independent function controlling defined parts of the machine helped very much in the recognition of chances for parallelism. However, the code parts are not as independent as it seemed at the first sight. Altogether, the foundation crane application is sticked together very closely with a lot of dependencies. While mutator functions for global shared variables are of minor importance in the single-core software they ease the synchronization in the parallel software because locks can be placed there instead of many different positions in the source code.

Currently, the critical issue is that suitable multi-core ECUs are available, e. g. Infineon AURIX [25], but the system software would have to be adapted by the ECU provider. Development tools would have to be changed to support changed debugging needs in multi-core processors, e. g. to find deadlocks or race conditions.

10.2 Conclusions for Parallelization in General

Parallelization of sequential legacy applications is highly desired to keep the testing and certification efforts low. However, systematic parallelization only seems to be a starting point. It looks like current industrial applications need to be more restructured to utilize more than a few cores. This is in accordance with the results of Kempf et

al. [32] and Gerdes et al. [20], who come to similar speedups and conclusions at the parallelization of industrial applications.

Furthermore, the potential of multi-core architectures may not be exploited by just parallelizing sequential legacy programs, but by utilizing the additional computational power for additional or more complex algorithms. This may not be catched by comparing speedups since multi-core architectures enable algorithms which were not possible on single-core architectures.

The further development of sequential legacy applications should respect the goal to move to a parallel platform. Therefore, paying attention at several points may increase the potential for parallelization. Developing functions in a way that they can be executed in parallel seems to be most important. This means to give them a specific region or scope where they work and to limit their side effects. The remaining side effects should be documented in detail. Computational intensive features should be implemented in a way that they could work on a dedicated core. Functions should get their input via parameters and not via global variables. The latter should be utilized only where necessary—ideally, they are only written in one function and read everywhere else. Large data structures should be broken down into smaller ones to enable parallel processing on independent parts.

Some developers already think "in a parallel way" because the execution of sequential program fragments—already on single-core platforms—is often interrupted by periodic tasks.

10.3 Conclusions for WCET Analysis of Parallel Applications

When employing a static WCET analysis tool like OTAWA [5], often very pessimistic assumptions have to be made, e. g. maximum conflicts on each global memory access or cache misses on almost every cache access. This leads to a high overestimation resulting in WCETs being far from what would be realistic. Measurement-based WCET tools like RapiTime [47] typically give better estimations, because they respect the real behaviour of the hardware. However, it is never sure if the worst case was catched during the measurements or if a state may be reached which is worse than that what was determined to be the "worst case".

Since both methods have shortcomings, current research focusses on alternative approaches. The first is probabilistic timing analysis [1,2,13], which assumes certain probabilities when accessing a cache or the global memory. Therefore, not each access is assumed as cache miss or a memory access with a very high latency, but realistic probabilities help to estimate a tighter WCET. Another solution could be typical worst case analysis [45,46], which basic idea is to ignore situations e. g. where the WCETs of two seldom activities add to each other when they happen at the same time. Thus, very unrealistic cases are not taken into account and the WCET estimation should also be better.

For estimating WCETs of parallel applications, these new approaches may be reasonable since assuming worst case access times everywhere seems to be too pessimistic.

With view to many-core architectures, a single global shared memory may be a bottleneck not only at the WCET, but also at the OET. Therefore, e. g. Metzlaff et al. [39] propose to utilize distributed memory in many-core architectures to facilitate better timing predictability.

11 Summary and Outlook

We presented a parallelization approach and applied it on two industrial applications. Our *parallelization approach for hard real-time systems* is an extension of the pattern-supported parallelization approach by Jahr et al., which was already applicable on embedded real-time systems. We extended it by an additional Phase respecting the needs of the implementation by employing timing-analyzable algorithmic skeletons (TAS) and code generation. Thus, our parallelization approach for hard real-time systems consists of three phases:

– Phase I: the legacy program is being analyzsed and represented in an APD. It consists of activities and PDPs exposing the program parts that can potentially be executed in parallel. Additionally, dependencies between the activities are represented in the diagram or in a list and the activities' WCETs are determined to show how much processing time they contribute to the overall WCET.
– Phase II: with the APD from Phase I and the additional information, a speedup approximation and parameter optimization tool can be utilized to optimize the APD. The goal is to minimize the overall WCET, the number of shared variables to be synchronized and the number of cores utilized.
– Phase III: while phases I and II worked on the model, in Phase III the source code is changed. The implementation takes place by employing the TAS and a custom code generator.

Finally, we added a refinement stage which means to repeat the phases II and III iteratively based on the WCET analysis results of the parallel version.

In our opinion, the parallelization approach is not limited to hard real-time systems and may be employed for any other program, too. The three phases were applied on a signal processing application from the avionics domain as well as on the industrial control code of a large construction machine—the foundation crane BAUER MC 128. Both are embedded hard real-time applications, having to fulfill timing constraints. The implementation took place for the parMERASA predictable multi-core platform with up to 18 cores. For evaluation, we estimated WCETs and WCET speedups for the sequential and parallel implementations. At the foundation crane control code, we compared the results with other researches.

We depicted not only the parallelization of these two industrial applications, but also the problems arising and why it is hard to get a WCET speedup. For the latter there is the problem of increasing WCETs at sequential program parts due to memory interferences and worst case waiting times (all other cores may interfere before the own memory request is processed). Therefore, the possible WCET speedup is strongly limited. At the signal processing application, we achieved a WCET speedup of 1.5 on 10 cores by utilizing the PDPs for data parallelism and pipelining. The control application of the foundation crane has a WCET speedup of 2.39 on 4 cores. These

results show that our parallelization approach works and it is possible to parallelize an embedded hard real-time application.

However, the control program of the foundation crane is sticked together very closely and therefore, with 8 cores a WCET slowdown of 0.15 occurs. This shows the lack of scalability of the foundation crane control code—a high contention caused by the software structure breaks the timing guarantees. Furthermore, the more cores are utilized, the stronger the effect of worst case memory access times is—alternatives at timing analysis techniques and/or to global shared memory may have to be employed. It should also be mentioned that the foundation crane control code is not a small example which can be parallelized easily, but the complete control application which controls everything on the machine and is a representative example for code from the automation domain. We assume that better results will be achieved by combining our parallelization approach with domain knowledge and a restructuring of the application.

Finally, we concluded what can be learned from the parallelizations: sequential software development can already prepare the way to increase the parallelization potential. Furthermore, alternative WCET estimation methods might be beneficial for the WCET analysis of parallel applications.

What we had to realize was that the tool support is low. Therefore, we developed some tools and also the TAS. Most of our developments are open source, the links have been provided throughout the article. As there is still lack of tool support for detecting situations where parallelism could be realized, we see this as our future work.

Acknowledgments The research leading to these results has received funding from the European Union Seventh Framework Programme under the name parMERASA and Grant Agreement No. 287519.

References

1. Abella, J., Hardy, D., Puaut, I., Quiñones, E., Cazorla, F.: On the comparison of deterministic and probabilistic WCET estimation techniques. In: 26th Euromicro Conference on Real-Time Systems (ECRTS), pp. 266–275 (2014). doi:10.1109/ECRTS.2014.16
2. Altmeyer, S., Cucu-Grosjean, L., Davis, R.: Static probabilistic timing analysis for real-time systems using random replacement caches. Real Time Syst. **51**(1), 77–123 (2015). doi:10.1007/s11241-014-9218-4
3. Amdahl, G.M.: Validity of the single processor approach to achieving large-scale computing capabilities. AFIPS Conference Proceedings, vol 30, pp. 483–485 (1967). doi:10.1145/1465482.1465560
4. Audsley, N., Tindell, K., Burns, A.: The end of the line for static cyclic scheduling? In: Fifth Euromicro Workshop on Real-Time Systems. Proceedings, pp. 36–41 (1993). doi:10.1109/EMWRT.1993.639042
5. Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: an open toolbox for adaptive WCET analysis. In: Software Technologies for Embedded and Ubiquitous Systems (Lecture Notes in Computer Science), vol 6399, pp. 35–46. Springer, Berlin (2011). doi:10.1007/978-3-642-16256-5_6
6. BAUER Maschinen GmbH: MC 128 foundation crane datasheet (2013). https://www.bauer.de/export/shared/pdf/bma/products/foundation_crane/905-659-2.pdf
7. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eSkel. In: Euro-Par 2005 Parallel Processing (Lecture Notes in Computer Science), vol 3648, pp. 761–770. Springer, Berlin (2005). doi:10.1007/11549468_83
8. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999). doi:10.1145/324133.324234
9. Bonenfant, A., Broster, I., Ballabriga, C., Bernat, G., Cassé, H., Houston, M., Merriam, N., de Michiel, M., Rochange, C., Sainrat, P.: 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 (2010)

10. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004). doi:10.1016/j.parco.2003.12.002
11. Cordes, D., Marwedel, P.: Multi-objective aware extraction of task-level parallelism using genetic algorithms. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, pp. 394–399 (2012). doi:10.1109/DATE.2012.6176503
12. Cordes, D., Marwedel, P., Mallik, A.: Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In: 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 267–276 (2010). doi:10.1145/1878961.1879009
13. Cucu-Grosjean, L., Santinelli, L., Houston, M., Lo, C., Vardanega, T., Kosmidis, L., Abella, J., Mezzetti, E., Quiñones, E., Cazorla, F.: Measurement-based probabilistic timing analysis for multi-path programs. In: 24th Euromicro Conference on Real-Time Systems (ECRTS), pp. 91–101 (2012). doi:10.1109/ECRTS.2012.31
14. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: QUAFF: efficient C++ design for parallel skeletons. Parallel Comput. **32**(7), 604–615 (2006). doi:10.1016/j.parco.2006.06.001
15. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
16. Fraboulet, A., Risset, T., Scherrer, A.: Computer systems: architectures, modeling, and simulation: third and fourth international workshops, SAMOS 2004, Samos, Greece, July 21–23, 2004 and July 19–21, 2004. In: Proceedings, Chap. Cycle Accurate Simulation Model Generation for SoC Prototyping, pp. 453–462. Springer, Berlin (2004). doi:10.1007/978-3-540-27776-7_47
17. Gebhard, G., Cullmann, C., Heckmann, R.: Software structure and WCET predictability. In: Bringing Theory to Practice: Predictability and Performance in Embedded Systems, vol 18, pp. 1–10. Dagstuhl, Germany (2011). doi:10.4230/OASIcs.PPES.2011.1
18. Gerdes, M., Jahr, R., Ungerer, T.: parMERASA Pattern Catalogue: Timing Predictable Parallel Design Patterns. Technical Report 2013-11, University of Augsburg (2013)
19. Gerdes, M., Kluge, F., Ungerer, T., Rochange, C., Sainrat, P.: Time analysable synchronisation techniques for parallelised hard real-time applications. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, pp. 671–676 (2012). doi:10.1109/DATE.2012.6176555
20. Gerdes, M., Wolf, J., Guliashvili, I., Ungerer, T., Houston, M., Bernat, G., Schnitzler, S., Regler, H.: Large drilling machine control code—Parallelisation and WCET speedup. In: 6th IEEE International Symposium on Industrial Embedded Systems (SIES), pp. 91–94 (2011). doi:10.1109/SIES.2011.5953688
21. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience **40**(12), 1135–1160 (2010). doi:10.1002/spe.1026
22. Gustavsson, A., Gustafsson, J., Lisper, B.: Toward static timing analysis of parallel software. In: 12th International Workshop on Worst-Case Execution Time Analysis, vol 23, pp. 38–47. Dagstuhl, Germany (2012). doi:10.4230/OASIcs.WCET.2012.38
23. Herlihy, M.: Wait-free synchronization. ACM Trans. Progr. Lang. Syst.: TOPLAS **13**(1), 124–149 (1991). doi:10.1145/114005.102808
24. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, pp. 522–529. IEEE (2003). doi:10.1109/ICDCS.2003.1203503
25. Infineon: AURIX—TC27x B-Step, 32-bit Single-Chip Micro-controller. User's Manual, v14.1
26. Jahr, R., Frieb, M., Gerdes, M., Ungerer, T.: 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, pp. 63–72. fortiss GmbH, München, Schloss Dagstuhl (2014). http://www4.in.tum.de/~schaetz/papers/MBEES2014.pdf
27. Jahr, R., Gerdes, M., Ungerer, T.: 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, pp. 50–59. Fortiss GmbH, München, Schloss Dagstuhl (2013). http://www4.in.tum.de/~schaetz/papers/MBEES2013.pdf

28. Jahr, R., Gerdes, M., Ungerer, T.: A pattern-supported parallelization approach. In: Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '13, pp. 53–62. ACM, New York (2013). doi:10.1145/2442992.2442998
29. Jahr, R., Gerdes, M., Ungerer, T., Ozaktas, H., Rochange, C., Zaykov, P.: Effects of structured parallelism by parallel design patterns on embedded hard real-time systems. In: IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–10 (2014). doi:10.1109/RTCSA.2014.6910546
30. Jahr, R., Stegmeier, A., Kiefhaber, R., Frieb, M., Ungerer, T.: User Manual for the Optimization and WCET Analysis of Software with Timing Analyzable Algorithmic Skeletons. Technical Report no. 2014-05, University of Augsburg (2014)
31. Kehr, S., Quiñones, E., Böddeker, B., Schäfer, G.: Parallel execution of AUTOSAR legacy applications on multicore ECUs with timed implicit communication. In: 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). San Francisco, USA (2015). doi:10.1145/2744769.2744889
32. Kempf, S., Veldema, R., Philippsen, M.: Is there hope for automatic parallelization of legacy industry automation applications? In: Gesellschaft für Informatik e.V. (ed.) Proceedings of the 24th Workshop on Parallel Systems and Algorithms (PARS), pp. 80–89 (2011). https://www2.informatik.uni-erlangen.de/publication/download/PARS2011.pdf
33. Keutzer, K., Massingill, B.L., Mattson, T.G., Sanders, B.A.: A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects. In: Proceedings of the 2010 Workshop on Parallel Programming Patterns, ParaPLoP '10, pp. 9:1–9:8. ACM, New York (2010). doi:10.1145/1953611.1953620
34. Liu, X., Zhou, J., Zhang, D., Shen, Y., Guo, M.: A parallel skeleton library for embedded multicores. In: 39th International Conference on Parallel Processing Workshops (ICPPW), pp. 65–73. IEEE (2010). doi:10.1109/ICPPW.2010.21
35. Lukas, R.G.: Dynamic compaction. Geotechnical Engineering Circular **No. 1**(FHWA-SA-95-037), 1–97 (1995). http://isddc.dot.gov/OLPFiles/FHWA/009754.pdf
36. Massingill, B.L., Mattson, T.G., Sanders, B.A.: Patterns for parallel application programs. In: Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP), Allerton Park in Monticello, IL (1999). http://www.hillside.net/plop/plop99/proceedings/massingill/massingill.pdf
37. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional, Boston, MA (2004)
38. Meade, A., Buckley, J., Collins, J.J.: Challenges of evolving sequential to parallel code: an exploratory review. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution, IWPSE-EVOL '11, pp. 1–5. ACM, New York (2011). doi:10.1145/2024445.2024447
39. Metzlaff, S., Mische, J., Ungerer, T.: A real-time capable many-core model. In: Proceedings of 32nd IEEE Real-Time Systems Symposium: Work-in-Progress Session, pp. 21–24. Vienna, Austria (2011). http://www.cs.wayne.edu/~fishern/Meetings/wip-rtss2011/WiP-RTSS-2011-Proceedings-Post.pdf
40. OMG Unified Modeling Language™(OMG UML), Version 2.5. Standardization Document (2015). http://www.omg.org/spec/UML/2.5/
41. Ozaktas, H., Rochange, C., Sainrat, P.: Automatic WCET analysis of real-time parallel applications. In: 13th International Workshop on Worst-Case Execution Time Analysis, vol 30, pp. 11–20. Dagstuhl, Germany (2013). doi:10.4230/OASIcs.WCET.2013.11
42. Panić, M., Kehr, S., Quiñones, E., Böddecker, B., Abella, J., Cazorla, F.J.: RunPar: an allocation algorithm for automotive applications exploiting runnable parallelism in multicores. In: 2014 ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS). New Delhi, India (2014). doi:10.1145/2656075.2656096
43. Predictable parMERASA Multicore Processor. Deliverable 5.3 of the parMERASA Project (2013). http://www.parmerasa.eu/files/deliverables/Deliverable_5_3.pdf
44. Puschner, P., Schoeberl, M.: On composable system timing, task timing, and WCET analysis. In: R. Kirner (ed.) 8th International Workshop on Worst-Case Execution Time Analysis (WCET'08) (OpenAccess Series in Informatics (OASIcs)), vol 8. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2008). doi:10.4230/OASIcs.WCET.2008.1662. Also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-237-3
45. Quinton, S., Bone, T.T., Hennig, J., Neukirchner, M., Negrean, M., Ernst, R.: Typical worst case response-time analysis and its use in automotive network design. In: Proceedings of the 51st Annual

Design Automation Conference, DAC ’14, pp. 44:1–44:6. ACM, New York (2014). doi:10.1145/2593069.2602977
46. Quinton, S., Hanke, M., Ernst, R.: Formal analysis of sporadic overload in real-time systems. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 515–520 (2012). doi:10.1109/DATE.2012.6176523
47. Rapita Systems: RapiTime explained. White Paper MC-WP-001-17, http://www.rapitasystems.com/downloads/white-papers/rapitime-explained
48. Report on support of tools for case studies. Deliverable 3.12 of the parMERASA Project (2014). http://www.parmerasa.eu/files/deliverables/Deliverable_3_12.pdf
49. Rochange, C., Bonenfant, A., Sainrat, P., Gerdes, M., Wolf, J., Ungerer, T., Petrov, Z., Mikulu, F.: WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In: 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), vol 15, pp. 90–100. Dagstuhl, Germany (2010). doi:10.4230/OASIcs.WCET.2010.90
50. Saifullah, A., Agrawal, K., Lu, C., Gill, C.: Multi-core real-time scheduling for generalized parallel task models. In: IEEE 32nd Real-Time Systems Symposium (RTSS), pp. 217–226 (2011). doi:10.1109/RTSS.2011.27
51. Schlingmann, S., Garbade, A., Weis, S., Ungerer, T.: Connectivity-sensitive algorithm for task placement on a many-core considering faulty regions. In: 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 417–422 (2011). doi:10.1109/PDP.2011.58
52. Sensortechnik Wiedemann GmbH: ESX-3XL. Data Sheet. http://www.sensor-technik.de/images/stories/pdf/download/esx-3xl_datenblatt_en.pdf (2014)
53. Sérot, J., Ginhac, D.: Skeletons for parallel image processing: an overview of the SKIPPER project. Parallel Comput. **28**(12), 1685–1708 (2002). doi:10.1016/S0167-8191(02)00189-8
54. Stegmeier, A., Frieb, M., Jahr, R., Ungerer, T.: Algorithmic skeletons for parallelization of embedded real-time systems. In: 3rd Workshop on High-Performance and Real-time Embedded Systems (HiRES) (2015). http://www.cister.isep.ipp.pt/hires2015/Algorithmic_Skeletons_for_Parallelization_of_Embedded_Real-time_Systems.pdf
55. Ungerer, T., Bradatsch, C., Frieb, M., Kluge, F., Mische, J., Stegmeier, A., Jahr, R., Gerdes, M., Zaykov, P., Matusova, L., Li, Z.J.J., Petrov, Z., Böddeker, B., Kehr, S., Regler, H., Hugl, A., Rochange, C., Ozaktas, H., Cassé, H., Bonenfant, A., Sainrat, P., Lay, N., George, D., Broster, I., Quiñones, E., Panić, M., Abella, J., Hernandez, C., Cazorla, F., Uhrig, S., Rohde, M., Pyka, A.: Parallelizing industrial hard real-time applications for the parMERASA multi-core. Trans. Embed. Comput. Syst.: TECS (2016) (To appear)
56. Ungerer, T., Bradatsch, C., Gerdes, M., Kluge, F., Jahr, R., Mische, J., Fernandes, J., Zaykov, P., Petrov, Z., Böddeker, B., Kehr, S., Regler, H., Hugl, A., Rochange, C., Ozaktas, H., Casse, H., Bonenfant, A., Sainrat, P., Broster, I., Lay, N., George, D., Quiñones, E., Panić, M., Abella, J., Cazorla, F., Uhrig, S., Rohde, M., Pyka, A.: parMERASA—multi-core execution of parallelised hard real-time applications supporting analysability. In: 2013 Euromicro Conference on Digital System Design (DSD), pp. 363–370 (2013). doi:10.1109/DSD.2013.46
57. Ungerer, T., Bradatsch, C., Gerdes, M., Kluge, F., Jahr, R., Mische, J., Stegmeier, A., Frieb, M., Fernandes, J., Zaykov, P., Petrov, Z., Böddeker, B., Kehr, S., Regler, H., Hugl, A., Rochange, C., Ozaktas, H., Casse, H., Bonenfant, A., Sainrat, P., Broster, I., Lay, N., George, D., Quiñones, E., Panić, M., Abella, J., Cazorla, F., Uhrig, S., Rohde, M., Pyka, A.: Experiences and results of parallelisation of industrial hard real-time applications for the parMERASA multi-core. In: 3rd Workshop on High-Performance and Real-Time Embedded Systems (HiRES) (2015). http://www.cister.isep.ipp.pt/hires2015/Experiences_and_Results_of_Parallelisation_of_Industrial_Hard_Real-time_Applications_for_the_parMERASA_Multi-core.pdf
58. Wang, Z., O’Boyle, M.F.: Mapping parallelism to multi-cores: a machine learning based approach. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’09, pp. 75–84. ACM, New York (2009). doi:10.1145/1504176.1504189
59. Wilhelm, R., Engblom, J., Aandreas, E., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution time problem-overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (2008). doi:10.1145/1347375.1347389