Parallelizing highly complex engine management systems

Julian Kienberger[1] | Stefan Schmidhuber[2] | Christian Saad[1] | Stefan Kuntz[3] | Bernhard Bauer[1]

[1]Department of Computer Science, University of Augsburg, Augsburg, Germany
[2]Timing-Architects Embedded Systems GmbH, Regensburg, Germany
[3]Continental Automotive GmbH, Regensburg, Germany

Correspondence
Julian Kienberger, Department of Computer Science, University of Augsburg, Germany.
Email: kienberger@ds-lab.org

Summary

The automotive industry seeks to include more and more features in its vehicles. For this purpose, the necessary policy shift towards multi-core technology is in full swing. To eventually exploit the extra processing power, there is much additional effort needed for coping with the tremendously increased complexity. This is largely due to the elaborate parallelization process that spans a vast search space. Consequently, there is a strong need for innovative methods and appropriate tools for the migration of legacy single-core software. We use the results of a data dependency analysis performed on AUTOSAR system descriptions to determine advantageous partitions as well as initial task-to-core mappings. Afterwards, the extracted information serves as input for the simulation within a multi-core timing tool suite. Here, the initial solution is evaluated with respect to proper scheduling and metrics like cross-core communication rates, communication latencies, or core load distribution. A subsequent optimization process improves the initial solution and enables a comparative assessment. To demonstrate the benefit, we substantially expand a previous case study by applying our approach to two complex engine management systems and by showing the advantages compared to a parallelization process without preceding dependency analysis and initial partition/mapping suggestions.

KEYWORDS

multi-core, model-based design, data dependency analysis, timing simulation, design space exploration, semi-automated parallelization

1 | INTRODUCTION AND MOTIVATION

The demand for increasing performance applies in particular for embedded systems. The automotive domain can vade this trend as it perpetually attempts to enhance driving properties, extend infotainment features and improve security and safety characteristics of its vehicles. Therefore, challenges like parallelization and multi-core platforms have to be addressed to enable, eg, highly automated driving and car-to-x communication that pose new challenges for automotive software systems concerning aspects like dependability or cloud interaction.[1]

However, adding functionality—paralleled by car domains that evermore correlate—ramps up complexity as well as required processing performance.[2,3] Furthermore, there is a prevalent endeavor to save space and reduce weight by decreasing the number of "Electronic Control Units" (ECUs), which can be achieved by replacing them with distinctly less (but more powerful) "domain controllers".[4,5] These intentions drive the pursuit of finding a possibility for boosting the available computing performance to stay competitive.

In a conference speech by Mader,[6] it is projected that 10 times as much processing power as currently available will be needed in only 10 years. The rising demand has exceeded the capabilities of single-core technology whose processing power is almost completely exhausted and does not significantly increase anymore.[7–9] This can be ascribed to the problem that further raising the clock speed is unreasonable from an economical and technical point of view, because it inevitably leads to a disproportionate growth of CPU power consumption and an enormous rise of corresponding heat dissipation efforts.[9]

According to the current state of research, embedded architectures featuring multiple cores (or generally speaking "independent execution units"— IEUs[*]) are—predominantly considered—the only solution

[*]Though "multi-core" is the prevailing term when referring to such architectures, it is just one specific solution that is frequently used to vicariously represent the whole idea of parallel computing.[10] Used as more correct and generalized term, "IEU" encompasses processing units that are independent of each other within the system's scope (like a core, a processor, or an ECU).

having the potential to provide enough processing performance and therefore to meet recent challenges as well as to satisfy upcoming requirements. Thus, it is hardly surprising that they are becoming increasingly important.[11]

In the area of desktop computing, the transition to multi-core platforms started about 10 years ago, whereas the automotive sector was rather recently forced to start migrating its ECU software to pave the way for further technical advancement. As opposed to typical desktop computing, embedded automotive software faces hard real-time and strict data consistency requirements, heterogeneous target platforms with heavily limited resources as well as high demands concerning safety and redundancy.

Here, potentially gained computing power could be used in different ways—not only to save space and reduce weight by performing the same work on less ECUs (like previously mentioned). Further ideas are, eg, distinctly distributing certain functionalities of an application to different cores to prevent them from interfering with each other ("separation"), increasing a system's reliability (and therefore its safety) by performing additional calculations to raise a result's correctness probability by using different calculation methods ("diverse redundancy") or putting strongly connected software parts from different applications on one IEU ("pooling").

Unfortunately, current automotive software (operating systems as well as applications) was usually neither designed for being executed in parallel on the "function level" (as addressed in this article) nor on "application level". Its proper migration to multi-core systems is therefore a challenging task.[6, 12, 13] It involves a paradigm change, because aspects like "expensive" cross-core communication, synchronization overheads, shared resources, significance of memory location and the complex scheduling of software parts come into play when processing is distributed and sequential data consistency[†] has to be guaranteed.[4, 14, 15]

To achieve the latter without producing unnecessary interference (ie, overhead) among cores, it is crucial to appropriately determine the software's fragments in the first place ("partitioning") and to purposefully distribute them on the cores afterwards ("mapping"). Moreover, coordinating multiple cores to execute parts of a common application is tremendously increasing the complexity of software because of dependencies between separately processed but still interdependent data including problems like race conditions, dead locks, nondeterminism and insufficient load balancing (seeking equal workloads for each core).[16, 17] The complexity rise correlates with the amount of software parts, because the number of possibilities to distribute them on cores grows exponentially, which results in a tremendous solution space making an exhaustive design space exploration infeasible.

When looking at a typical combustion engine management system including up to 8000 of AUTOSAR's[‡] "Runnable Entities" (REs—atomic executable and schedulable units), it is obvious that multi-core approaches massively increase the internal complexity of ECUs and finding a favorable partition within such highly interconnected software is costly.[12, 19] This calls for new concepts to overcome emerging challenges, more specifically, finding suitable leverage points and heuristics for the process of migrating to parallelized versions of existing application software as well as supporting this process with tools that, on the one hand, automatize as much work as possible and, on the other hand, illustrate detected problems to the engineer in a meaningful way.[4, 6, 12, 20–22]

Apart from these new requirements emerging in the course of the "multi-core era", a well-known aspect remains crucial: The heavy focus on models and their active inclusion (ie, not only as supplement) in the whole process of software development and deployment is indispensable to keep an overview.[19, 23] This becomes clear when taking a closer look at the work process in the automotive sector, which is typically to a high degree based on the division of labor: For example, there are usually several hundred engineers working on an engine management system since its multitude of interconnections with other ECUs makes it exceedingly complex. Efficient collaborative working on such a system is hardly possible without proper organizational structures and suitable work items, like models.

In the following, we address the stated challenges by showing how our data dependency analysis approach (cf previous study by Kienberger et al[8]) can be carried forward by specifically covering the process of partitioning and mapping. The results of the partitioning and mapping process are verified by concretely measuring the added value achieved with this procedure after simulation and optimization.

2 | OVERVIEW AND PREREQUISITES

To alleviate the parallelized systems' complexity, we strive after supporting the goal-oriented migration of legacy ECU software to an expedient multi-core architecture. We focus on parallelizing a single application addressing "function parallelism" (also known as "task parallelism"). The proposed migration process is depicted in Figure 1.

The gray box indicates the scope of our work described in this article. Both activities on its left side are supported by our tool "AutoAnalyze," which is implemented as plug-in on the basis of the "Eclipse Modeling Framework" and the "Model Analysis Framework" within the "AUTOSAR Tool Platform"[§] (Artop).[25–27] The two activities on the right side of the gray box can be carried out within several third-party tools that provide simulation and optimization features for embedded multi-core systems. We use the "TA Tool Suite" in our case study.[28]

In general, there is indispensable information for each specific working step which is provided in Table 1. For each subsequent step, only the additional required mandatory or optional information is listed. The last column of the table has been adapted from the work of Sailer et al[29] and indicates whether the data exchange formats AUTOSAR, AMALTHEA or ASAM MDX can be used to store and exchange the respective information.

Data dependency analysis and validation: The basis of the approach is a data dependency analysis run on AUTOSAR models to detect, visualize, and solve potential conflicts related to a software's distributed execution on multiple cores:

[†]We define "data consistency" as both, given "functional coherence" (processed data has uniform age) and "stability" (steady data during processing).

[‡]The "AUTomotive Open System ARchitecture" standardizes "[...] an open software architecture for automotive electronic control units (ECUs)",[18] cf http://www.autosar.org.

[§]Artop facilitates the construction of AUTOSAR tools by serving as Eclipse infrastructure and virtually acting as "persistence layer" that enables common base functionality like easy access on AUTOSAR models that adhere to specific meta-model versions.[24]

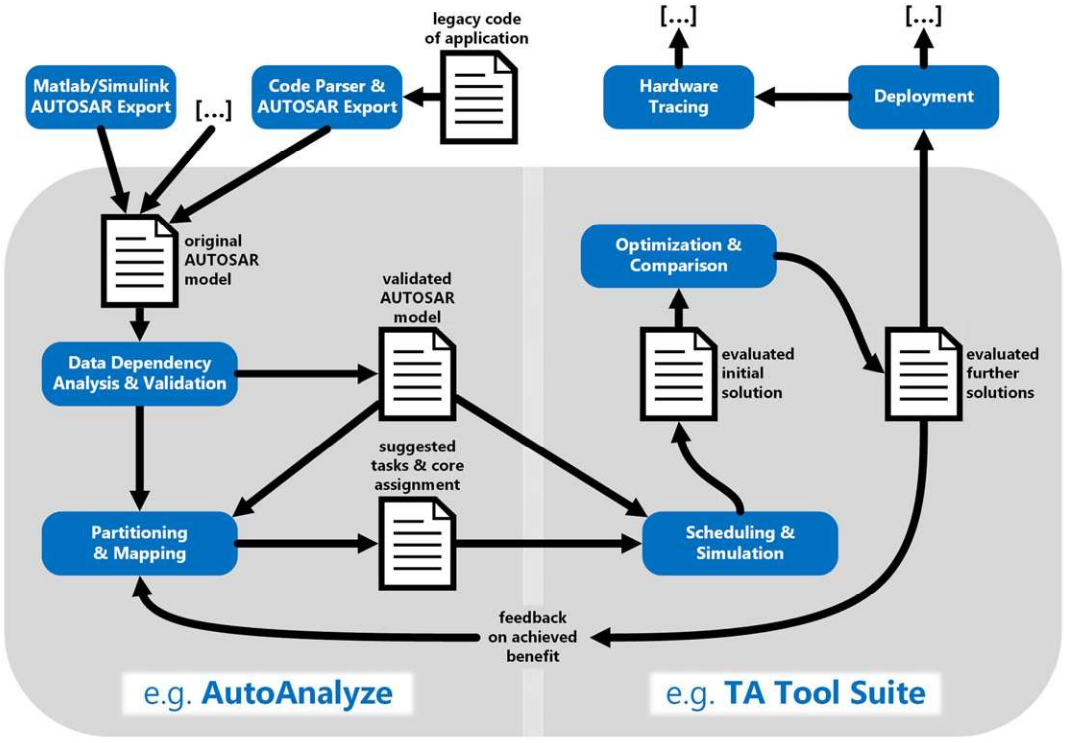
legacy code of application
[...]
[...]
Matlab/Simulink AUTOSAR Export
[...]
Code Parser & AUTOSAR Export
Hardware Tracing
Deployment
original AUTOSAR model
Optimization & Comparison
validated AUTOSAR model
Data Dependency Analysis & Validation
evaluated initial solution
evaluated further solutions
suggested tasks & core assignment
Partitioning & Mapping
Scheduling & Simulation
feedback on achieved benefit
e.g. AutoAnalyze
e.g. TA Tool Suite

FIGURE 1 Overview of the migration process

1. The analysis identifies a model's structural elements such as the aforementioned REs, their variable accesses, their recurrence (also called "triggering frequency"—TF— or "period"), their specific execution time, the software components (SWCs) containing the REs as well as already imposed timing constraints.
2. The gathered information is assessed, and potential conflicts regarding data consistency are determined.
3. Inconsistencies are addressed by an incremental (stepwise) application or modification of timing constraints on the lowest level (ie, on REs) for the purpose of achieving "multi-core robustness" in terms of data consistency.

This semiautomatic process provides the software with a consistent timing model to ensure the preservation of its original sequential (single-core) behavior on a multi-core platform.

Partitioning and mapping: This step's fundamental idea is to search on AUTOSAR's most fine-grained level (REs) for regions (sets of REs) with a relatively low coupling and to group them into tasks as means for creating a suitable partition as well as the subsequent task-to-core mapping. The immense search space can be remarkably reduced by providing a beneficial starting point for the simulation and optimization that are carried out to evaluate the initial solution and to search for further ones. This approach is based on techniques introduced in previous papers,[30–32] which were further developed in an approach by Götz et al.[33]

Since its initial proposition (in Kienberger et al[8]), the partitioning algorithm was extended to cope with highly complex models:

- A configurable search tolerance can be used to loosen the rather strict criteria for low-coupled regions, so that RE sets, which violate the demands to a certain tolerable extent, are not discarded right from the outset.
- The relevance of the connection between two REs is calculated to determine concrete pair-wise dependency weights depending on the number of variable accesses between two REs as well as their respective recurrence.
- Based on experience, analyzing a whole model often unnecessarily increases the search effort while simultaneously shrinking the result set. This effect can be attributed to the high degree of interconnection in large models which exacerbates partitioning attempts. Thus, different splitting strategies are provided, eg, dividing the model into parts with uniform recurrences before running searches on each of them.
- As these parts often remain highly complex, the search gradually ignores pair-wise dependencies below a growing threshold until a sufficient number of regions is found. This "relevance partitioning" roughly corresponds to the "Multi-Level Partitioning" approach of first "coarsening," then clustering and afterwards restoring a graph.[34]

The succeeding mapping algorithm is highly adjustable to meet the specific hardware platform's requirements. As demonstrated in the case study (Section 4), achieving a certain goal is possible by modifying the preferences when arranging the tasks and mapping them to a specific number of cores, eg, the preferred granularity of the regions, a concrete load balancing strategy or enforcing certain regions to be assigned to the same core (via "pairing constraints").

However, the quality of a calculated partition and an according mapping can hardly be objectively assessed. This is because optima are typically either not recognizable as such or are simply unknown and goals for parallelization efforts are often contradictory. For example, pooling strongly-connected software parts (low synchronization overhead), distinctly separating functionalities that should not interfere

TABLE 1 Lineup of required information for the specific working steps

	Mandatory	Optional	Exchange format
Data Dependency Analysis & Validation	• shared data definitions • REs including data accesses • recurrences of REs	• SWCs and compositions • timing constraints (ie, EOCs and ACs)	• AUTOSAR • AMALTHEA •ASAM MDX
Partitioning & Mapping	• basic hardware model (ie, number of cores)	• basic OS model (eg, scheduling algorithm)	• AUTOSAR • AMALTHEA
Scheduling & Simulation	• definitions of tasks and Interrupt Service Routines (ISRs) with RE call sequence	• precise hardware and OS model	• AUTOSAR (only mandatory information)
	• basic OS model	• precise RE runtimes (ie, runtime distribution)	• AMALTHEA
	• OS configuration (eg, task and ISR priorities and preemptability)	• precise task/ISR activation pattern (eg, jitter and sporadic activations)	
	• timing requirements (ie, task deadlines)	• precise task call graphs (ie, branches)	
Optimization & Comparison		• Affinity Constraints (eg, RE pairing or separation)	• AMALTHEA

Abbreviations: AC, AgeConstraints; EOC, Execution Order Constraints; ISR, Interrupt Service Routines; REs, Runnable Entities; SWCs, Software Components.

(support safety) and evenly spreading computational cost (proper load balancing) are usually not simultaneously achievable. Thus, obtained solutions mostly constitute a trade-off between a few personally prioritized goals.

Like previously mentioned, we are seeking a pooling-oriented partition together with a mapping that enables satisfactory load balancing. Of course, there are numerous other factors worth taking into consideration, eg, if the target platform's processor is “homogeneous” concerning equally equipped cores and how many of them are available.

In the end, this step's purpose is to provide an advantageous starting point (“initial solution”), which is effectively supporting the following simulation and optimization to find appropriate further solutions in an adequate amount of time.

Scheduling and simulation: The subsequent step employs a third-party simulation tool to evaluate the usefulness of the provided initial solution using various timing and performance metrics, eg, task response times. Regardless of what product is employed, some preparations have to be made before such a simulation can deliver expressive results:

1. Importing the validated AUTOSAR model together with the calculated partition and mapping from the previous working step (initial solution).
2. Setting up the underlying hardware model, eg, a generic processor model or a detailed automotive microcontroller simulation model.
3. Setting up the underlying operating system model, eg, how the cores are managed by the operation system (global/local scheduling, online/offline scheduling, and the scheduling algorithm).
4. Adding basic timing requirements (eg, task deadlines) to later allow a general classification into valid and invalid solutions.

Optimization and comparison: The aforementioned initial solution serves as starting point for further improvements by the optimization step. Alternative solutions are hereby created by systematically modifying the initial solution in an iterative process. To steer the optimization process, each alternative solution is simulated to evaluate the improvement compared to the initial solution with respect to timing and performance criteria. This optimization process can be carried out either manually or with tooling support.

A focus on “weak spots” discovered by means of interpreting the simulation results can act as beneficial leverage point for model changes, eg, splitting up “chunky” tasks that hamper proper load balancing or enforcing two heavily communicating tasks to be mapped to the same core. Having found better solutions in terms of certain characteristics—like reduced cross-core communication or a shorter overall cycle time—allows to “close” the round-trip engineering circle by providing customized data on advantageous model-specific search parameters for the "Partitioning and Mapping" step. This iterative proceeding seems sensible to effectively broaden the search span by means of a fresh "solution seed."

Running a timing simulation on highly complex models can take a lot of time, but eventually yields informative findings about a solution's general validity, occurring latencies, overhead caused by necessary synchronization, or a basic statements about a software's overall degree of potential parallelization (indicating possible speed-up). Optimizations for practical models typically require the simulation of several tens of thousands alternative solutions. As a consequence, the required time for optimization is mainly dictated by the simulation performance. Generally speaking, it can be stated that a "good" preceding partitioning and mapping facilitates the optimization step considerably which is particularly important when being confronted with cross-core communication as a substantial new resource bottleneck.[4]

3 | APPROACH IN DETAIL

In the following, we describe the introduced migration process (cf Figure 1) in detail and start by explaining how models are prepared for distributed execution on several cores.

3.1 | Data dependency analysis and validation

Like previously stated, the central task is to determine a timing model, so that an embedded application's original behavior is retained. In this context, conflicts can occur when functional blocks—originating from legacy (single-core) software—are processed in parallel instead of well-matched and rigidly consecutive like before. This poses a threat to data consistency, which we previously defined as stability (steady signals or values over a certain period of time) being paralleled by coherency (signals or values with uniform data age).

Possible conflicts are, eg, data not being available in time or data being read inconsistently. Of course, consistency conflicts like this can occur on single-core platforms too, but multi-core systems are more prone to "evoke" them because here, it is—due to concurrency—significantly harder to maintain consistency.

Therefore, it is inevitable to address any potentially unintentional behavior within a system. The challenge consists of the already mentioned complexity rise caused by the exponentially growing number of possibilities to distribute tasks on cores, which leads—together with scheduling—to a tremendous amount of possible execution sequences including many adverse (ie, conflicting) ones.

To assure that software will work properly regardless of a specific mapping of tasks to cores, it is necessary to completely preclude unintended behavior, which is accomplished by adding timing constraints to the model or by modifying existing ones.

Our approach is grounded on the following principles:

- Bottom-up: Both, analysis and validation are conducted on the most detailed (lowest) level, which corresponds to REs in AUTOSAR. A top-down approach would be barely appropriable in this case as ECU software is—due to competitive pressure—rather continuously revised and augmented than created "from scratch". Additionally, more general views on a system can nevertheless be built by deriving from collected low-level data.
- Incremental: Validation is run stepwise instead of all at once to prevent the creation of "fresh" conflicts by overlapping constraint scopes and to avoid indiscriminately calculating all possibilities at the very start which would require disproportionate computing power (and considerably more time).
- Minimal: For the purpose of not needlessly restricting the degree of freedom for subsequent steps (and thus not to unintentionally exclude promising solutions), only a minimum set of constraints is imposed.

Complying with these principles, a static data dependency analysis is performed directly on AUTOSAR models (see Kienberger et al[8] for details).

As a start, parsing the AUTOSAR model provides the information foundation necessary for following steps: The SWCs are AUTOSAR's central structural elements as their "Internal Behavior" involves the REs, the intra-component communication (between several REs) and the inter-component communication (between an ECU's different SWCs). As each RE can be multiply instantiated (eg, 4 wheel speed sensors of a car), every RE instance (REI)[¶] has its own data dependencies, which each emerge from the interaction between 2 REIs. In AUTOSAR, 7 kinds of variable accesses are employed for "local" (intra-SWC) access as well as for communication crossing SWC borders. All of them are considered.

Moreover, AUTOSAR's timing constraints are identified: Currently, we deploy 2 out of the 7 existing constraints: "Execution Order Constraints" (EOCs) and "AgeConstraints" (ACs).[23,35] The former are "[...] used to specify the order of execution of ExecutableEntities" (ie, specify a fixed order for multiple REs) and the latter "[...] to specify a minimum and maximum age that is tolerated when a variable data prototype is received" (ie, determine the tolerated data age of a read variable).[35]

EOCs are used to predetermine a rigid execution order between 2 or more REs. This is appropriate when they are logically (and semantically) linked like in a classical "sensor-controller-actor system" where sensors transmit measurement data to a controller which determines a suitable action that is afterwards carried out by associated actors, eg, brakes in an antilock braking system. In contrast, ACs resolve potential inconsistencies via tagging a possibly conflicting dependency as unproblematic by allowing that certain accessed data comes from a previous "computing cycle"[‖]. This is feasible if the reading RE does not imperatively need current data to work properly, eg, a speedometer that is not able (and is not intended) to react within milliseconds because of the speedometer needle's inertia.

Taking all this information into consideration allows to map them on a directed graph illustrating the data-flow by means of nodes that

[¶]This is not an official but an implicit AUTOSAR element. The existence of RE instances arises implicitly from the component structure where the same RE may appear in different contexts.
[‖]We define a computing cycle as the time elapsed between 2 events that involve periodically activated tasks being guided by the slowest (least triggered) task occurring.

represent the REs and edges standing for the variable accesses semantically connecting them.

For the second step, gathered information is employed so that sets of node neighborhood for the access on a specific variable can be derived from the graph (ie, successor and predecessor relations between REIs). These sets are expedient to identify possible execution sequences and to accordingly classify the dependencies, so that possible inconsistencies are found. The latter are represented by every contingency of unintended consuming prior to producing certain data within the scope of one computing cycle.

In addition, existing timing constraints are checked for correctness (validity). EOCs should only be set for REs with uniform recurrence, because when dealing with ECU software, static scheduling is prevalent. Thus, a once found execution order does not change anymore. As opposed to this, EOCs are hardly suitable for REs with diverging recurrences, because their execution order within a certain computing cycle can alter, eg, when a "consuming RE" is executed more often than a "producing RE". In such a case, an EOC requiring the producing RE to be computed at first (within a computing cycle) may cause additional latency when the consuming RE is triggered (and finished) earlier than the producing one. Here, ACs are usually the more suitable means.

Besides simply absent constraints, typical fault cases are, eg, EOCs imposed on REs with divergent recurrences, EOCs that contradict each other by forming a cycle (the easiest case is "(A before B) && (B before A)") or "insufficient" ACs that merely allow a smaller data age than effectively arising.

Via this validation process, "multi-core robustness" can be achieved by means of resolving the model's potential conflicts through preventing every unintentional consuming before producing. This is done by imposing a minimal amount of timing constraints to reduce all possible execution sequences to a set which supplies every REI early enough with its necessary input data.

As EOCs do distinctly narrow the "degree of freedom" for mapping the REs (grouped as tasks) to cores, the potential for parallel execution is strongly reduced when a model is heavily order-constrained. Parallelism can even be fully thwarted when the EOCs' combination enforce a "single-chain" execution order. Thus, as little EOCs as feasible are imposed while preferably setting them in a local scope (eg, being only valid within one SWC). In the case of imposing constraints on dependencies across SWC borders, ACs come in handy as their (global) impact is less limiting and they do not decrease the number of possible execution orders (which is advantageous for multi-core use cases). However, ACs and EOCs are not mutually exclusive: depending on the specific situation, combining them can be very expedient. This is particularly the case if a certain variable is used for different purposes, eg, the value of a current wheel speed is frequently read by an antilock braking system but usually only seldom by the speedometer.

Once all potential conflicts, including those that influence parallelization behavior, are wiped out, a system's validity—with regard to data age—can be ensured. Now, the corresponding model is ready to be split up safely into functional blocks that can be mapped on different cores.

3.2 | Partitioning and mapping

Once the consistency threats have been identified and solved with the help of constraints, the next logical step is to figure out how the software can be split up and distributed in an expedient way: At first, "partitioning" breaks up a model into sets of REs according to a given objective, then the succeeding "mapping" means to determine concrete tasks within the obtained partition and to assign them to specific cores. Afterwards, their actual execution can be scheduled.

3.2.1 | Overview

As a matter of fact, there is no universal approach to find a suitable partition or mapping, and it is difficult to assess whether a specific solution will satisfy certain properties. Therefore, it is essential to thoroughly consider the desired aspects of the target system and its according objectives in advance. This is usually done with respect to definite goals like reaching a preferably low coupling rate between the tasks and therefore rather little necessary synchronization as well as communication ("pooling"), ensuring the adherence to safety requirements like distinctly separating highly critical tasks or preserving the processing of logically related software parts on the same core, eg, REs contributing to one common function.

Since there are countless possibilities to partition a model, determining an optimal partition according to specific goals is classed as "NP-hard problem".[36] Moreover, a search for advantageous task-to-core mappings involves traversing an overwhelmingly huge solution space as the number of mapping possibilities grows exponentially according to the amount of given tasks. Collectively, both activities maybe pose the hardest challenge when trying to build a powerful and streamlined multi-core system.

An easy sample calculation shows how the search space quickly escalates even for small examples: The "Brake-by-Wire" application from projects "TIMMO"[**] and "TIMMO-2-USE"[††] consists of clearly organized 18 REs.[37, 38]

Assuming that each RE is supposed to be mapped separately on 1 of 3 available cores, there are about 387 million (3^{18}) different ways to do so. After choosing one of these distribution solutions, there are again many possible execution sequences: there are over 6 quadrillion ("18 factorial") sequences for executing all REs successively on 1 core. And there are a lot (exponentially) more options in a multi-core setting, because most REs can theoretically be processed in parallel (fully or partially overlapping). Generally speaking, every random set of REs can be simultaneously executed as long as it is valid regarding the absence of 2 REs being interconnected by an EOC. The exact count of possibilities depends on the number of available cores (defining the maximum set size), the number of tasks encapsulating the REs and possibly given minimum requirements for load balancing (together with execution times).

Therefore, we aim to reduce the number of possibilities to consider by first providing a beneficial initial partition and secondly—based on this starting point—an advantageous initial mapping, which increases

[**]The project "TIMing MOdel" developed "[...] a common, standardized infrastructure for the handling of timing information during the design of embedded real-time systems in the automotive industry."
[††]The project "TIMing MOdel - TOols, algorithms, languages, methodology, and USE cases" provides "[...] tools, algorithms, languages, methodology, and use cases for dealing with timing requirements and properties for timing analyses during the development of distributed embedded automotive systems".

the efficiency of the following scheduling, simulation, and optimization. As the partition is created with respect to imposed constraints and existing dependencies, the subsequent computational effort is limited to a "corridor" of preferably promising solutions. Since it cannot be guaranteed that proper paths are not discarded, this process should be repeated to ensure a balance between searching deeply and broadly.

Without a given partition and if no further knowledge of the system is available, a simulation tool would be forced to draw on simple strategies to obtain initial tasks (like employed in approaches by Long et al[39] or Monot et al[40]), eg, preferably encapsulating REs with equal recurrences and therefore creating homogeneous and easily relocatable tasks. Such regions are particularly suitable for being executed on a common core, so that the duration of one "computational iteration" on this core is not needlessly delayed due to REs' recurrences that are cumbersome to reconcile.

However, such a partition can be very adverse too, especially when load balancing is hampered by strongly differing task sizes or when—as it is almost always the case—cross-core communication is an issue and heavily connected REs are not assigned to the same core. According to our experience, this holds particularly true for highly complex models like those used in the case study (cf Section 4).

3.2.2 | Partitioning

As stated in Section 2, "low coupling" (corresponds to "pooling") acts as standard partitioning objective. It is determined by counting the dependencies that cross region borders within a certain partition, ie, data accesses that are "broken" by assigning the involved REs to different regions. Restoring these dependencies (preserving their function) requires additional synchronization effort, because at scheduling, the execution of the respective REs has to be coordinated according to their specific cross-linking. Furthermore, we start from the premise that a target system's processor has "homogeneous" (ie, equally equipped) cores.

This concept is realized by the "Single Entry Region Analysis" algorithm that searches for virtually isolated RE sets within the model. They are characterized by a common starting point (the "entry node") and by not having any dependencies to outside nodes before a common end point (a "merger node") "closes" the region. Details on the algorithm, an exact definition, its origin and implementation are available in the previous dissemination by Kienberger et al.[8]

As broached in Section 2, the algorithm used to be not productive enough for highly complex models as its strict rules were not defined for heavily interconnected graphs. To make it applicable to all kinds of models, we purposefully extended it to meet the emerging requirements:

Search tolerance: Being configurable according to the specific model's complexity, the algorithm accepts a certain number of "isolation violations" without discarding the identified RE set. This is useful to perform a search that takes the average node degree (ie, the number of dependencies per node) into consideration, making it possible to find "hot spots" even in dense graphs. Based on experience, it is—in most cases—relatively easy to detect a sensible upper limit for this tolerance, because found groups beyond this "turning point" are—often out of a sudden—bulky and evidently not significant anymore.

Dependency weights: Treating the connection between all node pairs equally is obviously not expedient when having to decide which one to "break" while trying to form RE sets. Therefore, we calculate weights for the connection degree of every connected node/RE pair using the information usually available in AUTOSAR models: the REs' period and the number of dependencies (variable accesses) connecting them. The weight value rises according to decreasing periods (corresponds to higher triggering frequencies) and a rising number of dependencies. The according formula is: $weight = (1/\text{periodA} + 1/\text{periodB}) * dependencies$.
It is easily adaptable if further information (like the amount of transferred data of a specific variable access) is given and serves as basis for the "relevance partitioning."

Relevance partitioning: It is in most cases rather fruitless to pursue simple partitioning approaches like, eg, "Sparsest Cut" which repeatedly cuts a graph into 2 (roughly) equal-sized pieces.[41] This is due to strongly differing model structures which are usually not suitable for being strictly divided into 2^x parts.
Thus, we use a more sophisticated approach vaguely resting on "Multi-Level Partitioning", which better adapts to specific model structures.[34] "Multi-Level Partitioning" reduces a graph via "edge contraction" ("coarsening") to cluster and afterwards restore it. However, we do not "erase" nodes/edges but gradually increase the relevance threshold for dependencies taken into consideration by the search until the graph is "manageable" enough to find appropriate RE sets.

Splitting strategy: As previously mentioned, it is basically advantageous to identify groups whose REs have a uniform recurrence. This can be achieved by different strategies:

- "Split, then analyze": In our experience, building subgraphs that consist of uniformly triggered REs and then running partitioning searches on each of them, has produced the most valuable results for highly complex models. In addition, the overall search effort is remarkably reduced.
- "Analyze, then split according to periods": This strategy takes the graph as is and assumes that the search finds sufficient groups, which can afterwards be split according to the number of diverging RE periods occurring. This is rather suitable for small heterogeneous or for huge but loosely connected models.
- "Do not split, discard mixed regions": Here, identified regions are discarded if they do contain REs with diverging periods. This can be useful for models with a small amount of different periods that are nevertheless relatively complex.
- "Do not split, keep mixed regions": As pretty simple strategy, this approach is rather used as starting point to gain an insight into the possible partitioning degree of a model in general.

Because of the dynamic adaption (eg, automatically rising the tolerance and dependency threshold until a certain coverage rate is reached), the algorithm can cope with models of any size and complexity. However, this does not mean that every application can be efficiently parallelized, but it is almost always possible to identify a proper partition according to circumstances.

3.2.3 | Mapping

The partitioning algorithm determines preferably large RE sets, which can—hierarchically structured—contain smaller ones. This is done deliberately to maintain RE sets of every size and therefore to retain all granularities for a later mapping of tasks to cores.

As the count for both the mapping of tasks to cores and the possible execution sequences strongly depends on the initial number of tasks, seeking to prevent a too fine-grained partition (many small tasks) is a reasonable trade-off because although fine granularities provide more flexibility, they involve much more effort to distribute and are harder to synchronize.

As opposed to this, a coarse-grained partition "[...] can more easily result in an improvement" and thus seems appropriate as a first step.[42] However, having only very few large tasks can make it difficult to distribute them properly on different cores without again causing overhead for additional synchronization, eg, if being forced to map 2 intensively connected partitions on different cores or when trying to achieve even workloads for cores (load balancing).

Of course, the latter—as well as the whole mapping process—is only possible when the essential features of the target hardware are known (number and homogeneity of cores).

Our principles remain pooling and load balancing, for which we need to sensibly choose a suitable size for each available RE set to find the most convenient mapping. This is due to the fact that a partition does usually not contain groups with uniform size and therefore following a rather coarse-grained approach should not lead to a clumsy method like "streamlining" the partition by reducing large groups.

Eventually, we take the following aspects into consideration when creating a mapping:

- Number of cores (IEUs) being available on the target platform
- Task clustering strategy: preferred relative regions size (if they are nested) and handling of remaining REs (eg, a new task for each or create clusters according to periods)
- Expected utilization: In case of available execution times (eg, specified by an SWC's 'ResourceConsumption' property in AUTOSAR), an RE's expected workload is computed according to the formula $utilization = (reInstances * reExecutionTime)/rePeriod$.
- Distribution: Taking heed of the calculated tasks' workload, their assignment to cores can be done by algorithms drawing on well-known patterns like "bin packing," "round robin" or—for small models—"exhaustive search space exploration".

For our case study (Section 4), we use an exporter tool that creates a CSV file comprising these aspects. Additionally, it produces basic timing requirements (task deadlines according to given periods) and sets task priorities in order to support the succeeding simulation.

3.3 | Scheduling and simulation

In pursuance of evaluating the initial partitioning and mapping solution, we use a discrete-event simulation tool to conduct the evaluation regarding valid scheduling (ie, fulfillment of task deadlines), specific reaction times of critical execution paths, communication overhead, memory consumption, and core load distribution. Compared to analytical methods, simulation techniques only yield approximated timing metrics like task response times.[43] However, analytical methods usually provide very pessimistic estimations resulting in, eg, overestimated worst-case response times. On the contrary, simulation techniques allow more realistic typical case approximations. Additionally, the proposed overall process as presented in Section 2 incorporates hardware measurements of simulated solutions after deployment as a final timing verification step.

3.3.1 | Discrete-event simulation

Discrete-event simulators use the fact that in between two consecutive events, a system cannot change its states.[44] Consequently, only the discrete points in time where state transitions occur are simulated. All state transitions which occur during simulation together with the respective time stamps are recorded in a Best Trace Format (BTF) trace.[45] The simulator operates on the AUTOSAR-compliant and AMALTHEA-compliant timing model,[46] which consists of abstract descriptions of the application software, hardware, operating system, runtime environment, and environment (ie, external stimuli).

As already broached in Section 2, the simulation requires certain information. For example, the operating system model must include a specification of the schedulers which manage the execution of tasks and Interrupt Service Routines (ISRs) on the respective cores. Moreover, the used scheduling algorithms and the scheduling-relevant properties of tasks and ISRs (like priorities) have to be provided. While simulation is already possible with basic hardware model information like the number of cores together with their clock frequency and instructions per cycle, detailed vendor-specific processor models greatly improve simulation precision. When the exact memory topology and behavior descriptions including memory modules, caches, bus networks or crossbars are provided, memory access times, cross-core communication delays as well as contention effects can be considered in a simulation.

3.3.2 | Timing and performance metrics

After the application of statistical estimators to the resulting event trace of a discrete-event simulation, various timing and performance metrics can be calculated. In the following, the most important metrics for the optimization step are introduced.

- Maximum Normalized Response Time (mNRT): The mNRT metric quantifies the relative worst-case response time which occurred in a simulation.[46] "Relative" means that the response times of each task have been normalized with respect to their relative deadline. If all deadlines are met, the mNRT is smaller than 1 whereas greater values denote deadline violations during simulation.
- Inter-Core Communication Rate (ICCrate): The ICCrate metric quantifies the amount of data in bits per time unit which is exchanged between the cores. It is an indicator for the expected cross-core communication overhead.
- CPU Load (CPULoad): The CPULoad metric quantifies the average load of a processor or individual core over the complete timespan covered by the simulation.
- Maximum Load Distance (MaxLoadDist): The MaxLoadDist metric quantifies to what extent the overall load is equally distributed to

the individual cores. It is the maximum of the absolute differences between the CPULoad values of each core and the per-core CPU-Load value obtained by dividing the overall load by the number of cores.

- Buffer Size (BufferSize): The BufferSize metric quantifies the additional required memory in bits needed to enforce data consistency by a buffering technique.[47]
- Event-Chain Duration (ECDuration): "Event-Chains" as defined in the AUTOSAR Timing Extensions[35] connect arbitrary subsequent events like the activation of a task, the termination of an RE or write accesses to a specific variable. An Event-Chain consists at least of a stimulus and response event but can also be further detailed by segments and strands. The ECDuration metric quantifies the timespan between a stimulus and response event of an Event-Chain. Thus, the reaction time of critical processing paths in the system, eg, across multiple REs of different tasks can be evaluated.

3.4 | Optimization

To create alternative solutions as a result of the optimization, tasks might be remapped to the different cores, existing tasks might be split into smaller ones and the variables will be mapped to the different memory modules. The underlying problem of repartitioning and remapping the software is equal to the Bin Packing Problem, which is known to be NP-hard.[48] Consequently an exhaustive search, ie, evaluating every possible alternative solution, is not an option in practice. As genetic algorithms are one feasible way to handle such problems,[49,50] we choose to integrate a genetic optimization tool in our workflow.

3.4.1 | Genetic Algorithms

These algorithms simulate natural selection and evolution in an iterative approach[51] and operate on a set of alternative solutions. This set of solutions ("population") is modified within each iteration ("generation") of the algorithm. Every genetic algorithm consists of the following steps:

1. Create initial population: The initial population consists of randomly created solutions, eg, using a uniform distribution.
2. Fitness assignment: A scalar fitness value is assigned to every solution of the population, which is used to quantify the quality of a solution compared to another one.
3. Selection: Solutions are sorted by descending fitness values. The best ones according to fitness are kept while the remaining ones are discarded and removed from the population.
4. Evaluate stop criterion: The algorithm terminates when the stop criterion is fulfilled. This can either be the case after a predefined number of solutions has been created or after a specific amount of generations. Another possibility is to stop after a stagnation threshold has been reached, eg, when the best solution did not improve for a certain amount of generations.
5. Perform variation: Mutation and crossover techniques are used to create new solutions. For the former, one or more properties of an existing solution are randomly modified to create a new solution. For the latter, the properties of two or more solutions are combined to create one or several new ones.

These steps only define the generic framework of genetic algorithms. In our case, the discrete-event simulator presented in section 3.3.1 is used to evaluate every created solution and provide the required metrics for fitness assignment. Regarding further implementation details of the used genetic algorithm, we refer the reader to the work of Schmidhuber et al.[46]

3.4.2 | Optimization Parameters

To configure a specific optimization run, configuration parameters have to be provided for each of the aforementioned steps. They are as follows:

- Per-Solution Simulation Time: This is the timespan covered in the simulation for each created solution during optimization.
- Configuration of the Fitness Function: Several timing and performance metrics as introduced in section 3.3.2 are aggregated together into a scalar fitness value for each solution by using a modified euclidean norm.[46, 52] For each incorporated metric, a weight factor as well as a lower and upper limit for normalization has to be provided.
- Population Size: This parameter defines the number of solutions created in the initial population as well of the number of new solutions which are created during each iteration of the algorithm.
- Selection Size: The selection size is the amount of best solutions according to fitness which are taken over into the next iteration. Those selected solutions are also used to create new solutions by means of mutation and crossover.
- Stop Criterion: For the stop criterion, the minimum and maximum number of solutions and/or generations are specified. Moreover, the stagnation threshold is configured, ie, the algorithm stops if the best solution according to fitness did not improve over a given amount of iterations. All these criteria are evaluated simultaneously, which means that all minimum requirements (eg, minimum number of generations) and at least 1 maximum requirement (eg, stagnation threshold) have to be fulfilled to result in the optimization's termination.

3.4.3 | Design Modifications

Design modifications denote different categories of architecture changes which are applied to an existing solution to create new alternative solutions during variation. It is hereby possible to perform multiple design modifications at once. For certain categories, design constraints that restrict the respective degree of freedom can be stated as well. If specified, such design constraints will be fulfilled by every single solution produced during optimization. One example for such constraints is the requirement to map certain tasks to different cores, eg, due to safety requirements, which demand spatial separation of the respective functionality.

- Process Mapping: Process mapping results in the remapping of tasks or ISRs to different cores.
- Task Splitting: Tasks are split into 2 or more tasks which are then mapped to separate cores. The split tasks are triggered one after another to maintain the original RE execution order.
- Data Mapping: Data mapping allows the optimizer to change the variable-to-memory mapping.

- Periodic Offset Assignment: This modification varies the offset of periodically activated tasks.

4 | CASE STUDY

In this section, we illustrate the case study that we conducted to demonstrate the effectiveness of the introduced migration process (cf in Figure 1).

4.1 | Overview and Goal

To demonstrate the benefit, we substantially expand a previous case study by Kienberger et al[53] by applying our approach to two complex engine management systems and by showing in-depth arising advantages compared to a parallelization process without preceding dependency analysis and initial partition/mapping suggestions.

We state the following hypothesis: An optimization algorithm will yield significantly better results compared to a predefined initial solution in the same given time if a preceding dependency analysis and the resulting initial partitioning/mapping are used as a starting point.

For each of the two engine management systems, the following experiment was conducted:

1. Definition of a "reference solution" (ie, reasonable initial solution) in terms of partitioning, mapping, and OS configuration.
2. Optimization I: Creating alternative solutions using "TA Optimizer" (TA-Opt).[28]
3. Optimization II: Using "AutoAnalyze" (AA) to provide the starting point for subsequently creating alternative solutions using "TA Optimizer".
4. Comparing the relative improvements to the reference solution yielded by Optimization I/II.

4.2 | Setup

As mentioned before, we use the following two complex AUTOSAR models:

The first one is a part of a huge real-world engine management system from Continental ("Conti-EMS"), which consists of 178 SWCs including 552 REs with 20 different recurrences, 11 460 variables/signals and 45 399 data dependencies (each arising from a write and according read access on a specific variable).

The second one is a "[...] a full blown performance model of a modern engine management system" ("Bosch-EMS"),[54] which is publicly available as "AMALTHEA"[‡‡] model for the "FMTV Verification Challenge" of the "WATERS" workshop.[56, 57] It comprises 1250 REs with 11 different recurrences, 9983 variables/signals, and 5195 data dependencies. We converted it to AUTOSAR for being able to apply our working steps.

In the following, we enumerate all configurations we have made for carrying out the experiment steps mentioned before:

Reference Solution: The following adjustments have been made to the Conti-EMS and the Bosch-EMS model to create the aforementioned reference solutions:

[‡‡]"AMALTHEA is an open source tool platform for engineering embedded multi-core and many-core software systems."[55]

- Conti-EMS. Since the AUTOSAR description only contains the SWCs, REs, and variables, we conducted the following initial partitioning, mapping and OS configuration. Moreover, we used the "Infineon AURIX TC27x"[58] simulation model as hardware description. The processor's architecture is heterogeneous. Two out of its three cores process instructions faster (two "performance cores", one "efficiency core"). Moreover, two of its cores are capable of lockstep execution. The clock frequency of the 3 cores is set to 200 MHz.
  For the partitioning, 1 task per TF has been created, which executes all REs belonging to it. The mapping was done by assigning the tasks to cores using a typical separation scheme for different TFs. In terms of OS configuration, each core is managed by one AUTOSAR scheduler. Priorities have been assigned using the "rate monotonic scheme"[59] (ie, the shorter the TF, the higher the priority) and each task was configured to be fully preemptive.
- Bosch-EMS. The existing configuration is used as reference solution, as the Bosch-EMS system already contains the complete partitioning and mapping information as well as the operating system configuration. However, we increased the clock frequency of all four cores from 200 MHz to 1 GHz to prevent scheduling errors as our analysis at experiment setup had shown that the system is not schedulable with 200 MHz.

Experiment Configuration: This case consists of eight experiments in total. Their configurations are itemized in Table 2. All experiments consist of two optimization runs. One only with TA Optimizer (TA-Opt) and one where AutoAnalyze is used additionally (AA + TA-Opt). Some general configuration parameters for simulation and optimization are equal for all experiments: We set the per-solution simulation time to 5 seconds, while the population size was set to create 32 solution for the initial population and 16 new solutions for each subsequent iteration. The selection size was set to keep the 16 best solutions according to fitness and discard the remaining ones. For the stop criterion, we used a fixed value of 256 alternative solutions.

Optimization Goals: All of the four optimization goals are detailed in Table 3 and denote the minimization of one single criterion or the simultaneous minimization of multiple criteria, respectively.

AutoAnalyze Configuration: The two different AutoAnalyze configurations—as mentioned in Table 2—are stated in the following:

- AA-1: partitioning: rather small groups, relatively high search tolerances (20 for Bosch-EMS, 30 for Conti-EMS); mapping: bin packing, preferably equal distribution (totally equal for Bosch-EMS, roughly equal for Conti-EMS)
- AA-2: partitioning: rather large groups, relatively high search tolerances (10/20 for Bosch-EMS, 30 for Conti-EMS); mapping: bin packing, preferably equal distribution (totally equal for Bosch-EMS, rather unbalanced for Conti-EMS)

TA Optimizer Configuration: There are four different TA Optimizer configurations used in this case study, which are distinct from each other regarding the applied design modifications as introduced in section 3.4.3.

TABLE 2 Lineup of the different experiments performed within the scope of this case study

Experiment	Model	Optimization goal	AA config (AA + TA-Opt)	TA-Opt config (TA-Opt)	TA-Opt config (AA + TA-Opt)
Exp-1	Conti-EMS	Goal-1a	AA-1	TAOPT-1	TAOPT-2
Exp-2	Conti-EMS	Goal-1b	AA-1	TAOPT-1	TAOPT-2
Exp-3	Conti-EMS	Goal-1c	AA-1	TAOPT-1	TAOPT-2
Exp-4	Conti-EMS	Goal-1a	AA-2	TAOPT-1	TAOPT-2
Exp-5	Conti-EMS	Goal-1b	AA-2	TAOPT-1	TAOPT-2
Exp-6	Conti-EMS	Goal-1c	AA-2	TAOPT-1	TAOPT-2
Exp-7	Bosch-EMS	Goal-2	AA-1	TAOPT-3	TAOPT-4
Exp-8	Bosch-EMS	Goal-2	AA-2	TAOPT-3	TAOPT-4

TABLE 3 Lineup of the different optimization goals stated in Table 2

Optimization goal	Metric	Weight	Lower limit	Upper limit
Goal-1a	**ICCrate**	**1**	**0**	**24.34 MBit/s**
Goal-1b	ICCrate	1	0	24.34 MBit/s
	MaxLoadDist	1	0	20 %
Goal-1c	mNRT	10	0	2
	BufferSize	5	0	17.88 kB
	MaxLoadDist	5	0	20 %
Goal-2	ECDuration (EffectChain1)	1	0	94.60 ms
	ECDuration (EffectChain2)	1	0	601.3 ms
	ECDuration (EffectChain3)	1	0	12.50 ms

TABLE 4 Results of the performed experiments

Experiment	Fitness (reference)	Fitness (TA-Opt)	Fitness (AA + TA-Opt)	Fitness improvement (AA + TA-Opt) [%]	Added improvement by AutoAnalyze [%]
Exp-1	0.04	0.007394	0.00265	93.38	11.86
Exp-2	1.719	0.08156	0.04013	97.67	2.41
Exp-3	43.14	3.944	0.87941	97.96	7.10
Exp-4	0.04	0.007394	0.00033	99.18	17.66
Exp-5	1.719	0.08156	0.03259	98.10	2.85
Exp-6	43.14	3.944	0.99263	97.70	6.84
Exp-7	0.06928	0.05366	0.03097	55.29	32.75
Exp-8	0.06928	0.05366	0.03784	45.38	22.83

- TAOPT-1: Process mapping, Task Splitting and Periodic Offset Assignment
- TAOPT-2: Process Mapping and Periodic Offset Assignment
- TAOPT-3: Process Mapping, Task Splitting, Periodic Offset Assignment and Data Mapping
- TAOPT-4: Process Mapping, Periodic Offset Assignment and Data Mapping

Note that Task Splitting is only configured when TA Optimizer is used without AutoAnalyze. Further splitting the tasks of a fine-grained task set would lead to an unnecessary increase of the vast search space.

4.3 | Results

The results of the experiments described in the previous section are shown in Table 4. For each experiment, the fitness of the reference solution, the fitness of the best alternative solution for both "TA-Opt" and "AA + TA-Opt" cases and the relative fitness improvement of the best alternative solution are compared. Moreover, the added value (additional improvement by AutoAnalyze) is provided.

All experiments yield a significant (around 50% or greater) improvement compared to the reference solution. AutoAnalyze always resulted in an additional improvement compared to the respective experiment where TA Optimizer was solely used to create alternative solutions. The highest improvement by AutoAnalyze was achieved with experiments Exp-7 and Exp-8.

4.4 | Evaluation

As stated in Section 2, full-scale optimizations usually consists of several ten thousand alternative solutions. However, the optimizations were configured to produce only 256 solutions. This setting represents a typical "potential exploration" to evaluate rather quickly to what extent an initial solution can be improved. This is due to the fact that simulations of complex systems like an EMS—especially when detailed simulation models for the hardware are used—are quite costly in terms of runtime. Therefore, the goal is to save time and resources by first evaluating the potential of improvement before starting a full-scale optimization. Each experiment conducted in the case study took around 16 hours to complete on a computer with "Intel Core i7-2930K" processor (6 cores, up to 12 simultaneous threads) with a clock frequency of 3.2 GHz with 16 GB RAM. We have configured TA Optimizer to use 5 out of the 6 cores to run up to 10 simulations in parallel while 1 core is reserved for running the OS. AutoAnalyze on the contrary only requires a few seconds to provide an appropriate initial partitioning and mapping on such a computer. Since AutoAnalyze led to an additional improvement for every experiment, the hypothesis we stated in Section 4.1 is fulfilled. In case of the experiments with the Bosch-EMS, the combination of AutoAnalyze and TA Optimizer could improve the reference solution more than twice as much within the same given time compared to the experiment where TA Optimizer was solely used.

5 | SUMMARY AND OUTLOOK

Because of the inevitable complexity associated with migrating single-core legacy ECU software for a proper execution to multi-core platforms, innovative methods and approaches are urgently needed.

With the objective of enabling an efficient parallelization of AUTOSAR application software on function level, we introduce a tool-supported systematic approach that supports software engineers when analyzing, validating, partitioning, and mapping AUTOSAR model data.

Following the intention to eventually determine advantageous solutions in terms of low overall latency, minimal cross-core communication rates as well as proper load balancing, the suggested proceeding and the corresponding tool enable to efficiently narrow down the search space for the following working steps scheduling, simulation, and optimization. This is accomplished by identifying and solving potential consistency conflicts from the outset and by facilitating the parallelization of AUTOSAR models.

In comparison with hitherto employed approaches that usually draw on very simple strategies to obtain an initial task set (partition) and its distribution to available cores (mapping), the developed algorithms included in the introduced tool considerably reduce the afterwards necessary search effort via automatically providing a beneficial starting point.

To verify the benefit of our approach, we apply it to two complex engine management system models to obtain different variants of solutions (partitions and according mappings), which we then compare with a previously calculated one. The case study shows that a preceding data dependency analysis combined with a skillful partitioning and mapping (that builds on its outcome) is able to significantly enhance the solution quality while reducing the required effort (time and resources) for finding a suitable solution.

As the automotive sector's demands are rapidly rising and even many-core technology becomes progressively common (like reflected by a growing number of cores with distributed memories or heterogeneous connectivity[4]), the presented approach can serve as promising starting point for overcoming the obstacles arising from this development.

ACKNOWLEDGEMENTS

We would like to cordially thank the Timing-Architects Embedded Systems GmbH for giving us the opportunity to use their software within TA research partner program.

REFERENCES

1. Fürst S. AUTOSAR Adaptive Platform for Connected and Autonomous Vehicles. *EMCC 2015 Proceedings*, Munich, Germany; 2016.
2. Deubzer M, Hobelsberger M, Mottok Jürgen, et al. Modeling and simulation of embedded real-time multicore systems. *Proceedings of the 3rd Embedded Software Engineering Congress*, Sindelfingen, Germany; 2010:228–241.
3. Schäuffele J, Zurawka T. *Automotive Software Engineering*. Berlin, Germany: Springer DE; 2010.
4. Mackamul H. AMALTHEA - an open source development platform for embedded multi- and many-core systems. embedded multi-core conference. *EMCC 2015 Proceedings*, Munich, Germany; 2015.
5. Grave R. Software integration challenge multi-core experience from real world projects. embedded multi-core conference. *EMCC 2015 Proceedings*, Munich, Germany; 2015.
6. Mader R. Timing and design tool support in continental powertrain multi-core platform. embedded multi-core conference. *EMCC 2015 Proceedings*, Munich, Germany; 2015.
7. Schüle T, Gleim U. *Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C*. Heidelberg, Germany: dpunkt. verlag, 2012.
8. Kienberger J, Minnerup P, Kuntz S, Bauer B. Analysis and validation of AUTOSAR models. *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, Lisbon, Portugal; 2014:274–281.
9. Sutter H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's J.* 2005;30(3):202–210.
10. Bohn M, Schneider J, Eltges C, Rößger R. Migration von AUTOSAR-basierten Echtzeitanwendungen auf Multicore-Systeme. *Workshop: Entwicklung Zuverlässiger Software-Systeme*, (Stuttgart, Germany); 2011.
11. Wirbel L. Embedded Multicore Goes Mainstream. 2011. http://www.designnews.com/author.asp?section_id=1386&amp;doc_id=231676. Accessed July 15, 2013.
12. Fürst S. An OEM's point of view on multi-core. Embedded multi-core conference. *EMCC 2015 Proceedings*, Munich, Germany; 2015.
13. GLIWA embedded systems. An Introduction to Automotive Multi-Core Embedded Software Timing. 2015. https://www.gliwa.com/downloads/Multi-core%20Poster.pdf. Accessed November 13, 2015.
14. Schatz B. Challenge multi-core TCU: An applied example of multi-core migration. embedded multi-core conference. *EMCC 2015 Proceedings*, Munich, Germany; 2015.
15. Schneider RM. The ARAMiS automotive LSSI demonstrators and the lessons learned. Embedded multi-core conference. *EMCC 2015 Proceedings*, Munich, Germany; 2015.
16. Padberg F, Denninger O. Multicore-Softwarefehler im Visier: Automatische Fehlererkennung in Entwürfen paralleler Programme. *OBJEKTspektrum, Ausgabe 01/2013*. 2013;20(1):72–76.
17. Patterson D. The trouble with multi-core. *IEEE Spectr.* 2010;47(7):28–32.
18. AUTOSAR. AUTOSAR Basic Information - Short Version. 2014. http://www.autosar.org/fileadmin/files/basic_information/AUTOSARBasicInformationShortVersion_EN.pdf. Accessed October 28, 2014.
19. Deubzer M. Multi-core software architecture - moving towards software engineering. Embedded multi-core conference. *EMCC 2015 Proceedings*, Munich, Germany; 2015.

20. Shih C, Wu CT, Lin CY, et al. A model-driven multicore software development environment for embedded system. *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, Seattle, USA, vol. 2; 2009:261–268.
21. Eißenlöffel T. *Embedded-Software entwickeln*. Heidelberg, Germany: dpunkt; 2012.
22. Sodan A. C, Machina J, Deshmeh A, Macnaughton K, Esbaugh B. Parallelism via multithreaded and multicore CPUs. *Comput*. 2010;43(3):24–32.
23. Flämig T. Software architecture methods for multicore - distributed development and validation of architecture in collaboratively engineered multicore systems. Embedded multi-core conference. *EMCC 2015 Proceedings*, Munich, Germany; 2015.
24. Artop Group. AUTOSAR Tool Platform. 2012. https://www.artop.org/ Accessed July 20, 2013.
25. Eclipse Foundation. Eclipse Modeling Framework Project. 2009. http://eclipse.org/modeling/emf/. Accessed July 15, 2013.
26. Saad C. Model Analysis Framework. 2009. http://www.informatik.uni-augsburg.de/en/chairs/swt/ds/projects/mde/maf/. Accessed July 20, 2013.
27. Saad C, Bauer B. Data-flow based model analysis and its applications. In: *International Conference on Model Driven Engineering Languages and Systems*. Miami, Florida, Springer Berlin Heidelberg; 2013:707–723.
28. Timing-Architects Embedded Systems GmbH. TA Tool Suite Version 15.04.0. TA Academic & Research License Program. 2016. http://www.timing-architects.com/ta-tool-suite/. Accessed April 14, 2016.
29. Sailer A, Schmidhuber S, Hempe M, Deubzer M, Mottok J. Distributed multi-core development in the automotive domain - A practical comparison of ASAM MDX vs. AUTOSAR vs. AMALTHEA. *Proceedings of the 1st FORMUSIC Workshop in conjunction with ARCS 2016*, Nuremberg, Germany; 2016:1–8.
30. Johnson R, Pearson D, Pingali K. The program structure tree: Computing control regions in linear time. *ACM SigPlan Notices*, vol. 29. ACM; 1994:171–185.
31. Ottenstein KJ, Ottenstein LM. The Program Dependence Graph in a Software Development Environment. *ACM Sigplan Notices*, vol. 19; 1984:177–184.
32. Tip F. A survey of program slicing techniques. *J Program Lang*. 1995;3(3):121–189.
33. Götz M, Roser S, Lautenbacher F, Bauer B. Token analysis of graph-oriented process models. *13th Enterprise Distributed Object Computing Conference (EDOC)*, Auckland, New Zealand; 2009:15–24.
34. Aykanat C, Cambazoglu BB, Uçar B. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J Parallel Distrib Comput*. 2008;68(5):609–625.
35. AUTOSAR. Specification of Timing Extensions; 2014.
36. Bui TN, Jones C. Finding good approximate vertex and edge partitions is NP-hard. *Inf Process Lett*. 1992;42(3):153–159.
37. TIMMO. Timing Model. 2007. https://itea3.org/project/timmo.html. Accessed November 16, 2015.
38. TIMMO-2-USE. Timing Model - TOols, algorithms, languages, methodology, USE cases. 2010. https://itea3.org/project/timmo-2-use.html. Accessed November 16, 2015.
39. Long R, Li H, Peng W, Zhang Y, Zhao M. An approach to optimize intra-ECU communication based on mapping of AUTOSAR runnable entities. *International Conference on Embedded Software and Systems, 2009. ICESS'09*, IEEE, Hangzhou, China; 2009:138–143.
40. Monot A, Navet N, Bavoux B, Simonot-Lion F. Multisource software on multicore automotive ECUs combining runnable sequencing with task scheduling. *IEEE Trans Ind Electron*. 2012;59(10):3934–3942.
41. Chawla S, Krauthgamer R, Kumar R, Rabani Y, Sivakumar D. On the hardness of approximating multicut and sparsest-cut. *Comput Complexity*. 2006;15(2):94–114.
42. Moyer B. *Real World Multicore Embedded Systems*. Oxford, United Kingdom: Newnes; 2013.
43. Gries M. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI J*. 2004;38(2):131–183.
44. Wehrle K, Günes M, Gross J. *Modeling and Tools for Network Simulation*. Berlin, Germany: Springer Science & Business Media; 2010.
45. Eclipse Automotive Industry Working Group. BTF Specification V2.1.3. 2016. http://wiki.eclipse.org/Auto_IWG#Documents. Accessed April 11, 2016.
46. Schmidhuber S, Deubzer M, Mader R, Niemetz M, Mottok J. Towards the derivation of guidelines for the deployment of real-time tasks on a multicore processor. In: *Model-Based Safety and Assessment*. Berlin, Germany: Springer International Publishing; 2014:152–165.
47. Michel L, Flaeming T, Claraz D, Mader R. Shared SW development in multi-core automotive context. *European Conference on Embedded Real-time Software and Systems*, Toulouse, FR; January 2016.
48. Coffman Jr EG, Garey MR, Johnson DS. Approximation algorithms for bin packing: A survey. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co.; 1996:46–93.
49. Deb K. *Multi-Objective Optimization using Evolutionary Algorithms*, vol. 16. Hoboken, USA: John Wiley & Sons; 2001.
50. Aleti A, Buhnova B, Grunske L, Koziolek A, Meedeniya I. Software architecture optimization methods: A systematic literature review. *IEEE Trans Softw Eng*. 2013;39(5):658–683.
51. Konak A, Coit D. W, Smith A. E. Multi-objective optimization using genetic algorithms: A tutorial. *Reliab Eng Syst Saf*. 2006;91(9):992–1007.
52. König F, Boers D, Slomka F, et al. Application specific performance indicators for quantitative evaluation of the timing behavior for embedded real-time systems. *Proceedings of the Conference on Design, Automation and Test in Europe*. Dresden, Germany, European Design and Automation Association; 2009:519–523.
53. Kienberger J, Saad C, Kuntz S, Bauer B. Efficient parallelization of complex automotive systems. *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, Barcelona, Spain; 2016:40–49.
54. Hamann A, Ziegenbein D, Kramer S, Lukasiewycz M. FMTV 2016 verification challenge. *Inf Process Lett*. 2016.
55. AMALTHEA Project. An Open Platform Project for Embedded Multicore Systems. 2015. http://www.amalthea-project.org/. Accessed November 13, 2015.
56. WATERS Ű 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems. FMTV Verification Challenge. 2016. https://waters2016.inria.fr/challenge/. Accessed March 23, 2016.
57. Quinton S. WATERS Community Forum. 2016. http://ecrts.eit.uni-kl.de/forum/viewtopic.php?f=27&p=69#p79. Accessed March 23, 2016.
58. Infineon Technologies AG. 32-bit TriCore Microcontroller. 2015. http://www.infineon.com/cms/en/product/microcontroller/32-bit-/tricore-tm-microcontroller/channel.html?channel=ff80808112ab68/1d0112ab6b64b50805. Accessed November 16, 2015.
59. Leung JosephY-T, Whitehead J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform Eval*. 1982;2(4):237–250.