World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:9, No:1, 2015

Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems

Robert Höttger, Lukas Krawczyk, Burkhard Igel

Abstract—This paper introduces novel approaches to partitioning and mapping in terms of model-based embedded multicore system engineering and further discusses benefits, industrial relevance and features in common with existing approaches. In order to assess and evaluate results, both approaches have been applied to a real industrial application as well as to various prototypical demonstrative applications, that have been developed and implemented for different purposes. Evaluations show, that such applications improve significantly according to performance, energy efficiency, meeting timing constraints and covering maintaining issues by using the AMALTHEA[1] platform and the implemented approaches. Furthermore, the model-based design provides an open, expandable, platform independent and scalable exchange format between OEMs, suppliers and developers on different levels. Our proposed mechanisms provide meaningful multicore system utilization since load balancing by means of partitioning and mapping is effectively performed with regard to the modeled systems including hardware, software, operating system, scheduling, constraints, configuration and more data.

Keywords—Partitioning, mapping, distributed systems, scheduling, embedded multicore systems, model-based, system analysis.

I. Introduction

SINCE the automotive industry is still facing specific challenges among parallelism exploitations for embedded multicore systems especially in context with the AUTOSAR [1] standard, the AMALTHEA platform [2] has been developed in order to face these challenges and create meaningful solutions by providing automatic and effective processes for application distribution. Furthermore, increasing demands and requirements like computational power, safety issues, energy efficiency, various standards or product-line engineering call for the need of new approaches. Partitioning and mapping are two mandatory and established methods for program parallelization, that exhibit enormous impact on the complexity of and the required effort for system engineering. Hence, facilitating these processes among an automatic, scalable and modular basis becomes an important issue. By providing a development comprehensive, model-based, open, expandable and platform-independent exchange format in AMALTHEA, partitioning and mapping in the same context provide meaningful relevance in terms of industrial- and scientific-based applications.

Partitioning focuses in our work on forming `tasks`, consisting of `runnables`, that define execution units with a specific amount of `instructions`, and dependencies, that are derived from read and write accesses to `labels` (abstract memory), via an automatic mechanism with different objectives. The mechanism examines weighted directed acyclic graphs (*WDAG*s) and resulting partitions can be effectively distributed among processors without violating ordering constraints of `runnables` while considering `activation` rates, independent `runnable` groups, cyclic dependencies and communication overheads.

Mapping faces the problem of assigning `tasks` to specific processors effectively and thereby identifying a `task` allocation solution, that utilizes given resources in form of hardware models by different objectives. Such objectives are either implemented using *ILP* (integer linear programming) solver with respect to *load balancing* or *energy-aware mapping* strategies or by using heuristic methods like *DFG* (*Data Flow Graph*) *load balancing* in order to find effective solutions.

The parallel execution of such partitioned and mapped applications benefits from lower execution times and less energy consumption compared with sequential execution due to running more processors with lower frequencies [3]. Since both mechanisms rely on the model-based architecture and models can be imported and exported to and from industrial established tools as well as verified against various specifications, the implementations remain AUTOSAR compliant and provide consistent integration to state of the art automotive developments.

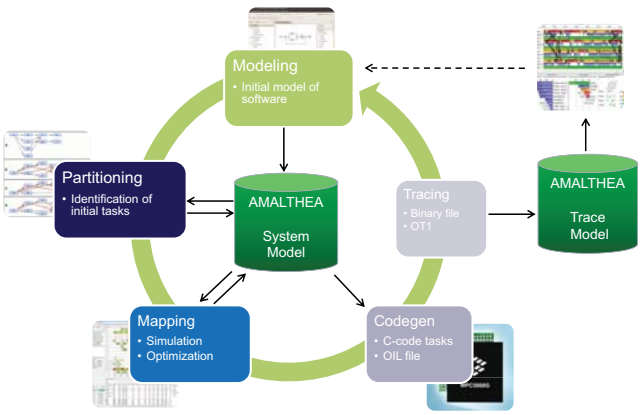

Fig. 1 Amalthea Platform: Processing Simulation and Analysis [has been composed in cooperation with Robert Bosch GmbH]

Fig. 1 shows both the partitioning and the mapping approaches integrated to the AMALTHEA platform. Each process accesses specific models and adapts them based on their configuration and corresponding model analyses. Further topics like *Modeling*, *Codegeneration* and *Tracing* provide necessary features for comprehensive system engineering but are not

Robert Höttger, Lukas Krawczyk and Burkhard Igel are with the Pimes Research Department at Dortmund University of Applied Sciences and Arts, Otto-Hahn-Str. 23, 44227 Dortmund, Germany (E-mail: {robert.hoettger, lukas.krawczyk, igel@fh-dortmund.de)

Funded by the ITEA 2 committee and the *Bundesministerium für Foschung und Bildung*, Germany

[1]ITEA 2 project call 4 Project No. 09013

scope of this paper. Further information is given in [4].

The paper is organized as follows. The next section II outlines basic related work, whereas specific related work is referenced all over the paper. Section III describes the partitioning approaches as well as corresponding necessary and optional features, among their scientific intend, implementations and an example. Section V describes the subsequent mapping phase and outlines main benefits among another example. Finally, Section VI concludes the presented contents, emphasizes on benefits for industrial applications and proposes some future work.

II. Related Work

Over the years, several strategies for partitioning and mapping in context of embedded software have been developed. In [5] a simulation methodology for embedded real-time and cyber-physical systems is introduced, which focuses on timing behavior as well as the combination of architecture properties and requirements. Each domain can be represented with specific computation models and recent work is addressing the multicore migration topic by means of specific algorithms.

Recent research featuring *Augmented Hierarchical Task Graphs* and several optimization techniques e.g. genetic algorithms and integer linear programming, has been described by Cordes in [6]. The author describes a parallelization framework, which integrates these techniques in order to perform an automated partitioning and mapping of software to heterogeneous hardware.

Furthermore, various scheduling publications like [7] or [8] or hardware and software co-synthesis publications using genetic algorithms like [9] address similar problems like the partitioning and mapping approaches of our work. However, compared to [7] our partitioning approach neither requires a unique exit node nor node duplication and merges `tasks` just within a specific configuration using a heuristic method. We further provide optimal *WDAG* partitions according to Amdahl's law [10].

We will see, that various algorithms and approaches will be used for different purposes in the course of this paper. References are stated correspondingly.

III. Partitioning Approach

Forming partitions mostly concerns the division of processes into subprocesses whereas each subprocess, mostly denoted as a node (or `runnable` in our case), consists of computational load (also denoted as weight i.e. `instructions`) [11]. A partitioning process in context of (*WDAG*s), which occur in most computing applications as computation *DAGs* or task graphs [12], influences system performance according to later parallel execution. The more efficient the partitioning process forms computation sets distributed among computation units i.e. processors, the more the system benefits from time issues, energy demands or resource utilization in order to improve real-time applications. These aspects are common topics of interest in almost all areas of science and technology. Effective partitioning and mapping mechanisms provide parallel executable applications, consuming less processor time, such that more computation power can be used for further needs or less energy is consumed due to lower processor utilization. The following sections describe specific AMALTHEA approaches, that have been implemented within the tool platform in [2], for distributing modeled applications.

A. Preliminary Features

In terms of graph-theoretical computing, partitioning *DAGs* comes with a wide range of problems and methodologies, spread across a variety of applications. Although many tools and algorithms have been developed for such purposes in the past decades, they are not fully applicable to embedded real-time systems due to their desktop-, computer- or high-performance focus. Therefore, we propose different phases, that address the challenges in the automotive embedded real-time domain.

AMALTHEA data is stored within open source *eclipse EMF* models for various representations. Information used within the approaches presented in this paper address the *software model* consisting of `labels` (abstract memory), `runnables`, `label accesses` and `activations` as well as the *constraints model* consisting `RunnableSequencingConstraints` (dependencies). More detailed AUTOSAR[1]-compliant model descriptions can be found at [4].

For the embedded real-time system focus, the initial phase addresses grouping `runnables` referencing the same `activation` instance, such that any group only contains `runnables` with the same `activation` rate. `Activations` can be set to `periodic`, `single`, `sporadic` or `custom` and define the call frequency or call intervals of the referenced model element. Groups are modeled within `ProcessPrototypes`, that reference a specific `activation` and define a pre-`task` state for analysis purposes. `ProcessPrototypes` are transformed to `tasks` 1-to-1 after the analysis phases, such that a `task` consists of multiple `runnables`, can be called by a scheduler and features different properties and data like call graphs, activations and more.

Since our partitioning approach requires *WDAGs*, the subsequent phase considers dependency analysis and the elimination of directed cycles, also denoted as feedbacks. In our case, dependencies are derived from `runnable`'s `label accesses`, such that, if `runnable` *A* writes a `label` and `runnable` *B* reads the same `label`, a `RunnableSequencingConstraint` (*RSC*) within the *constrains model* is created, containing *A* as parent `runnable` and *B* as child `runnable`. In other words, dependencies are automatically derived from `label` accesses and stored within the `constraints model`, representing directed edges. In order to utilize the advantages of *WDAGs*, the recently available *WDGs* have to be transformed to be acyclic. Since such cycle elimination corresponds minimal feedback arc set (*MFAS*) determination (where an arc denotes an edge i.e. a dependency within a modeled *RSC*) and is

an NP-complete problem [13], we firstly check whether the *WDAG* is planar. In the latter case, the problem can be solved within polynomial time [14]. In case the *WDAG* is non-planar, finding a minimum feedback arc set is trivial, enumerating all instances efficiently is not necessarily as simple [15]. However, different *MFAS* feature different resulting graphs, if the *MFAS* is removed or transformed for feedback elimination. Such transformation is addressed in our case, in order to retain the dependencies for later analysis phases or code generations. In model terminology, *MFAS RSCs* are decomposed into `AccessPrecedences`. These `AccessPrecedences` are used to represent dependency on the one hand but to keep the dependency invisible to graph analysis on the other hand. According to later phases i.e. the code generation, such `AccessPrecedences` also define the data `runnables` work with, since some `runnables` need to be informed to work with data, that has been calculated in preceding iterations instead of waiting for current iteration updates.

Hence, each *MFAS* resulting *WDAG* is assessed according to parallelization potential and sequential runtime (the graph's critical path), such that a *MFAS* is identified, that provides minimal edge transformation and minimal sequential runtime regarding the resulting graph in case the edges are transformed. The following (1) results in the cardinality of *MFAS* solutions:

$$|MFAS| = \sum^{C_i \setminus C_{mce}} \left(\prod^{C_j \setminus C_i \cup C_{mce}} |E_{C_j}| \right) - |E_{C_{mce}(mce)}| \tag{1}$$

C describes a cycle, E_{C_i} the edges within the cycle C_i, C_{mce} the cycle with the most common edges with other cycles and $|E_{C_{mce}(mce)}|$ the most common edge cardinality of the cycle with most common edges. Equation (1) adds the products of each remaining cycle's edge amounts to each edge, that belongs to more than one cycle. This ensures on the one hand, that only the minimal number of edges are transformed and on the other hand, that all *MFAS* combinations are considered. Since each product contains a solution, that includes all most common edges, these solutions must be subtracted (via the term $-|E_{C_{mce}(mce)}|$ in 1), since they are not valid or ineffective (because they mostly result in a complete sequence $\rightarrow$ not parallelizable). Simple cycles are determined via the Tarjan algorithm [16] using the JgraphT library [17]. The result of this phase are *WDAG*s and `AccessPrecedences` for the transformed dependencies.

Before the actual partitioning is performed, another phase can be performed, that identifies independent graphs. Independent graphs do not share any resource or `activation` and are modeled within `ProcessPrototypes`. Such methodology allows forming `tasks`, that can be totally distributed to either different cores or even to totally different systems or electronic control units (ECUs). Furthermore, such independent `ProcessPrototypes` can be used for components, providing modularity and reusability. This phase also allows independent `runnables` to be merged into the same `ProcessPrototype` to prevent unnecessary system overhead (context switches, data passing) via considering the system's critical path (*CP*), that provides the lower bound on the total time to execute a complete *DAG* [18]. The *CP* is defined by `runnables` and dependencies, forming a path from an entry node to an exit node, of which the sum of computation and communication costs is the maximum [19]. Communication costs is in this case derived from the `label` size of the information exchanged between `runnables`. The `ProcessPrototype` merging methodology is based upon a simple timing based combination heuristic, that merges two `ProcessPrototypes`, if they do not exhibit overlapping computations i.e. `runnables`, that are assigned to similar time slices at different `ProcessPrototypes`.

The resulting *WDAG* is defined via $G = (V, E, \mathcal{C}, \mathcal{T})$ where V is the set of vertices (`runnables`), E is the set of edges (dependencies), $\mathcal{C}$ is the set of communication costs and $\mathcal{T}$ is the set of node computation costs.

B. Critical Path Partitioning

The previously mentioned *CP* is also used for one partitioning approach. Sine the *CP* shall not be distributed due to increased execution time caused by communication and data exchange, it is assigned to the first `ProcessPrototype` and the graph's branches are assigned to further `ProcessPrototypes` following a specific algorithm shown in the above pseudocode Algorithm 1. Further `ProcessPrototypes` (next to the *CP*) never exceed the *CP*'s execution or cause the *CP* to wait on input data. [20] already presents, that critical path partitioning, also denoted as dominant sequence clustering, provides comparable or even better performance than much-higher-complexity heuristics.

```
1  Determine runnable's topological orders and earliest initial time
     (eit) and latest start time (lst) values
2  Let T denote the set of tasks
3  Determine the graph's critical path CP and assign it to the first
     task t in T
4  Let U denote all unassigned runnables
5  WHILE U is not empty
6    create task t_x, set tt=0;
7    WHILE task time tt < CPTime
8      let ar denote the set of assignable runnables according to tt,
         eit and lst values and preceding runnables
9      SWITCH ar.size
10       CASE 0:
11         No assignable runnable → increase tt to eit of the next
             applicable runnable
12       CASE 1:
13         Assign the runnable (ar[0]) to t and remove it from U
14       CASE >1:
15         determine each runnable's communication overhead and
             shifting potential and the corresponding most effective
             assignable runnable mer, assign mer to t, increase tt
             correspondingly and remove mer from U
16     ENDSWITCH
17   ENDWHILE
18 ENDWHILE
```

Algorithm 1. Pseudocode for critical path partitioning algorithm

The Algorithm 1 calculates a time frame for each `runnable` (line 1 in Algorithm 1), that provides earliest initial time (*eit*) and latest start time (*lst*) values. These values define, to which extent a `runnable` can be shifted with regard to the *CP* in order to not violate any order constraints defined by dependencies. The algorithm creates `ProcessPrototypes` (respectively `tasks` in line 6 in Algorithm 1) and checks time-slice-wise (line 7 in Algorithm 1) which `runnables` can be assigned according to ordering constraints i.e. dependencies (all preceding `runnables` must have finished their execution) (line 8 in Algorithm 1). From that `runnable`

amount, the algorithm decides, which runnable features the most limited time frame, such that runnables with more flexible time shifting values are selected at succeeding assignments (line 15 in Algorithm 1). This approach ensures minimal overall execution time and an accordingly low number of tasks. However, this approach is not able to limit the number of tasks, since the creation process is automated.

C. ESS Partitioning

Another implemented partitioning approach is based on an earliest start scheduling (*ESS*) and features a task number limitation. For this purpose, only runnable's *eit*-values are calculated, that define the sum of the longest preceding path's instructions.

```
1  Let R denote the set of all runnables
2  Let T denote the set of tasks (+init)
3  Let A denote the set of assigned runnables
4  WHILE A.size < R.size
5    let edr denote the runnable with lowest eit
6    let tdi denote the indices of tasks, to which latest runnables
       edr is dependent to
7    SWITCH (tdi)
8      CASE 0:
9        assign edr to task, that features the lowest tt / utilization
           , not earlier than edr's max end time according to current
           distribution
10     CASE 1:
11       assign edr to task, that edr is dependent to
12     CASE >1:
13       assign edr to task with latest dependency
14   ENDSWITCH
15 ENDWHILE
```

Algorithm 2. Pseudocode for ESS partitioning algorithm

The value of T is predefined by the user. However, in case a complete sequential runnable ordering would define the algorithm's input, all runnables would be assigned to the same partition (task), since line 8 and the subsequent switch case in Algorithm 2 keeps track of the dependencies and distributing a sequence of runnables would increase overall execution time due to additional synchronization and communication. Such synchronization or communication is required for partitions to share data via shared memory for example. We will later see, that the *ESS* approach provides better partitions compared with approaches based on earliest-deadline-first scheduling for instance. Line 9 in Algorithm 2 ensures, that load is balanced among partitions in case no parent of the runnable is located at the current tasks, since the runnable is assigned to the task featuring the lowest utilization and no order constraint is violated (parent assignment consideration). Line 11 and 13 assign runnables to the task, that contains the runnable's (latest) parent. This keeps also communication low since no data has to be exchanged to other tasks and balances runnables among the predefined number of partitions.

Fig. 2 outlines all previously described features and phases i.e. activation analysis, label access analysis, cycle elimination and the partitioning approaches denoted as graph analysis. The different phases provide a modular structure and efficient runnable distribution among partitions. Assessments are stated in section IV. Each phase can be configured and performed via a configuration- and a context menu or via a workflow engine provided by the AMALTHEA Tool platform.

Two more features are implemented within the partitioning methodology, which concern the graph visualization with the help of Jgraph [17] applets and the generation of more flexible graph representation models. The latter case addresses further analyses and optimization methods with commercial tools like the TA Toolsuite [21].

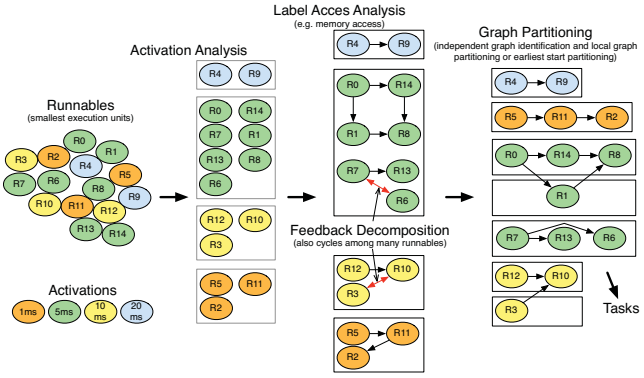

Fig. 2 Partitioning Phases: Activation Analysis → Label Access Analysis → Feedback Decomposition → Graph Partitioning

IV. Partitioning Evaluation

The following sections describe key metrics for performance assessment in IV-A and evaluation regarding applications features and types including an actual embedded system application evaluation in IV-B.

A. Key Metrics

The partitioning approaches perform various calculations among *WDAG*s, in order to gain information about partial orders and effective load balancing. *WDAG* information is defined via the *span*, that is the length of the critical path and via the *work*, that is the sum of the node's execution cycles (instructions). The partitioning's result can be evaluated with regard to key metrics like *parallelism*, that is the sequential runtime divided by the parallel runtime, and the *slackness*, that is the factor by which the parallelism exceed the number of processors in the system [18]. *EDF* results are based on earliest deadline first scheduling [22]. Given an example graph from Fig. 3, we can derive

$$span = \sum_{R_{CP,i=0}}^{R_{CP,n}} I(R_{CP,i}), CP = \{D9, J4; span = 9{+}4 = 13 \tag{2}$$

with $I(R_i)$ defining the Runnable R_i's instructions,

$$work = \sum_{R_{i=0}}^{R_n} I(R_i) = 34 \tag{3}$$

parallelism factors p_x:

$$p_{CPP} = \frac{34}{13} \approx 2,62; p_{ESS2} = \frac{34}{17} = 2; p_{ESS3} = \frac{34}{14} \approx 2,43;$$
$$p_{EDF2} = \frac{34}{22} \approx 1,55; p_{EDF3} = \frac{34}{18} \approx 1,89 \tag{4}$$

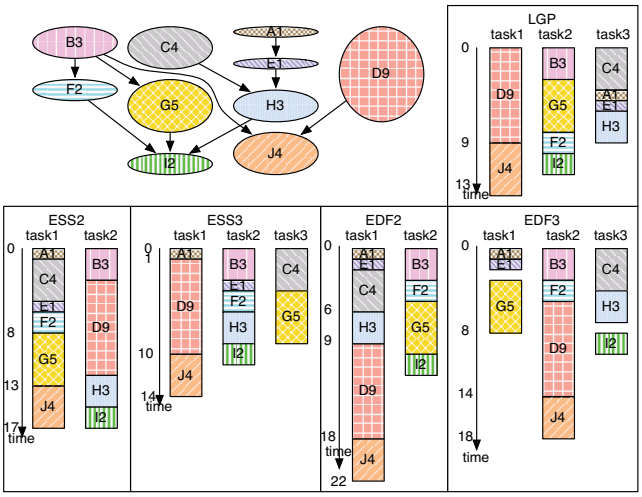

Fig. 3 Partitioning solutions with example graph

and slackness factors s_x:

$$s_{CPP} = \frac{34}{3 \cdot 13} \approx 0,87; s_{ESS2} = \frac{34}{2 \cdot 17} = 1;$$
$$s_{ESS3} = \frac{34}{3 \cdot 14} \approx 0,81; s_{EDF2} = \frac{34}{2 \cdot 22} \approx 0,77; \quad (5)$$
$$s_{EDF3} = \frac{34}{3 \cdot 18} \approx 0,63$$

Furthermore, Amdahl's law [10] is used for speedup assessment using the following (6):

$$\frac{Time_{before}}{Time_{after}} = \frac{1}{\frac{f}{K} + (1 - f)} \leq Speedup_{max} = \frac{1}{1 - f} = \frac{T_1}{T_\infty} \quad (6)$$

Amdahl's law in (6) defines the total speedup, in case a change improves a fraction f of the workload by a factor K. Some references also denote f as the amount of parallelizable code and K as the amount of processors.

Since the *CPP* approach always considers the critical path (*span*) as the most cost intensive `task` and creates `tasks` with no limitation among the branches of the graph, it always creates optimal system partitions, such that the resulting speedup corresponds Amdahl's law's best solution $\frac{T_1}{T_\infty}$ and thereby the maximal speedup. Both *ESS*'s and *EDF*'s speedup factors are calculated using the left part of 6 and thereby mostly do not achieve maximal speedup due to their `task` number limitation. However, the *ESS* approach is still able to create partitions that feature better slackness and speedup factors compared with *EDF* solutions. With regard to solutions shown in Fig. 3, the *CPP* provides the best complete system execution time of 14 instructions, followed by the *ESS3* solution, that executes within 15 instructions.

Fig. 4 shows the system's speedup depending on the critical path's length among three different amount of partitions. Such system speedups can be achieved using the *CPP* approach without task number limitation. The values are calculated using Amdahl's law and the parallelism factors based on a simple system with 1000 equal `runnables`. Fig. 4 reveals, that the shorter the critical path gets, the more speedup can be achieved using the *CPP* partitioning.

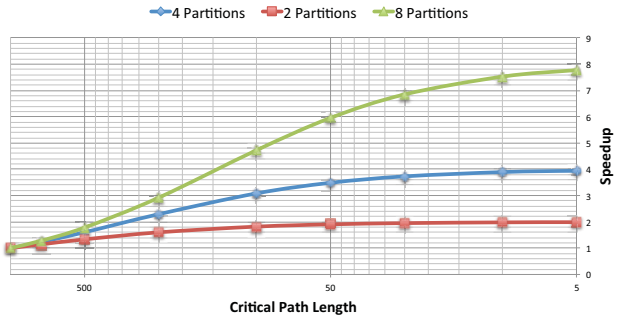

Fig. 4 CPP's Speedup (critical path length) diagram

B. Application Features and Types

For the purpose of comparing the presented approaches and assessing them among features, types and resulting partitions, Fig. 5 provides four graphs from different origins with regard to the presented key metrics. The first graph describes a fictional system, that features a balanced graph structure, such that optimal slackness factors of 1 can be achieved using the *CPP* or ESS_5 approaches (index of 5 refers to the configured amount of tasks). The second graph represents the example from Fig. 3. The third graph has been derived from a real automotive engine control system and the fourth graph has been transformed from a control flow oriented example.

The *ESS* approach provides more industrial relevance because of its task number limitation feature. As control flow oriented applications especially according to bigger structures feature relatively low *span* but rather high *work* values, the amount of tasks grows disproportionately. Hence, it is important to note, that the great parallelism factor of 6.28 at the $G4_{ESS}$ result could only be achieved using 53 tasks. The rather low utilization of the result can be derived from the low slackness value of 0.12 such that the ESS_4 result, featuring four tasks, describes a more reasonable solution.

According to smaller and certain data flow oriented applications, the *CPP* approach has been proven to provide more effective parallel executable partitions according to overall system execution time as seen in Fig. 5. The slackness factors thereby indicate the load balance among tasks and *ESS* solutions should be preferred as soon as the *CPP*'s slackness value falls below the *ESS*'s. *EDF* results feature higher execution and worse (lower) slackness factors compared with *ESS* solutions according to all four graphs and applications respectively.

With regard to slackness, execution and parallelism values, the user gets informed about which partitioning method creates more reasonable solutions within AMALTHEA [2].

V. Mapping Approach

While the focus of the partitioning approach lies in determining an ideal granularity and `runnable` distribution for the executable software partitions (`runnables`, `tasks`), the goal of the mapping process is to find one optimal allocation to the processors (cores) of a multi- or many-core hardware target. For this, we have implemented

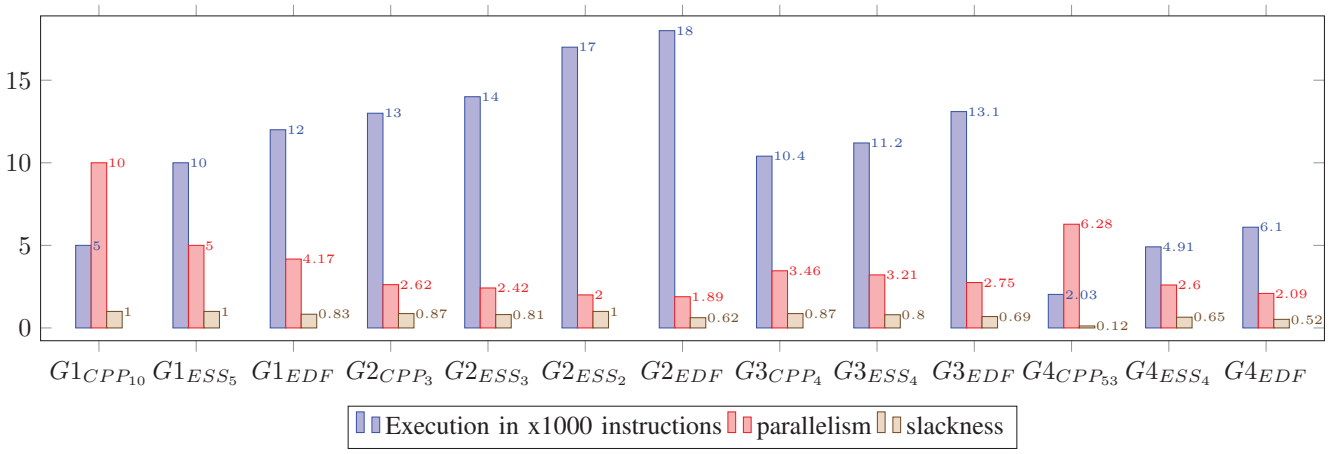

Fig. 5 CPP, ESS and EDF partitioning comparison regarding execution, parallelism and slackness values

several ILP based strategies in order to support the creation of pareto-optimal mapping solutions, each optimized towards one specific goal, e.g. minimizing the total execution time [23] or energy consumption [24]. The strategy for execution time minimization performs load-balancing in order to allocate tasks to cores, whereas the strategy for minimizing the energy consumption utilizes an heuristic algorithm. Minimizing the energy consumption is furthermore achieved by exploiting less power consuming voltage levels, which slow down tasks without harming the applications deadline. For both strategies, the ILP models describing the allocation problem are automatically created by the Amalthea Tool Platform, using only abstract models of the hardware and software descriptions.

Since determining such an optimal allocation is well known to be a NP complete problem, finding solutions for especially larger problem sizes will usually require a substantial amount of time. By following our two-phased approach however, we are able to significantly reduce the number of allocation subjects to a mere fraction of the original problem's size using one of the presented partitioning functionalities. A typical engine control system for instance can usually consist of over 1200 `runnables`, which leads to approx. 6×10^{933} valid allocations. By applying the partitioning approach, these can be agglomerated into 54 `tasks`, abstracting the problem and reducing the amount of possible allocations to approx. 1×10^{72}, which are comparatively trivial to optimize.

The results of our experimental evaluation using both presented mapping strategies are illustrated in Fig. 6. It shows the minimal, average and maximal run-time, which is based on ten batched executions of the respective mapping strategy. During this evaluation, the energy minimization strategy is used for distributing 43 `tasks` on a hardware platform with four homogeneous cores, each consisting of two voltage levels, whereas the execution time minimization strategy is applied on a system with 54 `tasks` and a hardware platform with six heterogeneous cores. The functionality for solving both ILP models is provided by the open-source *oj! Algorithms* library[2].

[2]see http://ojalgo.org/

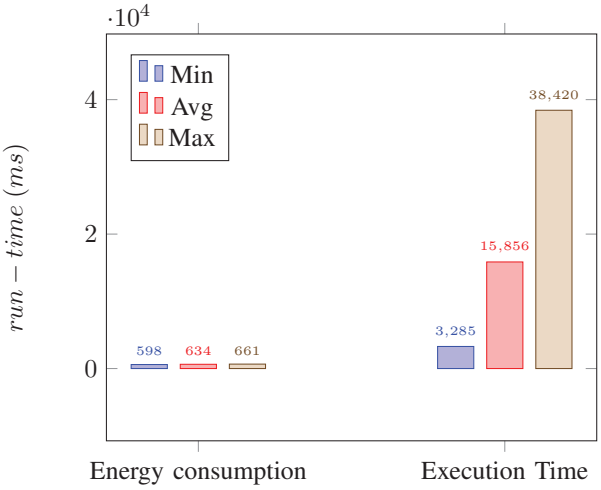

Fig. 6 Measured minimal, average and maximal run-times for applying the strategies for minimizing the *execution time* and *energy consumption*.

As we can see, both approaches will generate allocations in less then one minute, which makes the mapping functionality suitable for rapid development of embedded systems. The quickest mapping generation can be observed on the energy minimization approach, which remains stable at less then one second. Since the approach for minimizing energy consumptions can be solved in a polynomial *WCET* (worst case execution time) for target cores with two voltage levels [24], this method will be suitable for even larger problem models. The run-time for the execution time minimization approach takes up to 39 seconds, which also indicates the upper bound of allocable `tasks` if rapid development is desired. Increasing the number of `tasks` to e.g. 72 will lead to run-times of up to 10 minutes.

VI. Conclusion

In this paper, we have presented partitioning and mapping approaches among the AMALTHEA Tool platform as well as their benefits in developing automotive embedded multicore systems. Different required analyses like the cycle elimination,

activation consideration or the model-based design address various demands of the automotive industry. By using the interfaces of AMALTHEA, the user is able to develop and automatically distribute AUTOSAR compliant applications for embedded multicore systems. We discussed *ESS*, *EDF* and *CPP* based partitioning techniques for *WDAG*s, which are used for solving the `task` partitioning problems. We show, that partitions can be formed optimally with no `task` number limitation and effectively with `task` number limitation according to key metrics.

With regard to mapping, *ILP* based methods have been presented, which provide optimal mapping generations towards specific goals, e.g. energy efficiency. In future work, we expect to extend the partitioning and mapping methods and offer further optimization objectives, e.g. memory utilization. Furthermore, measurements based on daily modality will be performed to show the precise gain of the AMALTHEA tool platform and in order to reveal further optimization potential.

Finally, we can conclude, that the partitioning approach can be used to agglomerate the `runnables` into `tasks` in order to simplify the mapping process and to facilitate optimization techniques, that could not be applied to the high (unpartitioned) amount of `runnables` otherwise. The important innovation is the two phase approach as well as the combination and adaption of various methods and techniques among *WDAG* partitioning and *ILP* mapping.

Acknowledgment

The authors would like to express their appreciation to the AMALTHEA consortium for sharing expertise, experience and knowledge.

References

[1] "Autosar - automotive open system architecture," http://www.autosar.org, June 2014.
[2] "Amalthea project homepage," http://www.amalthea-project.org/, April 2014.
[3] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '94. Berkeley, CA, USA: USENIX Association, 1994. [Online]. Available: http://dblp.uni-trier.de/db/conf/osdi/osdi94.html
[4] *Amalthea platform: Help documentation*, Itea 2 project 09013 Amalthea, www.amalthea-project.org, 2014.
[5] T. Kuhn, T. Forster, T. Braun, and R. Gotzhein, "Feral - framework for simulator coupling on requirements and architecture level." in *MEMOCODE*. IEEE, 2013, pp. 11–22. [Online]. Available: http://dblp.uni-trier.de/db/conf/memocode/memocode2013.html
[6] D. A. Cordes, "Automatic parallelization for embedded multi-core systems using high-level cost models," Ph.D. dissertation, Technische Universität Dortmund, 2013.
[7] T.-Y. Choe, "Task scheduling algorithm to reduce the number of processors using merge conditions," *International Journal on Computer Science and Engineering (IJCSE)*, February 2012.
[8] K. Kanoun, D. Atienza, N. Mastronarde, and M. van der Schaar, "A unified online directed acyclic graph flow manager for multicore schedulers," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, 2014, pp. 714–719. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6742974
[9] G. S. Hornby, L. Sekanina, and P. C. Haddow, "Evolvable systems: From biology to hardware," in *8th International Conference, ICES 2008*. Springer, 2008.
[10] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring), New York, NY, USA, 1967, pp. 483–485.
[11] R. Preis, "Analysis and design of efficient graph partitioning methods," Ph.D. dissertation, University Paderborn, 2000.
[12] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.
[13] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.
[14] C. Lucchesi, *A Minimax Equality for Directed Graphs*. Thesis (Ph.D.)–University of Waterloo, 1976. [Online]. Available: http://books.google.de/books?id=KA8nnQEACAAJ
[15] B. Schwikowski and E. Speckenmeyer, "On enumerating all minimal solutions of feedback problems," *Discrete Applied Mathematics*, vol. 117, no. 1-3, pp. 253–265, Mar. 2002.
[16] R. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, 1972.
[17] "Jgrapht - a free java graph library," http://jgrapht.org/, November 2013.
[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009, vol. 3, ch. 21, 24 and 27.
[19] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 7, no. 5, pp. 506–521, 1996.
[20] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, 1993.
[21] T. A. GmbH, "Personal meetings and technical discussions," March 2014, unpublished.
[22] J. Hong, X. Tan, and D. Towsley, "A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1736–1744, 1989. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=40851
[23] M. Drozdowski, *Scheduling for Parallel Processing*, ser. Computer Communications and Networks. Springer, 2009.
[24] Y. Zhang, X. S. Hu, and D. Z. Chen, "Task scheduling and voltage selection for energy minimization," in *Proceedings of the 39th annual Design Automation Conference*. ACM, 2002, pp. 183–188.

Robert Höttger (*robert.hoettger@fh-dortmund.de*) started his Ph.D. studies in 2014 among a collaboration between Dortmund University of Applied Sciences and Arts and Technical University Dortmund, Germany. His research focuses on automotive parallel software engineering and adaptive system behavior using trace data with logical clocks. Especially feedback-aware requirements and tracing as well as DAG-algorithms form recent activities in the *AMALTHEA4public* project.

Lukas Krawczyk (*lukas.krawczyk@fh-dortmund.de*) recieved his M.Sc. degree in informatics and started his Ph.D. studies in 2014 among a collaboration between Dortmund University of Applied Sciences and Arts and University Bielefeld, Germany. His current research focuses on optimized deployment of software to embedded-systems hardware and many-core scheduling.

Burkhard Igel (*igel@fh-dortmund.de*) received his doctoral degree (Ph.D.) in computer science from University of Dortmund after studying electrical engineering and computer science. After more than 15 years working for Siemens Corporation now he is Professor at University of Applied Science and Arts in Dortmund and chairman of the supervisory board of itemis AG. His research area covers distributed and parallel computing as well as requirements engineering and model-based design.