Supertask: Maximizing Runnable-level Parallelism in AUTOSAR Applications

Sebastian Kehr *, Miloš Panić †‡, Eduardo Quiñones †, Bert Böddeker *, Jorge Becerril Sandoval *, Jaume Abella †, Francisco J. Cazorla †§ and Günter Schäfer ¶

* DENSO AUTOMOTIVE Deutschland GmbH, Eching, Germany † Barcelona Supercomputing Center (BSC), Spain
‡ Universitat Politècnica de Catalunya, Barcelona, Spain § Spanish National Research Council (IIIA-CSIC)
¶ Telematics/Computer Networks Group, Ilmenau University of Technology, Germany

***Abstract*—The migration of legacy AUTOSAR automotive software from a single-core ECU to a multicore ECU faces two main challenges: 1) data dependencies between AUTOSAR *runnables* must be respected, which may limit the level of parallelism; 2) the original *data-flow* from the single-core must be reproduced, in order to guarantee the same functional behaviour without exhaustive validation and testing efforts afterwards.**

This article proposes the concept of *supertask* that maximizes the level of parallelism among runnables and maintains the original data-flow from the single-core. Supertasks group consecutively scheduled AUTOSAR tasks into a unique scheduling entity with a period equal to the least common multiple of tasks composing it. We evaluate supertasks with a real automotive application and compare it with existing state-of-the-art approaches with the same objectives. Our results show that supertasks effectively increase the performance with respect to current state-of-the-art, resulting in an overall performance improvement of the application when combining supertask with current approaches.

***Index Terms*—parallelization, AUTOSAR, automotive control software, multi-core, legacy migration.**

I. Introduction

The performance requirements for an automotive embedded system have increased steadily in recent years. Today, road vehicles contain a complex in-vehicle network of 70 or more *electronic control units (ECUs)* [7]. Hence, minimizing the resource consumption is one of the most important design objectives [11]. Fortunately, embedded multicore processors, like the Infineon AURIX [8], have become widely available. 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 multicore ECUs.

Automotive software is described according the *AUTomotive Open System ARchitecture (AUTOSAR)* standard [2]. An application is described by a hierarchical *Software-Component (SW-C)* model, in which *runnables* (i.e. elementary code pieces) inside SW-Cs implement the functional behaviour. Runnables with the same release time (periodic or sporadic) are grouped into AUTOSAR task structures and scheduled by the AUTOSAR OS. The original runnable-to-task mapping and the task scheduling define a valid *application configuration*, for which the application is tested and validated.

The migration of an AUTOSAR legacy application to a multicore ECU 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 *data-flow* (i.e. an order in which runnables process data). Hence, re-arranging runnables in new tasks to increase the level of parallelism (e.g., with simulated annealing [6] or with bin-packing heuristics [12]), requires a re-validation of the functional correctness. The associated expenditure of human labour is high and therefore undesirable.

Several approaches for increasing parallelism of AUTOSAR applications and maintaining the data-flow (for reducing the validation effort) exist. These approaches can be divided in two groups: (1) *task-level parallelism:* under this approach, already supported by AUTOSAR, tasks are the unit of scheduling and they are distributed to available cores. In order to increase the level of parallelism, the new communication method *timed implicit communication (TIC)* [9] can be used; the communication among tasks is supplemented by timestamps. This allows for maintaining the same data-flow in any multicore implementation. TIC however introduces a communication overhead, which may reduce new parallelization opportunities. (2) *Runnable-level parallelism:* under this approach, runnables of the same task are distributed to cores and the original application configuration is kept. Graph decomposition [4] distributes a task with different levels of granularity. But, this is not supported by AUTOSAR as it only supports scheduling of tasks that contain runnables. Alternatively, a partitioned scheduler like *RunPar* [14] distributes runnables of the same task to cores. This guarantees the same data flow, as in the original application configuration for the single-core, and the validation effort is drastically reduced. As a drawback, this approach can introduce large idle intervals due to a long critical path inside a task.

This article investigates possible performance improvements of existing methods and how runnable- and task-level parallelisms are ideally employed. The contributions are as follows:

1) A new AUTOSAR structure named *supertask* is proposed, which further exploits runnable-level parallelism of applications

The research leading to these results were funded by the EU FP7 no. 287519 (parMERASA), the ARTEMIS-JU no. 621429 (EMC^2), the German Federal Ministry of Education and Research (EMC^2) as well as the Spanish Ministries of Science and Technology (no. TIN2015-65316-P), Education (no. FPU12/05966), and moEconomy and Competitiveness (no. RYC-2013-14717).

and maintains the original data-flow of the applications. Our technique groups runnables from (originally) consecutive scheduled tasks into a supertask. Runnables of the supertask are then scheduled, whereas inter-task data dependencies are respected. This allows for maintaining the original data-flow and increasing the parallelism in the legacy application.

2) The scalability of supertasks is evaluated and compared against state of the art runnable-level parallelization. To that end, a diesel engine management system is used, because the application contains a large amount of highly connected runnables. Our results show better scalability of supertasks on 2 and 4 cores. The average speed-up increased from 1.56 to 1.71 on 2 cores and from 1.72 to 1.98 on 4 cores, respectively.

3) For a better classification, supertasks are compared against task-level parallelization (based on TIC). Supertasks provide a higher speed-up under low processor utilization and TIC provides better performance under high processor utilization. Interestingly, our observations suggest the use of supertasks and TIC as complementary strategies for increasing the overall system performance.

This article is organized as follows: the next section defines the objectives for automotive software parallelization and describes the challenges addressed in this article. Section III describes state of the art for task- and runnable-level parallelization approaches. Section IV presents the concept of supertask. The evaluation is presented in section V. Finally, section VI concludes the article.

II. Objectives and Problem Definition

Software parallelization approaches [17] decompose the application into subtasks, conduct a dependency analysis, define a mapping to cores, and schedule tasks. We discuss these steps in the following and explain why considering the original application configuration as parallelization constraint is a reasonable approach. Moreover, the challenges resulting from this approach are presented.

The structure of an automotive application however, does not necessarily represent a decomposition into parallel or independent components. Quite the contrary, each component realizes a subtask (implemented by runnables) of the overall control and communicates frequently with other components. Hence, an automotive application must be decomposed according to the control function, which is realized by the application. Rearranging the runnables in new tasks to increase the level of parallelism requires a re-validation of the functional correctness. That means, extensive tests must be conducted, for which the cost is very high and which is therefore undesirable. Fortunately, the original application configuration defines a well-tested and validated system configuration. Therefore, it is reasonable to use the original application configuration as constraint for the parallelization. The functional properties of the application remain unchanged as well as the tasks.

Hence, when migrating to multicore processor, data dependencies between runnables of the same task must be derived [1] . Here, it is important to take the original application configuration into account, to reflect the original data flow. Afterwards, the multicore scheduler must respect the partial order imposed by data dependencies and guarantee end-to-end latency bounds between sensor and actuator. The latter point marks an essential difference between automotive software parallelization and classical software parallelization.

Formally, an application $\mathcal{A}$ consists of a set of tasks, where each task τ_i has a set of attributes (T_i, G_i). τ_i is instantiated periodically with period T_i and executes all runnables within this period. Deadlines are implicitly defined. The precedence constraints between runnables are represented as a *directed acyclic graph (DAG)* $G_i = (V_i, E_i)$. A node in V_i represents a runnable mapped to this task and the edge $(j,k) \in E_i$ means j precedes k ($j \rightarrow k$) with $j,k \in V_i$. Each runnable r is characterized by a *worst-case execution time (WCET)* estimate C_r. The utilization of runnable $r \in V_i$ is defined as $u_r = {}^{C_i}/_{T_i}$, where $0 \leq u_r \leq 1$.

The problem of partitioning the tasks of the application $\mathcal{A}$ can be formalized as follows: Let $c_1, \ldots, c_m$ be m identical cores. Let G_i be the DAG for τ_i. Let φ_k be a subset of runnables $G_i^k \subseteq G_i$ mapped to core k. The *static partitioning* for τ_i onto m identical cores is represented by $\Phi_i = (\varphi_1, \ldots, \varphi_m)$. The allocated resources of a static partitioning is represented by the cumulative utilization and is defined as $u_{sum}^k = \sum_{r \in G_i^k} u_r$. Similarly, the WCET of the partitioned τ_i is defined as $C_i = max(\forall \varphi_k \in \Phi_i \mid \sum_{r \in G_i^k} C_r)$. The partitioning and scheduling of task τ_i is limited by constraints, which can be summarized as follows:

1) The partial order defined in E_i is guaranteed. That means, all predecessors of a runnable $r \in V_i$ finish before r starts.
2) A runnable $r \in V_i$ is assigned to one core.
3) $u_{sum}^k \leq 1$, $k \in \{1, \ldots, m\}$

III. State of the Art

This section describes approaches for extracting task- and runnable-level parallelism from automotive software. The approaches are discussed in the end of the section.

A. Task-level Parallelism

Timed implicit communication (TIC) [9] is a communication mechanism that transforms the data flow of the original task set. Producer and consumer tasks are decoupled by this transformation, which allows for parallel execution of communicating tasks. The reception of data is shifted by one producer period and bound to the task period. Therefore, the producer stores data in a buffer and attaches a publication timestamp equal to the time at the end of the current task period. The consumer reads from the previous producer instance, as compared to the original task set execution, by selecting a value with the appropriate timestamp. Hence, a task instance can be scheduled at any point within its period.

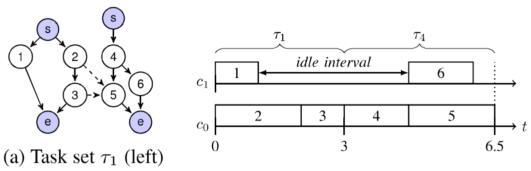
B. Runnable-level Parallelism

An important factor during the migration to a multicore ECU is the validation of the functional correctness. Thus, we consider only approaches that reuse the application configuration and are applied to each task of the application $\mathcal{A}$.

1) Graph-based Decomposition: [4] describes an automatic parallelization process in which the application is analysed 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. Using different levels of granularity is a benefit, on the one hand, because idle times are minimized. But it is also a drawback, on the other hand, because the computational complexity grows with the task size and the integration in AUTOSAR is not possible, as a runnable-to-task mapping is required.

2) RunPar: Structural partitioning is a fundamental design characteristic of AUTOSAR applications. A SW-C (and task) is statically assigned to one core. Hence, partitioned multicore scheduling [5] is preferred for automotive software. RunPar($G_i = (V_i, E_i)$, m) [14] computes a valid allocation $\Phi_i = (\varphi_1, \cdots, \varphi_m)$ for the runnables in V_i of τ_i to m identical cores, respecting the data dependencies E_i. Runnables from different tasks are not executed in parallel; instead, task execution follows the single-core scheduling. Correct communication with past instances of the same task is guaranteed by a sequential task scheduling.

RunPar classifies runnables of V_i in *dependent* and *independent*. A runnable $r \in V_i$ is dependent, if the node degree $\deg(r) > 0$, i.e. the runnable communicates with other runnables from the same task within the same period; the runnable is considered independent, otherwise. The runnables are allocated in decreasing order of their *combine utilization*, which is computed as the highest sum of utilization across the chains of dependencies starting from the observed runnable. Hence, runnables on the critical path are allocated first. The processor is selected using a *worst-fit* decreasing heuristic, i.e. to the least occupied core (the smallest u^k_{sum}). Dependent runnables are allocated first and independent runnables are allocated afterwards. Figure 1 shows an example for the



(a) Task set τ_1 (left) and τ_4 (right).

(b) Schedule with RunPar.

Figure 1: Example for RunPar.

allocation with RunPar. The task set composed of τ_1 and τ_4 is scheduled according to the precedence constraints defined in fig. 1a. The dashed lines represent inter-task dependencies, i.e. $2 \rightarrow 5$ and $3 \rightarrow 5$. Both tasks are scheduled separately and executed consecutively to guarantee the inter-task dependencies. Runnable 2 is allocated as first to core 0, because it has the highest combined utilization. RunPar reserves a time window for each runnable in the length of its WCET. This concept is also known as *logical execution time (LET)* [10].

C. Discussion

Task-level parallelization with TIC keeps the original application configuration, which guarantees a correct data flow inside a task. The communication between tasks is transformed in a predictable manner, which guarantees an identical data flow for all target platforms. The approach is compliant to AUTOSAR and it allows legacy applications to execute tasks in parallel without any modification at source code level. However, an overhead for storing and selecting data is introduced as well as an additional latency between the sensor and actuator (due to delayed communication). Despite the benefits of TIC, it requires careful choice in applying this mechanism.

Runnable-level parallelization reduces the WCET of the task, but it does not necessarily reduce the response time. The reason is, the task activation and scheduling are identical to the execution on the single-core. However, the overall processing time is reduced, because tasks execute faster. This additional computational time can be used for other functionalities or for a reduction of the clock frequency.

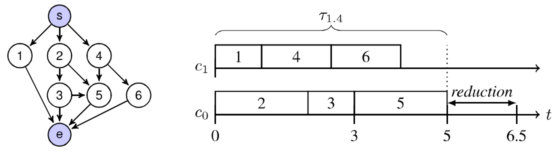
The main advantage of RunPar is its compliance with AUTOSAR and its practicability. The generated solution can be implemented with schedule tables. Start times are defined by the WCET, which guarantees that no race conditions can occur. However, the sequential execution of parallelized tasks might produce large idle intervals (cf. fig. 1). Hence, these intervals should be utilized, unless subject to data dependencies.

Obviously, task- and runnable-level parallelism provide good performance in different scenarios. The former one for a sufficiently large number of tasks and processors; the latter one for a single task, but it is limited by data dependencies. However, a direct comparison of both approaches has not been conducted so far. Moreover, RunPar was evaluated with one bin-packing heuristic and priority rule so far. Consequently, the next section presents a concept for utilizing idle intervals and section V investigates when parallelization strategies are ideally used.

IV. Supertask

This section presents a mechanism for utilizing idle intervals (fig. 1) of consecutively executed tasks: the *supertask*. There are two reasons for this large gap: first, precedence constraints $(2, 3), (4, 5), (4, 6)$ prevent an earlier execution of runnables 3, 5 and 6. Second, consecutive task execution ($\tau_1 \rightarrow \tau_4$) shifts the execution of runnable 4 beyond runnable 3, although no precedence constraint exists. Therefore, we propose combining tasks into one supertask structure, whereby inter-task dependencies are transformed to intra-task dependencies of the supertask. This allows for parallel execution of runnables from both tasks, guaranteeing the same order as in the original task set, and idle intervals are filled.

In general, given two consecutively scheduled tasks $\tau_a \rightarrow \tau_b$, the attributes $(T_{a.b}, G_{a.b})$ of the supertask $\tau_{a.b}$ are defined as: $T_{a.b} = \mathrm{lcm}(T_a, T_b)$ and $G_{a.b} = (V_{a.b} = V_a \cup V_b, E_{a.b} = E_a \cup E_b \cup E_{inter})$. The edges in E_{inter} represent inter-task dependences. They are identified by dependency analysis whereby the control flow is defined by the original task execution order: $\tau_a \rightarrow \tau_b$ (section II). However, the number of combined tasks can be larger than two. Therefore, the process is repeated several times.



(a) Supertask $\tau_{1.4}$. (b) Schedule with RunPar.

Figure 2: Example for a supertask schedule.

Figure 2 shows an example for a supertask, which is based on the RunPar example in fig. 1. τ_1 and τ_4 are combined into $\tau_{1.4}$ with $T_{1.4} = \mathrm{lcm}(1, 4) = 4$, $V_{1.4} = \{1, 2, 3\} \cup \{4, 5, 6\}$, and $E_{1.4} = \{(2, 3)\} \cup \{(4, 5), (4, 6)\} \cup \{(2, 5), (3, 5)\}$. The runnables 2, 3, and 5 represent the critical path of $\tau_{1.4}$. They are allocated first on core 0. Runnable 4 can execute immediately after runnable 1, followed by runnable 6. Overall, supertasks reduce the utilization of a subset of tasks executed within a given cycle. In the example, the idle interval is filled and the total execution time is reduced from 6.5 to 5.

A. Integration in AUTOSAR

A supertask extends the schedule table by a new entry. Similar to tasks, each supertask has its starting point, the order in which it is activated, and an associated runnable scheduling table derived with RunPar. The supertask is activated at $pT_{a.b}$ instead of the original tasks. Supertasks do neither change the runnable-to-task assignment nor the single-core task scheduling. The tasks and the execution order of tasks remain the same. For example, the supertask in fig. 2 is activated with a period of 4 ms. At runtime, the AUTOSAR operating system (AR-OS) task scheduler checks first, if a supertask must be executed in the current cycle. If this is the case, the AR-OS schedules runnables that form the supertask as defined by RunPar (or any other intra-task runnable scheduler), preventing execution of those tasks already executed within supertasks.

The scheduling with supertasks is, like RunPar, based on LETs. However, common automotive software contains sporadic tasks (like the crank-angle task in an *engine management system*) that require a quick response after an interrupt. Therefore, we explain a method to handle interrupts, for preempting a task or supertask during execution. Let τ_p be a periodic task and let τ_s be a sporadic task (parallelized on runnable-level). An interrupt I_s for τ_s during execution of τ_p is handled in a co-operative manner, as shown in fig. 3. First, the latest finish time $f^{\max}$ from all running runnables is determined. All runnables that finish execution until $f^{\max}$ are executed, the earliest finish time f^{min} is determined, and core c idles for the duration d_c until τ_s starts. τ_p relinquishes control to the scheduler at $f^{\max}$ and τ_s starts synchronously on all cores. This can be guaranteed by the mechanism in [3]. The original task interleaving is restored after τ_s by injecting idle intervals of duration d'_c. Hence, shifting the start time s_r of a runnable r to the future start time $s'_r = s_r - f^{min} + C_s$. Overall, the total overhead with a duration for one context switch o_{context} is: $o = C_s + 2 \cdot o_{\text{context}} + \max\{d_0, \cdots, d_{c-1}, d'_0, \cdots, d'_{c-1}\}$. Hence, supertasks are compatible with sporadic tasks.

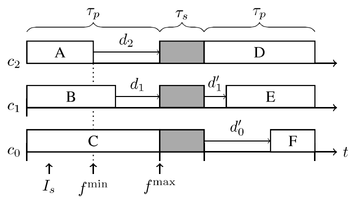


Figure 3: Interrupt of a task or supertask.

B. Discussion

The response time for interrupts is potentially smaller with task-level parallelization, because a dedicated core can immediately serve the interrupt. But, the worst-case load for this task must always be guaranteed by this core and task execution takes longer (in comparison with supertasks). Alternatively, in a system with supertasks, the interrupt is latest served after a time equal to the WCET of the largest runnable in the supertask. Moreover, the load is distributed to all cores. However, sporadic tasks take a small share in processor load and further evaluation is not conducted therefore.

A supertask is a promising solution for filling idle intervals between consecutively scheduled tasks. Clearly, the WCET of a supertask is shorter than with separately parallelized tasks. However, automotive software is characterised by many data dependencies. Thus, we investigate whether supertasks reduce idle intervals in the presence of data dependencies.

V. Evaluation

This section evaluates the concept of supertask. This is done in three steps: first, we look for an ideal RunPar configuration, which is then used for scheduling runnables inside supertasks. Second, we evaluate the performance of supertasks. Finally, we compare and discuss runnable- and task-level parallelization strategies in a comparison.

We selected a diesel *engine management system (EMS)* as use case, because the application contains roughly one thousand runnables, which frequently communicate with each other (also via inter-task communication). The EMS contains eleven periodic tasks: τ_1, τ_4, τ_5, τ_8, τ_{16}, τ_{20}, τ_{32}, τ_{64}, τ_{96}, τ_{128}, and τ_{1024}, where the index equals the period.

A. Experiment Configuration

Analysable 2- and 4-core processors (cf. [18, 15]) are considered as target. Each core has a private instruction scratchpad, a data cache (256 KB), and is connected to an on-chip SDRAM memory device through a *network on chip (NoC)*. The maximum access delay to shared resources is bound by a pre-computed *upper bound delay (UBD)*. The NoC has a wormhole-based tree topology [16] implementing 3 simple pipelined 2-to-1 routers, so each core requires 2 hops to reach the memory. The NoC and the memory interface are sources of interferences. Let L_t be the tree traversal time and let L_m be the memory latency: UBD= $L_t + (c-1) \cdot L_m$. We consider $L_t = 1$ cycles for the 2-core processor and $L_t = 2$ for a 4-core processor, i.e. a message traverses 1 and 2 routers, respectively. The memory latency is $L_m = 10$ cycles. This configuration provides an UBD= 11 cycles for the 2-core and UBD= 32 cycles for the 4-core processor. The static timing analysis tool set OTAWA [13] is used to estimate the WCET. A model of the target processor, the source code, and the compiled executable are used to calculate a trustworthy upper bound for the WCET. The multicore environment is represented by defining the UBD for a request to the processors communication subsystem and memory resources. Hence, the WCET of tasks are time-composable. That means the timing behaviour is independent of simultaneously executed tasks, insensitive to the core allocation, and independent of sharing the cache state with other tasks. Hence, RunPar can be used.

B. Ideal Configuration for RunPar

As preparation, we first look for an ideal bin-packing heuristic and priority rule of RunPar for executing EMS. The speed-up is used as metric, which we define as follows: let C_i be the WCET for sequential execution of τ_i, where interferences from other cores are assumed. Let C_i^{Φ} be the WCET for τ_i as defined in Φ by RunPar. The speed-up for τ_i is $\mathcal{S}_i = {}^{C_i}/_{C_i^{\Phi}}$.

RunPar can be used with any bin-packing heuristic or priority rule. Several criteria can be used for prioritizing runnables during allocation: deadline, period, density, or utilization. RunPar favours the combined utilization. Intuitively, this seems to be a good choice, because the critical path is considered implicitly. A worst-fit decreasing is used as bin-packing heuristic. Again, this seems to be an intuitively good choice, because always the least loaded core is selected. To investigate different configurations, the specification of RunPar is updated as follows: RunPar($G_i = (V_i, E_i)$, m, P, H^D, H^I) computes a valid allocation $\Phi_i = (\varphi_1, \cdots, \varphi_m)$ for the runnables in V_i of τ_i to m identical cores constraints by E_i, using the priority rule P, the bin-packing heuristic H^D for dependent runnables and H^I for independent runnables. RunPar is executed with *worst-fit (WF)* and *first-fit (FF)* bin-packing heuristics, in combination with *utilization (U)* and *combined utilization (CU)* as priority rule. Best-fit and next-fit are variants of the WF and FF, which would produce similar results. They are not considered therefore. Setups 1 - 4 in table I apply the same heuristic to dependent and independent runnables. Setups 5 - 8 apply different heuristics.

Figure 4 shows the average and maximum WCET speed-up for the setups in table I. Setups that allocate dependent runnables with WF (S_2, S_4, S_6, S_8) provide a higher average speed-up than FF. Hence, WF is the superior allocation strategy

Param.	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
H^D	FF	WF	FF	WF	FF	WF	FF	WF
H^I	FF	WF	FF	WF	WF	FF	WF	FF
P	U	U	CU	CU	U	U	CU	CU

Table I: Investigated heuristics and priority rules.

for this application. Surprisingly, using CU as priority rule only provides a small improvement over the U in the average case. This observation is unexpected, because the CU indirectly considers the critical path and thus seems to be the superior rule. However, RunPar considers precedence constraints and hence CU and U are equivalent priority rules. However, our findings confirm S_4 as best choice for the EMS, due to its slightly better average performance (S_2 vs. S_4). Therefore, we present detailed results for S_4 in the rest of this section.

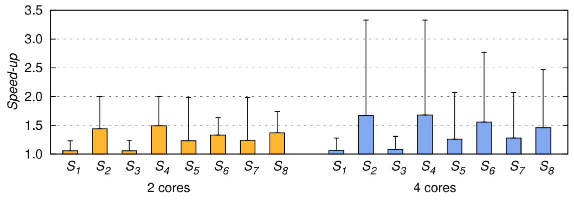


Figure 4: Average and maximum speed-up for setups 1 - 8 on 2 cores (left) and 4 cores (right).

C. Scalability of Task- and Runnable-level Parallelism

We create all (23) possible supertasks from the original application configuration that can occur at runtime: $\mathcal{T} = \{\tau_1, \tau_4\}$, $\{\tau_1, \tau_5\}$, ... , $\{\tau_1, \tau_4, \tau_5, \tau_8, \tau_{16}, \tau_{20}, \tau_{32}, \tau_{64}, \tau_{96}, \tau_{128}, \tau_{1024}\}$. Tasks are parallelized with the following strategies: a) with RunPar, i.e. runnable-level parallelization within tasks and tasks are executed sequentially (labelled as *Separate*), b) supertask (labelled as *Supertask*), and c) at task-level, applying TIC with an overhead of 50 and 100 cycles per buffer access (labelled as *TIC (50)* and *TIC (100)*, respectively. The speed-up for a task set $\mathcal{T}$ is $\mathcal{S}_{\mathcal{T}} = \sum_{\tau_i \in \mathcal{T}} {}^{C_i}/_{\mathrm{makespan}(\mathcal{T})}$. Figure 5 shows the speed-up on two cores. The x-axis denotes the size of the parallelized task set (these tasks are combined to supertasks). To maintain readability, only the best results per task set size

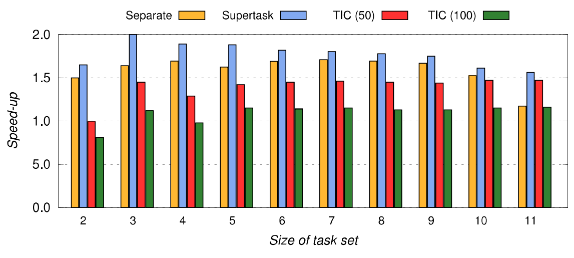


Figure 5: Speed-up on 2 cores.

are shown. All other results are very similar to the shown ones. The results show a better scalability for supertasks over separate task parallelization in any case. The improvement ranges between 5% and 35%. The maximum speed-up is 2, which means an efficiency of 100%. The average speed-up improved from 1.56 to 1.71. Figure 6 shows the speed-up on four cores the same supertask selection for fig. 5. Like in the previous case, similar results are omitted to maintain readability. Similar to the two-core processor case, supertasks obtain a better scalability than separate task parallelization in all cases. The improvement ranges between 5% and 43%. The maximum speed-up is 2.66, which means an efficiency of 66%. The average speed-up improved from 1.72 to 1.98. However, task-level parallelism beats supertask for large supertask sizes (e.g. supertasks 8 to 11). This is discussed in next section.

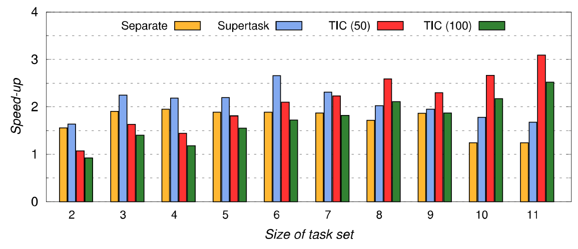


Figure 6: Speed-up on 4 cores.

D. Discussion and Comparison with Task-level Parallelism

On the two-core processor, runnable-level parallelization, i.e. RunPar and supertasks provides better performance compared to TIC (50) (also for cases not shown in the figure). Supertasks provide competitive performance on two cores, without adding latency in the chain of control as TIC does. However, the benefit of the supertask shrinks as the size of supertasks grows, resulting in similar speed-ups like TIC (50) in case of the supertask 11. Contrarily, TIC provides similar performance for almost all task set sizes. Also on four-core processor, supertasks provide better or equal results as TIC for task sets with a size up to 7. But, for larger task sets TIC utilizes cores better, whereas supertask performance decreases. TIC (50) achieves a speed-up of 3.1 for 11 tasks, i.e. an efficiency of 77%. The shrinking benefit of supertasks in comparison to TIC, as the number of runnables increases, results from the fact that TIC ignores inter-task dependencies, while supertasks respect all dependencies. Thus, TIC exploits a four-core processor better.

These results indicate that none of the approaches is the ultimate choice. The processor utilization changes over time due to simultaneous activation of tasks, in multiperiodic systems like automotive software. Thus, the parallelization method should change as the utilization changes. Runnable-level parallelism is ideal for a small number of activated tasks and task-level parallelism is ideal when large tasks are activated. Thus, runnable- and task-level parallelism are rather complementary. Combining these two levels of granularity also reduces the end-to-end latency increment of TIC. Moreover, the overall performance is increased. This makes it possible to either execute more complex algorithms or execute the same application on multiple cores with a reduced clock rate (such that deadlines are still kept) to save energy.

VI. Conclusions

This paper introduces the concept of *supertask* for utilizing idle intervals between parallelized tasks. The original task set is used as parallelization constraint. Consecutive executed tasks are grouped and executed with a period equal to the least common multiple of tasks composing it. Hence, the original execution order is kept and the validation effort after the migration is drastically reduced. The implementation only requires minimum modifications at operating system level and sporadic tasks are suppoted. Supertasks are evaluated with a complex diesel engine management system and found to provided better performance than separate task parallelization. The average speed-up improves from 1.56 to 1.71 (2 cores) and from 1.72 to 1.98 (4 cores), respectively. Additionally, supertasks are compared against task-level parallelization. Supertasks provide better or equal performance on 2 cores and better performance for a task set size up to 7 on 4 cores, respectively. Due to our results, runnable- and task-level parallelism are found to be complementary strategies. Their combination remains as a future work.

References

[1] D. I. August et al. Automatic Extraction of Parallelism from Sequential Code. In V Pankratius et al., editors, *Fundamentals of Multicore Software Development*, part 9. Chapman & Hall / CRC Press, 2011.
[2] AUTOSAR GbR. AUTomotive Open System ARchitecture (AUTOSAR). Standard (v4.1). 2014.
[3] C. Bradatsch et al. Synchronous Execution of a Parallelised Interrupt Handler. In *RTAS, WiP session*, 2014.
[4] D. Cordes et al. Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming. In *Proc. 8th IEEE/ACM/IFIP Int. Conf. CODES+ISSS*. ACM Press, New York, USA, 2010.
[5] R. I. Davis et al. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM CSUR*, 43(4), 2011.
[6] H. Faragardi et al. A Communication-aware Solution Framework for Mapping AUTOSAR Runnables on Multi-core Systems. In *Proc. IEEE Int. Conf. ETFA*, 2014.
[7] S. Fürst. Challenges in the Design of Automotive Software. In *DATE*. IEEE, 2010.
[8] Infineon. *AURIX - TC27x B-Step, 32-bit Single-Chip Micro-controller, User's Manual, v14.1.*
[9] S. Kehr et al. Parallel Execution of AUTOSAR Legacy Applications on Multicore ECUs with Timed Implicit Communication. In *Proc. 52nd DAC*. ACM, San Francisco, 2015.
[10] C. Kirsch et al. The Logical Execution Time Paradigm. *Advances in Real-Time Systems*, 2012.
[11] J Mössinger. Software in Automotive Systems. *IEEE Software*, 27(2), 2010.
[12] F. Nemati et al. A Flexible Tool for Evaluating Scheduling, Synchronization and Partitioning Algorithms on Multiprocessors. In *Proc. IEEE Int. Conf. ETFA*. IEEE, 2010.
[13] H. Ozaktas et al. Automatic WCET Analysis of Real-Time Parallel Applications. In *WCET*, 2013.
[14] M. Panić et al. RunPar: An Allocation Algorithm for Automotive Applications Exploiting Runnable Parallelism in Multicores. In *Proc. 14th IEEE/ACM/IFIP Int. Conf. CODES+ISSS*. ACM Press, New York, USA, 2014.
[15] M. Paolieri et al. Timing Effects of the Memory System in Real-Time Multicore Integrated Architectures: Problems and Solutions. In *ACM TECS*, 2012.
[16] A. Roca et al. Enabling High-Performance Crossbars Through a Floorplan-Aware Design. In *Parallel Processing (ICPP), 2012 41st International Conference on*. IEEE, 2012.
[17] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley, Newark, 2007.
[18] T Ungerer et al. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 30(5), 2010.