Evaluation of Fine-grained Parallelism in AUTOSAR Applications

Alexander Stegmeier*, Sebastian Kehr†, Dave George‡, Christian Bradatsch*, Milos Panic§, Bert Bödekker†, Theo Ungerer*

* Department of Computer Science
University of Augsburg, Augsburg, Germany
Email: alexander.stegmeier,christian.bradatsch,ungerer@informatik.uni-augsburg.de
† DENSOS AUTOMOTIVE Deutschland GmbH
Corporate Research & Development Eching, Germany
Email: s.kehr,b.boedekker@denso-auto.de
‡ Rapita Systems Ltd., York, UK
Email: dgeorge@rapitasystems.com
§ Barcelona Supercomputing Center, Barcelona, Spain
Email: milos.panic@bsc.es

Abstract—Parallelization of AUTOSAR legacy software is a fundamental step to exploit the performance of multi-core electronic control units (ECUs). However, communication between runnables causes serialization and intra-task parallelization can therefore introduce large idle intervals, if a task contains a long critical path. Distributing the instructions of a runnable over cores (fine-grained parallelism) can reduce the serialization to a shorter time, but this requires an efficient and timing analyzable implementation.

This paper investigates the efficiency of fine-grained parallelism for reducing the worst-case execution time in automotive applications. A pattern-supported parallelization approach is applied to extract parallelism of runnables in a structured way. Algorithmic skeletons are used to implement fine-grained parallelism in a dynamic (assignment at runtime) and in a static (a priori assignment) way. The performance evaluation showed that the static assignment is as efficient as a state-of-the-art barrier. Therefore, parallelism is explicitly expressed in a model and implemented in a timing analyzable way.

I. INTRODUCTION

In recent years, the computational power of single-core electronic control units (ECUs) reached a performance limit and multi-core ECUs (MCEs) (e.g. the AURIX [1]) are seen as the hardware platform for current and future automotive control software. They provide a higher absolute performance in comparison with single-core processors. Nevertheless, a vast majority of automotive control software is re-used for multicore processors, because the code is well tested and much effort was spent on the optimization and maintenance. Consequently, this article focuses on the migration of automotive legacy software to MCEs.

The software architecture of automotive control software is standardized by the AUTomotive Open System ARchitecture (AUTOSAR) consortium [2]. An application is described by a hardware and implementation independent hierarchical software component (SW-C) model, in which runnables (i.e. elementary code pieces) inside SW-Cs realize the functional behavior. Runnables with the same release time (periodic or sporadic) are grouped into a task and scheduled by the AUTOSAR OS. The runnable-to-task mapping and the task scheduling define a valid application configuration, for which an existing application is tested and validated.

The migration of an AUTOSAR legacy application to a MCE imposes two main challenges: first, frequent communication of runnables (either mapped to the same or to different tasks) imposes data dependencies that limit the degree of parallelism. Second, the correct functionality of the system is validated with the original application configuration, which defines a specific dataflow (i.e. an order in which runnables process data).

Several approaches for increasing parallelism of AUTOSAR applications and maintaining the dataflow exist. (1) Inter-task parallelism: under this approach tasks are the unit of scheduling and they are distributed to available cores. Timed implicit communication [3] can be used to increase the level of parallelism with a form of asynchronous communication. However, this method introduces a communication overhead and increases the end-to-end latency, which may reduce new parallelization opportunities. These approaches are therefore out of the scope of this article. (2) Intra-task parallelism: under this approach [4] complete runnables of the same task are distributed to cores and the original application configuration is kept. This guarantees the same dataflow as in the original application configuration for the single-core. As a drawback, this approach can introduce large idle intervals due to a long critical path inside a task. Alternatively, graph decomposition techniques [5], [6] distribute a task with different levels of granularity. This is a benefit, because the serialization can be limited to a shorter time span, although this requires a timing analyzable and efficient implementation.

Consequently, this article studies the efficiency of fine-grained parallelism in AUTOSAR legacy applications. We apply fine-grained parallelization to single runnables of a task and determine the worst-case execution time (WCET) of the task with a hybrid method that is based on static source code analysis and measurements. These are conducted on a multi-core processor with two, three, and four cores that is specifically designed to support timing analyzability.
number of data dependencies is typically high in automotive control software. Therefore, we choose a real diesel engine management system (EMS) as example that is composed of roughly one thousand runnables that frequently communicate with each other.

We discuss three different methods for fine-grained parallelization. Among them are Partitioned Scheduling and Graph Decomposition. Furthermore, runnables are parallelized with best-practice solutions in the form of Parallel Design Patterns (PDPs) and represented in an Activity and Pattern Diagram (APD). These diagrams are implemented with Timing-analysable Algorithmic Skeleton (TAS) in a dynamic (assignment at runtime) and in a static (a priori assignment) way. As we will see, a static TAS implementation is as efficient as the state-of-the-art barrier.

The remainder of this article is structured as follows. Section II provides the background to this article. State of the art approaches for intra-task parallelization are described and discussed in section III. Afterwards, section IV describes the evaluated implementations and discusses the impact of fine-grained parallelism on AUTOSAR. Experiments are conducted in section V. Finally, the conclusion is presented in section VI.

II. BACKGROUND

This section explains the WCET analysis method and gives a brief overview about automotive software as well as the use case that is used for the evaluation of fine-grained parallelism.

A. Measurement-based WCET Analysis

In order to investigate the changes of the execution time by parallelization, we derive the WCET. Therefore, the commercial tool RapiTime [7] from Rapita Systems is used. This tool supports the WCET analysis of parallel programs. In contrast to most traditional tools, which are rather analyzing paths through the code statically on a compiled binary, RapiTime utilizes a measurement-based approach.

Measurement-based WCET analysis captures information from execution of the program under analysis on its intended target in the form of execution traces. These traces contain information allowing the tool to create a WCET analysis of the application using real measurements of time taken on target. Trace collection is implemented using an integration library, which implements the routine for an instrumentation point (ipoint). Calls to these routines are inserted throughout the source code at build time, and a call to this library outputs a trace entry. In this article, the ipoint library is implemented using assembly labels and a map of ipoints to addresses. Trace information are issued when code at any of the designated addresses is executed. The overhead for each ipoint is 1 cycle, as a NOP instruction needs to be added to prevent combination of assembly labels at compile time.

As a result, measurement-based timing analysis typically leads to less pessimism in WCET values and it is therefore preferred over sole static analysis for the experiments in this article.

B. Automotive Software and Use Case

The structure of an AUTOSAR application is described with an SW-C model and the communication within is based on the concept of a virtual function bus [2]. The structural element is the SW-C, which can be nested. A port enables interaction with other SW-Cs and runnables inside implement the functional behavior. Two main communication mechanisms for the exchange of a single data element between runnables are defined. Inter-runnable-variable and sender-receiver communication are used between runnables from the same or from different SW-Cs, respectively. The run-time environment guarantees data consistency for both mechanisms. The developer can define the mode of the communication as explicit, i.e. a precedence constraint is imposed from producer to consumer, defining a strict order of execution. The consumer uses the most recent value of the producer.

Alternatively, communication can be defined as implicit: that means a data element is distributed to all consumers after the producer has finished execution. On the consumer side data elements are buffered and calculations are performed on copies. As a result, concurrent execution of runnables is possible, because data are buffered and delivered with a delay. This is a form of asynchronous communication.

We selected a diesel EMS as use case, because the application contains roughly one thousand runnables, which frequently communicate with each other (also via inter-task communication). The internal state of the SW-Cs is updated at different rates. For this reason, runnables with the same release period are mapped to tasks. The EMS contains eleven periodic tasks: τ1, τ3, τ5, τ6, τ16, τ20, τ32, τ64, τ96, τ128, and τ1024, where the index equals the period. An additional task (τ0) executes after an interrupt from the camshaft sensor. In this use case, a runnable is assigned to one task only. The communication within one task is typically explicit and this results in precedence constraints. Figure 1 illustrates this with a task graph for the 96 millisecond periodic task of a real diesel EMS. The thick solid line represents the critical path (period 96 milliseconds) of an EMS.

![Figure 1: Task graph and critical path in a task (period 96 milliseconds) of an EMS.](image)

III. INTRA-TASK PARALLELIZATION

This section describes approaches for intra-task parallelization and discusses their abilities to extract fine-grained par-
allelism, in order to find an appropriate mechanism that is applied to the use case afterwards. Relevant criteria for the discussion are the timing analyzability, efficiency, and compliance with AUTOSAR applications of the approaches.

A. Partitioned Scheduling

A straightforward way for intra-task parallelization is the allocation of complete runnables, because structural partitioning is a fundamental design characteristic of AUTOSAR applications. A SW-C that contains runnables is statically assigned to one core. Hence, partitioned multicore scheduling is ideal for automotive software. Panič et al. describe a scheduler RunPar [4], which computes a valid allocation for the runnables of a single task to a set of identical cores, respecting the data dependencies among them. A time window for the execution of each runnable in the length of its WCET is reserved. This concept is also known as logical execution time [8]. Runnables from different tasks are not executed in parallel. Instead, task execution follows the single-core scheduling. This also guarantees correct communication with past instances of the same task. The runnables are allocated in decreasing order of their combine utilization. Hence, runnables on the critical path are allocated first. The processor is selected using a worst-fit decreasing heuristic. Dependent runnables are allocated first and independent runnables are allocated afterwards.

B. Graph Decomposition

Cordes et al. [5] describe an automatic parallelization process in which the application is analyzed and represented with an hierarchical task graph (HTG). The graph is supplemented by weights (measurement-based execution time) and communication cost. The parallelization of the annotated HTG is done recursively, with depth-first-search. Nodes are combined to an input set for an integer linear programming (ILP) and a solver is used to calculate a mapping of child nodes to cores. The results of the solver are combined until the root node is reached. The ILP solver minimizes the critical path by splitting a node into three sections: sequential, parallel, and sequential. The solution can contain different levels of granularity, because every child node contains parallel sets with different execution times. Platform specific task creation overhead or communication cost can be considered.

Kanehagi et al. [6] demonstrated the use of the compiler OSCAR for the parallelization of a task of an engine management system. The approach is similar to the previous one, i.e. the compiler relies on the decomposition of the task as macro-flow graph. A method called earliest executable condition analysis is applied to identify parallel macro tasks. OSCAR considers the average processing time of operations to determine parallelism. That means, the processing time must be provided as input data for the compiler. To this end, OSCAR uses a list scheduling heuristic to minimize the length of the critical path inside the task.

C. Structured Parallelization with Design Pattern

Jahr et al. [9] proposes a pattern-supported parallelization approach that is applicable to hard real-time (HRT) applications. The original source code is migrated in two phases: Reveal Parallelism and Optimize Parallelism. In the first phase, the code is analyzed with a high degree of parallelization in mind (platform independent). Subsequently, the degree of parallelism is reduced to reach a reasonable level for the target platform (platform dependent). This approach is applicable to distribute a runnable over cores.

Reveal Parallelism is characterized by a code analysis and a subsequent mapping of the analysis result to a model describing structured parallelism. In this model parallel executed segments of code are indicated by PDPs [10], which are textual descriptions of best-practice solutions for parallelization of code. Furthermore, the model can be represented in an Activity and Pattern Diagram (APD) [11], which is derived from the UML activity diagram. In contrast, the APD provides an additional kind of node that enables the modeling of PDPs and the (parallel) fork and join operators are not allowed.

The target of the code analysis is determining whether two segments of a program can execute in parallel in the PDP. Therefore, state-of-the-art analysis techniques (c.f. [12]) are used identify dependencies and represented in a dependence graph. These segments are the basis for mapping the sequential code to the PDPs applying APD. Two segments, which access a shared memory location, must be serialized if at least one of those processes modifies the location [13].

In Optimize Parallelism the degree of parallelism is reduced to a reasonable level for the target platform. Thereby, the ratio between computation of workload and the computation of parallelization overhead is considered as an important factor. Thus, PDPs executing small code segments (few high-level language statements) in parallel are serialized, because a benefit cannot be expected due to synchronization overhead. In addition, unbalanced PDPs (i.e. a sequence or branch with a long and a short code segment) are removed as well. However, determining the length (WCET) of the segments requires measurements on the target platform. Afterwards, it is possible to decide if these code segments remain parallel or not. Segments with uncertain length stay in parallel.

The PDPs describe the parallelism in an implementation and platform independent way. Thus, the implementation remains as last step and can be done with Timing-analyzable Algorithmic Skeletons (TASs) [14]. Therefore, threads are bound to the corresponding skeleton and the workload is allocated before the execution takes place. Afterwards, the threads are released from the skeleton. Thus, a TAS invocation is split into the parts initialization, execution, and finalization. TAS support the implementation of data, pipeline, and task parallelism, whereas the latter one refers to a task in the sense of a PDP. The use of timing analyzable synchronization techniques [15] facilitates the analyzability of the skeletons.

D. Discussion

Graph decomposition techniques do not distribute runnables directly. Instead a complete task is first disassembled with different levels of granularity and the parts are distributed to the available cores. Smaller code segments are automatically agglomerated and synchronization operations are inserted. This is a benefit, because idle times are minimized. However, the discussed approaches are applied to a complete task and they produces annotated source code. That means, they are only applicable to embedded systems that support OpenMP and an implementation independent description, as it is typical for AUTOSAR, is not provided. The timing analysis of such source code is hard, because of the unknown start times of parallel threads.
The main advantage of RunPar is its compliance with AUTOSAR and its practicability. The generated solution can be implemented with AUTOSAR schedule tables. Start times are defined by the WCET, which guarantees that no race conditions can occur. However, the sequential execution of parallelized tasks might produce large idle intervals, if the task has a long critical path. For example, the speed-up for the task in fig. 1 is limited by the critical path to 1.4. Hence, these intervals should be utilized, unless subject to data dependencies. Timing analysis of RunPar schedules is easy, because the start time and the core for the execution of a runnable are predefined.

The pattern-supported parallelization exploits fine-grained parallelism, which makes it possible to reduce serialization from data dependencies to the shortest possible time span. The implementation with TAS eases the WCET analysis by providing structured parallelism, because a synchronized start and end of parallel executed sections is ensured. However, runnable are already small pieces of code and an efficient implementation with low overhead is required, but not described in existing literature. The pattern-supported parallelization is a structured approach that provides an implementation and platform independent representation of parallelism. This is an advantage when fine-grained parallelism is integrated in AUTOSAR. Potentially only small adaptations are needed to integrate the proposed implementation in the AUTOSAR standard. However, this also depends on the implementation of the TAS and thus a more detailed discussion follows in section IV-D2.

Summarizing, graph partitioning has a very limited compatibility with AUTOSAR and partitioned scheduling has a limited granularity, which can produce large idle intervals. Contrarily, the parallelization with pattern provides a structured approach for fine-grained parallelism that can be integrated in AUTOSAR with potentially small effort. However, a specific method for execution of the exposed parallelism is not described so far. Consequently, the next section presents an efficient and timing analyzable implementation of PDPs with TASs.

IV. IMPLEMENTATION OF FINE-GRAINED PARALLELISM FOR AUTOMOTIVE SOFTWARE

This section describes the fine-grained parallelization of the use case in section II-B with the pattern-supported approach as well as the implementation with TASs. These implementations are used to evaluate fine-grained parallelism for automotive software (section V).

A. Selection of Runnables

For determining appropriate bottlenecks we focus on the tasks with the highest load. Therefore, we determine the WCET for every task in the EMS with the method described in section II-A by sequential execution on one core. The load is the result of the WCET multiplied by the number of task executions in a hyperperiod. Selecting runnables from tasks with the highest load for parallelization has potentially the greatest benefits from parallelizing. In concrete four particular runnables were selected for investigation we subsequently call R1 to R4.

After the first phase (Reveal Parallelism) of the applied parallelization approach (see section III-C) the selected Diesel EMS runnables enable possible parallelization applying two different patterns: task and data parallelism. However, the second parallelization phase (Optimize Parallelism) shows that data parallelism is not applicable.

The part of code with the highest potential for using task parallelism is the function FN_func. It obtains a relatively high workload and covers large parts of several runnables. The APD of FN_func as stated after optimization phase is shown in figure 2.

The function starts with a sequential executed initialization phase, while the rest of the code can be executed as task parallelism in two threads. One thread encapsulates the invocation of another function (subsequently named FN2_func) and the other executes the rest of the code in sequential. The APD of FN2_func is displayed in figure 3. It shows that this function can also apply task parallelism. Therefore, the parallelization splits the execution into three threads.

FN_func is included in two of the four parallelized runnables (R1 and R3). In both runnables the function covers large
parts of their workload. Thus, the impact of this function to the runnables is very high. In concrete, it covers 98.82% of the execution time of R1 and 95.63% of the execution time of R3.

Unfortunately, the analysis of the runnables R2 and R4 exhibits dependencies within the runnables, which limit their parallelizable workloads to a size that is too small for distributing it over cores. Hence, in this evaluation we concentrate on the runnables R1 and R3.

C. Implementation

A specific method for execution of the obtained parallelism is not stated in the pattern-supported parallelization approach. Hence, there are several strategies possible. We examine several methods which seem to be appropriate for executing runnables in parallel. They are mainly distinguished by the time when the workload is assigned to the threads.

One of these methods is characterized by assignment of workload at runtime, which means assigning and releasing of workload to particular threads directly before and after execution of a parallel segment in a runnable. Due to the fact that only a part of the time-critical code can be executed in parallel the assignment takes place during time-critical execution. Thus, time demands of the assignment must be considered. Another method is to assign the workload a priori, which means before the execution of the regarded runnable starts. A feasible time for the assignment could be before task scheduling starts. In contrast to the previous methods, we also see the possibility of assigning workload to threads at design time respectively at compile time. Graph decomposition (see section III) is applicable to implement this kind of assignment. Nevertheless, we do not use it because of its incompatibility to AUTOSAR and the possibly different produced parallel code structure compared to the pattern-supported approach (see section III-D). Instead, we manage the assignment by directly specifying it in code applying the observed parallelism.

We provide two different implementations, (1) one with TASs [17], [14] and (2) another one where barriers are utilized by the developer to synchronize parallel sections in code. (1) is applied to implement assignment at runtime and a priori, while (2) realizes assignment at design time. Utilizing TASs for assignment at runtime takes place by invoking its initialization and finalization parts directly before and after the execution part of the skeleton. Subsequently, we call this kind of usage dynamic assigned TAS (dTAS). The assignment a priori is realized by invoking initialization and finalization before respectively after the execution of the runnable which includes the skeleton. As stated before, a feasible time could be at scheduling start respectively end. This kind of TAS usage will subsequently be named static assigned TAS (sTAS). The assignment at design time takes place by inserting the corresponding code to the code of the corresponding thread. Each segment is enclosed by two barriers to ensure a synchronous start and finalization of the parallel executed segments. In case of a conditional execution of the parallel segments, the according conditions have to be checked before reaching the barriers. Therefore, an additional barrier is needed before verification takes place to ensure consistency of the outcome across all cores. In the following this implementation is called barrier-based implementation.

D. Discussion

This section discusses the characteristics of the described parallelization and explains to required extensions for integrating it in AUTOSAR.

1) Parallelization Characteristics: A comparison of the presented implementations obtains significant differences in implementation effort. At conditionally executed parallel code the implementation effort can significantly increase for the barrier-based implementation, because all conditions of previous sequential code need to be checked to decide if the parallel code in fact has to be executed. Thus, the amount of additional work correlates with the length of code that shall be executed previously to the parallel section. In contrast, the presence or absence of the conditional execution does not influence the implementation effort of the TASs.

Participating threads of a parallel runnable execution can exhibit idle times, because of data dependencies within runnables. The overall efficiency of the application can be improved by avoiding these idle times executing workload of other runnables on idle threads. However, some of the presented methods do not provide this kind of usage. While assignment at runtime supports utilization of idle times, it is not possible for assignment a priori. The reason for this is that using idle time is only possible when no workload is currently assigned to the thread. The assignment at design time enables usage of idle times, but in the case of barrier-based implementation it causes high effort and is error prone.

The applied parallelization approach provides a close WCET estimation by enabling structured parallelism. Furthermore, all methods utilized for implementation fulfill HRT requirements. These requirement are ensured by a timing-analyzable implementation of barriers and ticket locks (c.f. [18]). Both synchronization idioms are based on fetch & add hardware primitives.

2) Impact on AUTOSAR: Our approach requires an extension of the AUTOSAR standard. An OS-Application is currently assigned to one core. This is not sufficient for the efficient implementation of parallelized runnables. Hence, a fundamental extension is the support for multi-core assignments to an OS-Application. The consequences of this extension are:

1) SW-Cs must be distributed over cores.
2) The start and termination of tasks must be synchronized. Therefore, schedule tables can be used. Alternatively, the method presented in [19] can be applied to systems based on PharOS [20].
3) The error handling must span multiple cores and react across the borders of a single core. The fault detection, isolation and recovery requires an extension at the application level [21]. During the detection all cores executing a part of the runnable must synchronize about the occurrence of a fault (or error). When a fault has been detected the isolation performs a damage assessment and damage control. Subsequently, the root cause (and the core) causing the fault are detected. In the last step recovery actions are initiated on all cores that execute parts of the parallelized runnable. Hence, every core of the OS-Application must at least be able to detect and isolate an error. Furthermore, the standard needs
extension at basic software level, for example the error hooks.

The implementation with TAS is timing-analyzable and the integration is possible with limited effort. However, the efficiency of the different implementations and the impact of fine-grained parallelism for automotive software is not clear. Thus, we evaluate the sTAS and dTAS in the next section.

V. Evaluation

This section evaluates the efficiency of sTAS and dTAS as well as the impact of fine-grained parallelism on the performance of automotive control software. The speed-up is used as metric for the performance evaluation and defined in the following. Let $C_i^s$ be the WCET for sequential execution of a task $\tau_i$, where interferences from other cores are assumed. Let $C_i^p$ be the WCET for $\tau_i$ the parallel execution with fine-grained parallelism. The speed-up for $\tau_i$ is $S_i = C_i^s / C_i^p$.

A. System configuration

We target time-analyzable two-, three- and four-core processors [18], with a private instruction scratchpad and a data cache for each core (like in [4]). The core is assumed to exhibit no timing anomalies [22] and is connected to an on-chip SDRAM memory device through a tree network-on-chip (NoC). We implemented these processors in a cycle-accurate simulator built upon the SoClib simulation framework1. The interconnection network is modeled using the gNoCSim cycle-accurate flit-level NoC simulator2.

In order to obtain trustworthy WCET estimate, every request that accesses the NoC and the on-chip memory is artificially delayed by the worst-case delay (UBD) [23]. We consider latency of tree NoC 1 cycle for a 2-core processor, and 2 cycles for 3- and 4-core processors (i.e. each core has to traverse 1 and 2 routers respectively) and 1-cycle routers. The memory latency is set to $L_{mem} = 10$ cycles. This configuration provides an UBD of 11 cycles for the 2-core architecture, 22 cycles for the 3-core architecture and 32 for the 4-core architecture.

The applied operating system (OS) complies with automotive standards (OSEK/VDX, AUTOSAR) but comprises only essential functionalities. The OS supports the API defined in the OSEK/VDX specification [24] extended by parts of the AUTOSAR OS module [25] regarding multi-core support. In addition, it provides basic communication and synchronization features for data transmission inside a multi-core processor.

The extensions proposed in section IV-D2 are partly prototyped in the applied OS. In concrete, SW-Cs are allowed to be distributed over cores (see IV-D2-1), but we did not integrate this aspect into AUTOSAR methodology. Further, the synchronized start and termination (see IV-D2-2) is prototyped, but restricted by the constraint that the additional cores are solely utilized for parallel execution of runnables (i.e. no runnables can be scheduled on these cores). Error handling (see IV-D2-3) is not considered until now.

1http://www.soclib.fr/trac/dev

2NaNoC design platform. http://www.nanoc-project.eu

Figure 4: Speed-ups for R1, R3 and FN_func parallelized on various cores utilizing sTAS.

B. Results

Subsequently, the benefits of the parallelization are discussed. We present the obtained speed-ups, efficiency3 and the impact on the critical path. The lines in figures 4 to 9 are edited to ease readability. They do not show measured values.

1) Performance of Runnables:

We show the results of R1, R3 and FN_func. There are different versions of runnables for executions of up to four cores. The sequential execution on one core is applied as reference execution. For an execution on two cores, all investigated runnables utilize the parallelization stated in figure 2. The parallelization on three cores is realized by applying the parallelism of figure 3 and finally, the combination of both leads to an execution on four cores. The implementation with dTAS does not show reasonable benefits. In contrast, it shows even a slow down for some versions of runnable execution. Overall, the speed-up of dTAS is in a range of 0.76 to 1.05, depending on the number of utilized cores. Subsequently, we do not go into further detail for this implementation.

The speed-ups of the sTAS implementation (see figure 4) reach from 1.11 on two cores to a maximum of 3.50 on four cores. The speed-up is similar for all runnables, because their code structure is comparable (see section IV-B). However, R3 performs slightly worse. This is caused by a higher portion of sequential execution compared to the execution of R1 and FN_func. The smaller parallelization overhead of sTAS compared to dTAS results in a better performance and hence, overcomes the shortcomings of the dTAS implementation. While the results obtain only marginal speed-ups for two cores, they are significant when more cores are used. Hence, maximal efficiency is 0.55 on two cores, 0.62 on three cores, and 0.87 on four cores. This value is reached with the sTAS implementation of FN_func on four cores. The low efficiency on two cores is mainly caused by an unbalanced parallelism and on three cores the sequential part of FN2_func prevents a better performance. The parallelism is more balanced on four cores and thus gains a high efficiency.

The speed-ups of the barrier-based implementation are shown in figure 5. They reach from 1.33 on two cores to 3.09 on four cores. On three cores a speed-up of 1.73 is observed for FN_func and R1. Unfortunately, due to simulation problems, the value for R3’s three cores version is missing. The barrier-based implementation reaches an efficiency of 0.67 on two cores, 0.58 on three cores, and 0.68 on four cores. Since,
in contrast to speed-up, efficiency also considers the number of applied cores, it is lower for 3 core than for 2 cores, even though the speed-up raises with the number of cores. As the parallelization overhead is similar to the sTAS, the obtained speed-up values are similar, too. Like the sTAS, this implementation obtains significant speed-up values for execution with multiple cores.

Figure 6 shows the comparison of sTAS and the barrier based implementation. The speed-ups of both implementations increase with the number of cores, but with different gradients. While the speedup of 2 cores is higher for the barrier-based implementation than for sTAS, it is vice versa for 3 and 4 cores. Hence, the parallelization overhead of barrier-based implementation increases faster with a raising number of cores than the overhead of sTAS. This is caused by a difference in synchronization overhead, since the barrier-based implementation needs three barriers for synchronization, while sTAS only applies two barriers during execution phase. The worse performance on 2 cores of sTAS is due to a constant overhead for invoking the skeleton.

2) Impact on the Critical Path: Next, we discuss the impact of the parallel implementations on the critical path of the task they belong to. The runnable R1 is part of the critical path in task 32ms and R3 is part of task 1ms, respectively.

Figure 7 shows the impact of R1 on the critical path in task 32ms. It obtains a reasonable reduction in execution length for the barrier-based and the sTAS implementation. They reduce the length to 89% (barrier-based) and 88% (sTAS) on four cores, respectively. Figure 8 shows the impact of R3 on the critical path in task 1ms. The barrier-based and sTAS implementations again gain reasonable benefits. The length is reduced to 52% and 49%, respectively. This is a significant reduction of the load and is also noticeable at application level.

To investigate the impact on the system in more detail the load of all periodic tasks is accumulated. The 1ms and 32ms task are replaced by the different implementations and compared to the original system load (see figure 9). The sTAS and barrier-based implementation reduce the system load to 84% and 85%, respectively. Please note, this result is achieved by parallelization of two selected runnables. These results support the purposeful parallelization of single runnables, especially when tasks cannot be parallelized by distributing runnables over cores.

C. Discussion

The speed-up gained by using parallelization is directly correlated with the amount of parallelizable code and the ability of balancing parallel execution. Thereby, parallel execution is balanced when the length of execution is equal for all parallel segments. The effect of an unbalanced parallelism can be seen at the implementations for 2 cores where one thread takes significantly more time than the other one. The speed-up of those implementations is limited by the long segment (in terms of execution time) of the parallel execution and in the considered case marginal. However, for 4 cores significant speed-up is observed, which is enabled by the similar lengths of the parallel segments.

The low execution times of runnables causes a significant influence of parallelization overhead. This is especially noticeable for dTAS, which is not beneficial at all because of the high assignment overhead compared to the possible reduction of execution time.

A further usage of idle times is only provided by dTAS and the barrier-based implementation. However, for both implementations it is not practicable to use this ability, because dTAS gains no speed-up and for the barrier-based implementation it is purchased by high additionally implementation effort.
Hence, the support of idle time avoidance is not investigated in detail.

Summarizing, the sTAS implementation provides the best overall performance accept for 2 cores, where the barrier-based implementation is slightly better. However, the barrier-based implementation provides a similar performance for 3 respectively 4 cores and is easier to analyze in addition. The dTAS implementation provides poor performance due to high parallelization overhead. For all presented parallelizations, extensions to AUTOSAR are required.

VI. CONCLUSION

This paper evaluates the efficiency of fine-grained parallelism in AUTOSAR applications. We reviewed state-of-the-art approaches and applied a pattern-supported parallelization approach to single runnables of a real diesel EMS. Experiments are conducted on a multi-core processor with two, three, and four cores that is specifically designed to support timing analyzability. We compared three different methods for the implementation of fine-grained parallelism. The results showed significant speed-ups and high efficiency for the implementation with static implementations. The length of the critical path and the load of periodic tasks are reduced significantly. The results support the purposeful parallelization of single runnables. Thus, the runnable parallelization is particularly suitable when approaches that are parallelizing tasks by distributing runnables show disadvantages because of e.g. many precedences between runnables of a task. However, these results are achieved at the cost of a high effort for implementation or at the cost of outsourcing the assignment of workload to the threads before execution starts.

REFERENCES