Parcus: Energy-aware and Robust Parallelization of AUTOSAR Legacy Applications

Sebastian Kehr *, Eduardo Quiñones †, Dominik Langen *, Bert Böddeker *, Günter Schäfer ‡
* DENSO AUTOMOTIVE Deutschland GmbH, Eching, Germany
† Barcelona Supercomputing Center (BSC), Spain
‡ Telematics/Computer Networks Group, Ilmenau University of Technology, Germany

***Abstract*—Embedded multicore processors are an attractive alternative to sophisticated single-core processors for the use in automobile *electronic control units (ECUs)*, due to their expected higher performance and energy efficiency. Parallelization approaches for AUTOSAR legacy software exploit these benefits. Nevertheless, these approaches focus on extracting performance neglecting the system's worst-case sensor/actuator latency and energy consumption.**

This paper presents *Parcus*; an energy- and latency-aware parallelization technique that combines both runnable- and task-level parallelism. Parcus explicitly models the traversal of data from sensor to actuator through task instances, enabling to consider the latency imposed by parallelization techniques. The *parallel schedule quality (PSQ)* metric quantifies the success of the parallelization, for which it takes the latency and the processor frequency into account.

We demonstrate the applicability of Parcus with an automotive case study. The results show that Parcus can fully utilize the processor's energy-saving potential.

I. Introduction

Multicore *electronic control units (ECUs)* have become widely available for the automotive industry. They provide a surplus of computational power while the energy consumption is lower in comparison to a sophisticated single-core processor with a complex pipeline structure. Additional idle time gained from parallelization can be used for reducing the clock rate to save power [1], which is of high interest for the automotive industry. Nevertheless, there is a strong requirement for re-using existing automotive control software for multicore ECUs. This poses a challenge on the migration of legacy software to exploit the performance benefits of 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 functions. Runnables with the same release time (periodic or sporadic) are grouped into the same task and scheduled by the OS. A high number of data dependencies, resulting from the order in which runnables process data (*data-flow*), is characteristic for automotive software.

An application has an acceptable *first-in-last-out (FILO) latency* [3], which is the maximum time between an input value change and its last corresponding controller output, for which the it is successfully validated and tested [4]. For example the time between the push on a gas pedal and the resulting injection. Guaranteeing the same latency after parallelization is important for ensuring satisfaction of critical reaction requirements. Hence, the migration of AUTOSAR legacy software to a multicore ECU must extract parallelism in the presence of many data dependencies and guarantee the validated FILO latency of the original application configuration.

Preserving the original data-flow is one way to achieve this. Different parallelization approaches of this kind are proposed; for *runnable-level* (RunPar [5] or task splitting [6]) and for *task-level* (*timed implicit communication (TIC)* [7]). The former one statically allocates runnables of the same task to cores, assigning them a *logical execution time* window [8] for the execution. The latter one decouples the task's communication by logically transmitting data at the beginning and the end of a task's period; at the cost of an increased latency due to buffering and distribution with a delay. These techniques enable the parallelization of AUTOSAR legacy applications; reducing the time for execution in a serial way. However, the impact of parallelization on the latency and on the energy consumption have not been studied so far.

This paper investigates how the combination of runnable- and task-level parallelism can explore the aspired energy-saving potential of multicores under strict latency constraints. The contributions are as follows:

1) We propose the energy- and latency-aware parallelization approach *Parcus*[1]. The traversal of data from sensor to actuator through task instances is explicitly modelled to consider the latency imposed by parallelization techniques.
2) We propose an *evolutionary algorithm (EA)* to explore the large number of scheduling possibilities that uses the *parallel schedule quality (PSQ)*, a metric to quantify the success of a parallelization, as fitness function to select the best solution. This allows for handling the large solution space and selecting the best schedule.
3) We evaluate Parcus in extensive simulation studies with a real diesel *engine management system (EMS)*, measure a the Infineon AURIX TC277 [9] to get realistic data for the energy-saving potential of voltage-frequency scaling, and link the simulation results and with the measurements.

The remainder of this paper is organizes as follows. The next section introduces the background to this work. Section III

978-1-5090-5269-1/17/$31.00

[1]Latin: sparing, thrifty, economical, moderate.

first introduces related works and subsequently derives the resulting research challenge addressed in this paper. Section IV introduces Parcus and establishes the new metric PSQ, which is used as fitness function of an EA presented in section V. Section VI presents the evaluation and section VII finally concludes the paper.

II. Background

We define automotive control software as follows.

Definition II.1 (AUTOSAR application $\mathcal{A}$): *The AUTOSAR application $\mathcal{A}$ consists of a set of n tasks: $\mathcal{A} = \{\tau_i \mid 1 \leq i \leq n, i \in \mathbb{N}\}$. The real-time attributes $(\pi_i,\ T_i,\ G_i = (V_i, E_i),\ C_i,\ O_i,\ D_i)$ characterise each task $\tau_i \in \mathcal{A}$. π_i is the priority. T_i is the period. G_i is the runnable dependence graph (RDG) of τ_i, i.e. a directed acyclic graph (DAG) that represents the precedence constraints between runnables of τ_i. Each node V_i represents a runnable. The edge $(q, r) \in E_i$ means q precedes r ($q \to r$), with $q, r \in V_i$. A worst-case execution time (WCET) estimate c_r, expressed in time units, further characterises each runnable $r \in V_i$. C_i is the task's WCET expressed in time units, i.e. the sum of runnable WCETs of τ_i: $C_i = \sum_{r \in V_i} c_r$. O_i is the release time of the first instance of the task, i.e. the offset with respect to the start time of the system. $D_i \leq T_i$ is the relative deadline of the task; they are implicitly defined, i.e. $D_i = T_i$. The release time of τ_i^p is $o_i^p = O_i + pT_i$. The absolute deadline of τ_i^p is $d_i^p = o_i^p + D_i$.*

Definition II.1 also covers sporadic tasks, if they are considered as periodic task with their maximal possible frequency. The WCET in time units (C_i) depends on the clock rate of the processor f. Therefore, the WCET of a $\tau_i \in \mathcal{A}$ is converted from processor cycles to time units depending on f. γ_r is the WCET of runnable $r \in V_i$ in processor cycles. The WCET of $\tau_i \in \mathcal{A}$, in processor cycles, is denoted as Γ_i, i.e. the sum: $\Gamma_i = \sum_{r \in V_i} \gamma_r$. The WCET in time units for runnable (c_r) and task (C_i) scales with the processor's clock rate f: $c_r = \gamma_i/f$ and $C_i = \Gamma_i/f$. We call this *frequency-scaled* WCET.

The clock rate has a direct impact on the schedulability of an application, as a higher clock rate shortens the WCET in time units. The relation is described by constraint II.1.

Constraint II.1 (Schedule Feasibility): *Let $\mathcal{A}$ be an AUTOSAR application according to definition II.1 with frequency-scaled WCET. Let the tuple $\mathcal{S} = (\mathrm{st}_1^0, \mathrm{st}_1^1, \ldots, \mathrm{st}_i^n)$ be a schedule that assigns start times to the task instances of $\mathcal{A}$. The finish time of a task instance is $\mathrm{ft}_i^p = \mathrm{st}_i^p + C_i = \mathrm{st}_i^p + \gamma_i/f$. Thus, the schedule is feasible under the clock rate f, iff*

$$o_i^p \leq \mathrm{st}_i^p < \mathrm{ft}_i^p \leq d_i^p \quad \Rightarrow \quad o_i^p \leq \mathrm{st}_i^p < \mathrm{st}_i^p + \frac{\gamma_i}{f} \leq d_i^p$$

A. Precedence Constraints

Precedence constraints between runnables of the same period are expressed by a RDG. Inter-task communication is expressed as repetitive pattern according to Forget et al. [10]. The *extended precedence constraints* between two tasks τ_i and τ_j correspond to a set of precedences between the instances of the tasks ($\tau_i \to \tau_j$).

Definition II.2 (Extended Precedence Constraint): *For any $k \in \mathbb{N}$, let $\mathcal{I}_k = [0, k]$ and let $\mathrm{lcm}(a, b)$ be the least common multiple of a and b. Let $\tau_i^n \to \tau_j^{n'}$ denote a precedence from instance n of τ_i to the instance n' of τ_j. Let $p_{i,j} = lcm(T_i, T_j)$. The extended precedence constraints are:*

$$M_{i,j} = \{(n, n') \mid \tau_i^n \to \tau_j^{n'},\ (n, n') \in \mathcal{I}_{p_{i,j}/T_i} \times \mathcal{I}_{p_{i,j}/T_j}\}$$

The precedence relation between two tasks repeats every time when both tasks are released simultaneously. The repetitive pattern according to [10] is defined as follows.

Definition II.3 (Periodic Extended Precedence): *The periodic extended precedence $M'_{i,j}$ are imposed by the extended precedence constraints $M_{i,j}$ such that:*

$$M'_{i,j} = \left\{ (n, n') \;\middle|\; \begin{array}{l} \exists k \in \mathbb{N}, (m, m') \in M_{i,j}, \\ (n, n') = (m, m') + (k\frac{p_{i,j}}{T_i}, k\frac{p_{i,j}}{T_j}) \end{array} \right\}$$

B. Data-flow and Latency Semantics

The main criterion for the robustness of automotive real-time controllers is the latency between sensor and actuator. The focus here is on the FILO latency [11], which is the worst-case latency of the controller after a value change. A formal model for end-to-end paths is introduced in [3]. In this model, a system run r produces several *timed paths (TPs)* δ with timed events (k_i) and timestamps (e_i): $\delta = \langle k_1, e_1\rangle, \ldots, \langle k_\ell, e_\ell\rangle$. A TP can describe any chain of events. The subset $\mathrm{TP}_r^{\mathrm{FILO}} \subseteq \mathrm{TP}_r$ contains all FILO paths in run r. The representation is adapted for the purpose of this paper, as we focus on the covered path through task instances from sensor to actuator only; not on every single event. We only consider each path $\kappa \in \mathrm{TP}_r^{\mathrm{FILO}}$ as a data-flow path through the task instances that κ traverses:

$$\hat{P}_{i,j} = \tau_{l_0}^{n_0} \ldots \tau_{l_k}^{n_k} = \tau_i^{n_0} \ldots \tau_j^{n_k} \tag{1}$$

The $\hat{}$ (hat) indicates that P represents the longest possible path; the FILO path. A superscript s denotes the path in a sequential schedule and a superscript p denotes the path in a parallel schedule. The FILO latency ($\mathrm{FILO}_{i,j}$) is calculated from the start of the path in $\tau_{l_0}^{n_0} = \tau_i^{n_0}$, reading sensor input, and the finish time of the last task in the path $\tau_{l_k}^{n_k} = \tau_j^{n_k}$, writing to the actuator:

$$\mathrm{FILO}_{i,j} = \mathrm{ft}_{l_k}^{n_k} - \mathrm{st}_{l_0}^{n_0} = \mathrm{ft}_j^{n_k} - \mathrm{st}_i^{n_0} \tag{2}$$

For better illustration, fig. 1 shows an artificial single-core schedule example and the longest possible sensor-/actuator path (red arrow). Here, an input datum traverses the task instances in a typical rate monotonic fashion; from the sensor, entering the controller in τ_1^0, to actuator, leaving the controller in τ_4^1. In the consequence, the input value is processed in τ_1 and written into a buffer register that is later read by τ_2^2 etc. The FILO path is taken, if a value change takes place *just after* the sensor read-instruction in τ_1^0. Hence, τ_1 does not process the changed value. The resulting FILO path is $\hat{P}_{1,4}^s = \tau_1^0\, \tau_1^1\, \tau_1^2\, \tau_1^3\, \tau_1^4\, \tau_2^2\, \tau_4^1$. The FILO latency in this example is $\mathrm{FILO}_{1,4}^s = \mathrm{ft}_3^1 - \mathrm{st}_1^0 =$

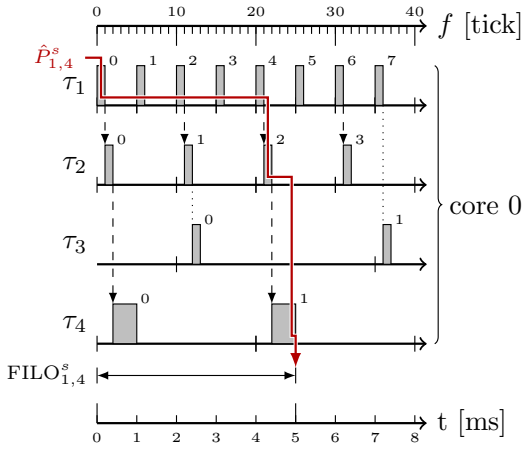


Figure 1: The original system's schedule $\mathcal{S}^s$ including the critical sensor/actuator path $\hat{P}^s_{1,4}$ (red) and its $\text{FILO}^s_{1,4}$ latency.

$T_3 + C_1 + C_1 + C_3 + C_4 = 5\,\text{ms}$. τ_3 does not participate in the computation, but it has a higher priority and thus increases FILO^s. Such a situation is likely to appear in automotive software. The total traversal time in this example is 5 ms. This reference latency must be guaranteed after the parallelization. We define this constraint as follows.

Constraint II.2 (Robust Parallelization): *Let* $\text{FILO}^s_{i,j}$ *be the reference latency validated for the single-core ECU. Let* $\text{FILO}^p_{i,j}$ *be the latency on the multicore ECU. The parallelized application is said to be robust, iff* $\text{FILO}^p_{i,j} \leq \text{FILO}^s_{i,j}$.

III. Related Works and Problem Analysis

This section describes approaches for automotive software parallelization. They are discussed with respect to energy-efficiency and robustness afterwards.

A. Automotive Software Parallelization

Several approaches for increasing parallelism of AUTOSAR applications and maintaining the data-flow exist. They can be divided in two groups.

a) Runnable-level parallelism: Under this approach, runnables of the same task are distributed to cores and the original application configuration is kept. Graph decomposition [12] distributes a task with different levels of granularity, but this is not supported by AUTOSAR as tasks can only contain runnables. A partitioned scheduler, like *RunPar* [5], distributes runnables of the same task to cores. Therefore, RunPar computes a *static partitioning* of the RDG $G_i = (V_i, E_i)$ of τ_i onto m identical cores: $\Phi_i = (\varphi_1, \ldots, \varphi_m)$, where each $\varphi_k \in \Phi_i$, $1 \leq k \leq m$, denotes a subset of runnables $\varphi_p \subseteq V_i$ that are mapped to core k. Thus, the WCET of the partitioned task τ_i is defined as

$$C_i^{\Phi} = \max_{k=1,\ldots,m} \left\{ \sum_{r \in \varphi_k} c_r \right\}. \tag{3}$$

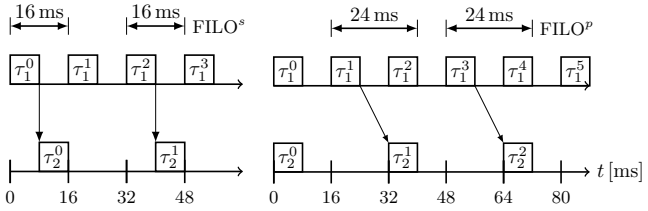
The approach guarantees the same data-flow, as in the original application configuration for the single-core, and the validation effort is drastically reduced. Large idle intervals due to a long critical path can be filled by interleaving tasks [13].

b) Task-level parallelism: Under this approach tasks are the unit of scheduling and they are distributed to available cores. timed implicit communication (TIC) [7] can be used to increase the level of parallelism through supplementing communication among tasks by timestamps. The communication between producer and consumer is decoupled by buffering; shifting the reception of data by one producer period (compared to the single-core schedule). The producer stores data in a buffer and attaches a publication timestamp equal to the time of the end of the current task period. Then, the consumer reads data with the appropriate timestamp from the buffer. Thereby, producer and consumer task can execute in parallel at an arbitrary point within their period. Formally, the TIC transformation according to [7] is defined as follows.

Definition III.1 (TIC Transformation): *Let* $M_{i,j}$ *be the periodic extended precedence constraints (definition II.3) of the single-core ECU. The periodic extended precedence constraints* $M^{\mathcal{MC}}_{i,j}$ *of the multicore ECU are*

$$M^{\mathcal{MC}}_{i,j} = \left\{ (n^*, n') \;\middle|\; \begin{array}{l} \exists n : (n, n') \in M_{i,j} \wedge \\ n^* = \max(n \in \mathcal{I}_{p_{i,j}/T_i} | d^n_i \leq o^{n'}_j) \end{array} \right\}$$

Figure 2 illustrates the idea of TIC for $\tau_1 \to \tau_2$ with $T_1 = 16$ and $T_2 = 32$. We consider a single-core schedule, with $O_4 = O_7 = 0$, as reference. The arrows represent communication. First, we consider $\tau_1 \xrightarrow{(0,0)} \tau_2$ (fig. 2a). They execute in a



(a) Single-core ECU. (b) Multicore ECU.

Figure 2: Example for parallelization with TIC.

serial way, whereas τ_1 executes first and produces input data for τ_2 that are consumed without delay. In contrast, using TIC on the multicore (fig. 2b) results in τ_1 publishing data every 16 ms. The receiver τ_2^1 reads data from the previous producer instance $\tau_1^1 \rightsquigarrow \tau_2^1$, because τ_1^2 publishes data after τ_2^1 is released. Nevertheless, τ_2^1 and τ_1^2 can execute in parallel. The data flows $M_{2,1} = \{(0,1),(0,2)\}$ for the multicore ECU are defined accordingly.

B. Discussion

The original runnable-to-task mapping of an AUTOSAR application establishes, in combination with the task scheduling, a valid configuration, for which the application is tested and validated. This configuration establishes: (a) a specific data-flow, i.e. the order in which runnables process data between sensor and actuator, and (b) an acceptable upper bound on the response time on a stimulus, which is equal to the end-to-end

FILO latency. The approaches described above are designed to maintain the data-flow.

On the one side, runnable-level parallelization reduces a task's WCET, but it does not necessarily reduce its 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 rate.

On the other side, task-level parallelization reduces the response time of tasks, as they can potentially start earlier, at the cost of increasing the latency between two communicating task instances, as fig. 2b illustrated. Generally, the latency for a path using TIC is at least the sum of task periods composing it. It is important to remark that the actual value depends on the finish time of the last task in the path.

Runnable- and task-level parallelism are complementary strategies according to [13]. Unfortunately, none of the approaches considers the impact of parallelization on the latency. The challenge in coordinating runnable- and task-level parallelization is to optimize contradictory targets. Parallelization adds additional idle times that can be used for reducing the clock rate, but this is limited by a (typically) high number of data dependencies among tasks. Yet, TIC enables elimination of those dependencies at the cost of increasing the latency.

Consequently, the next section proposes a mechanism for combining both runnable- and task-level parallelization. This mechanism must maintain the robustness and provide a high energy-saving at the same time.

IV. Parcus

This section establishes a method for minimizing the energy consumption of parallelized AUTOSAR applications on a multicore ECUs, while ensuring robust parallelization. Parcus operates on an AUTOSAR application $\mathcal{A}$ (definition II.1) with frequency-scaled WCET. A feasible parallel schedule $\mathcal{S}^p = (st_1^0, st_1^1, \ldots, st_i^n)$ (constraint II.1) is computed that assigns start times to the task instances of $\mathcal{A}$. Basically, any scheduling can be used to derive $\mathcal{S}^p$. Section VI describes a scheduling heuristic based on an EA used for the evaluation in this paper.

Parcus optimizes the given schedule $\mathcal{S}^p$ in three steps. (1) *Task-fitting:* the lowest possible *clock rate is derived*; this fills idle intervals in the schedule. (2) *Latency-fitting:* the task periods and the clock rate are adjusted to equal the FILO latency of $\mathcal{S}^p$ to the validated reference of the single-core. (3) Finally, the parallel schedule quality (PSQ) metric is computed to assess the parallelization success and enable comparison with other schedules.

To motivate the optimization of the schedule, we look at the effect of task-level parallelization (cf. definition III.1) on the single-core scheduling example from fig. 1. Figure 3 shows a possible parallel schedule $\mathcal{S}^p$ for the same application. In the figure, all inter-task communication on the FILO path is replaced by TIC and the clock rate is set to the value of the single-core schedule $\mathcal{S}^s$. We consider a situation, in which a value change took place just after the sensor read instruction in τ_1. TIC buffers data until publication at the end of the producer

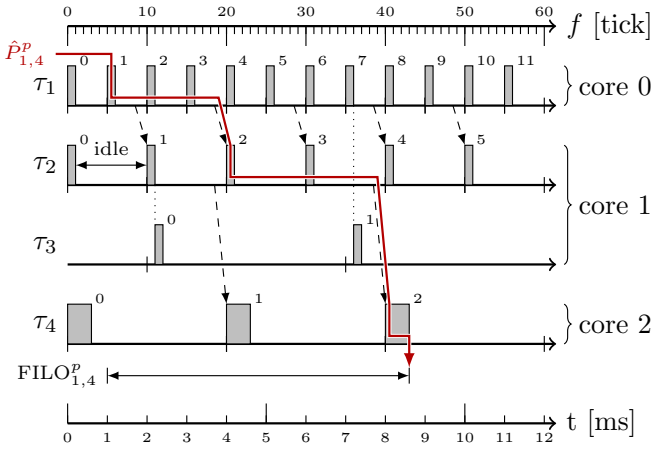


Figure 3: Parallel schedule with FILO path $\hat{P}^p_{1,4}$ and FILO latency $\text{FILO}^p_{1,4}$; running with f =5 kHz.

task's period. Hence, data are not immediately processed by subsequent execution of the receiver task. Thus, the resulting FILO path is

$$\hat{P}^p_{1,4} = \tau_1^1\,\tau_1^2\,\tau_1^3\,\tau_2^2\,\tau_2^3\,\tau_4^2 \tag{4}$$

and the latency $\text{FILO}^p_{1,4}$ (7.6 ms) is larger than the latency $\text{FILO}^s_{1,4}$ (5 ms) of $\mathcal{S}^s$ in fig. 1. The positive side effect from parallelization is a potentially larger idle interval after a task has finished execution, see τ_2^0 in fig. 3 for example. These additional idle intervals allow for the aspired reduction of the clock rate to save energy.

A. Task-Fitting: Adjustment for Energy-saving

During task-fitting, an intermediate result $f'^p_{\min}$ for the frequency is derived, which is thus marked with $\prime$ (prime). Therefore, the clock rate $\underline{f}_i^n$ is derived, which is the frequency to the task instance τ_i^p before its deadline d_i^p. The underline is used to distinguish the frequency for each task instance from the frequency of the application overall. The final frequency for the parallel schedule is computed after determining the latency impact in the next step.

The clock rate $f'^p_{\min}$ is computed in three steps as follows.

1) $\mathcal{S}^p$ is initially derived for $f'^p_{\min} = 1$. That means the WCET of a task in time units is equal to the WCET in processor cycles: $C_i = \gamma_i$.
2) $\underline{f}_i^n$ for τ_i^p is derived from: $\text{ft}_i^p \le d_i^p \;\Rightarrow\; f_i^n \ge \gamma_i / d_i^p - st_i^p$.
3) Consequently, the minimal clock rate needed to guarantee all deadlines is $f'^p_{\min} = \max(\underline{f}_i^n)$.

Knowing $f'^p_{\min}$ makes it possible to exactly calculate the latency of the FILO path $\hat{P}_{i,j}$, because this specifies the concrete finish time of a task instance (constraint II.1). This is done analogous to [3, 11], whereas the transformation with TIC is respected. That means produced data elements are not available for the receiver task before the end of the producer period, if TIC is used. The *frequency impact* reflects

the improvement by a reduction of the clock rate by parallel execution and is defined as follows.

Definition IV.1 (Frequency Impact): *Let $f^s_{\min}$ and $f'^p_{\min}$ represent the minimal clock rate of the sequential schedule $\mathcal{S}^s$ and the parallel schedule $\mathcal{S}^p$ for the AUTOSAR application $\mathcal{A}$, respectively. The frequency impact between the schedules $\mathcal{S}^s$ and $\mathcal{S}^p$ is*

$$\mathcal{F}^{s,p} = \frac{f^s_{\min}}{f'^p_{\min}} \tag{5}$$

The clock rate for the parallel schedule is potentially smaller than the one for the sequential schedule, because parallelization allows more tasks (or runnables) to execute in a parallel way. Thus, this fraction indirectly reflects the degree of parallelism, similar to the well-known speed-up.

Figure 4 shows the schedule from fig. 3 after task fitting, with a reduced clock rate $f'^p_{\min} = 1\,\text{kHz}$ (previously $5\,\text{kHz}$). Here, TIC makes it possible to execute τ_1, τ_2, and τ_4 in a parallel way. The clock rate is reduced until all task deadlines are just met. Here, τ^n_1 and τ^n_3 limit the reduction, i.e. smaller frequencies would result in deadline violations of both tasks.

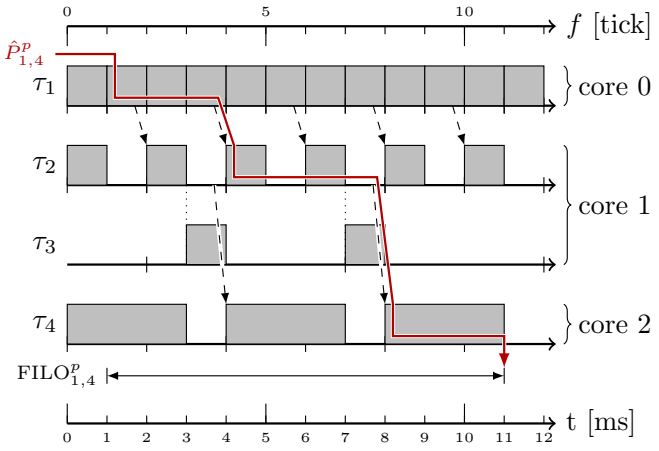


Figure 4: Multicore schedule with a reduced clock rate ($f'^p_{\min} = 1\text{kHz}$).

B. Latency-Fitting: Adjustment for Robustness

The target of latency-fitting is satisfaction of constraint II.2. Lowering the clock rate (fig. 4) increases the finish time of τ^2_4, which in turn enlarges $\text{FILO}^p_{1,4}$ from $7.6\,\text{ms}$ to $10.0\,\text{ms}$. A straightforward way to bound the latency is limiting the usage of TIC, by selecting an appropriate subset of inter-task communication that tolerates a delay. This serialises the execution of both tasks, although they are executed on different cores. The resulting idle intervals can be utilized by applying runnable-level parallelism.

Limiting the usage of TIC is a necessary step, but it is not sufficient to reduce the latency down to the reference value. For example, introducing a precedence constraint between τ_1 and τ_2 would reduce the latency at least by T_1 ($1\,\text{ms}$). Thus, the idea is to also adjust the task period. Our assumption is that, the period is a changeable design parameter initially based on the legacy application's configuration. Therefore, the negative impact on the critical FILO path, when TIC is used, is quantified by the *latency impact*.

Definition IV.2 (Latency Impact): *For the AUTOSAR application $\mathcal{A}$, let $\text{FILO}^s_{i,j}$ represent the latency of the sequential schedule $\mathcal{S}^s$ and let $\text{FILO}^p_{i,j}$ represent the latency of the parallel schedule $\mathcal{S}^p$ after task-fitting. The latency impact is*

$$\mathcal{L}^{s,p} = \frac{\text{FILO}^s_{i,j}}{\text{FILO}^p_{i,j}}. \tag{6}$$

For example, the latency impact for the parallel schedule in fig. 4 in comparison with an assumed baseline latency of $5\,\text{ms}$ is $L = \text{FILO}^s_{i,j}/\text{FILO}^p_{i,j} = 5\,\text{ms}/10\,\text{ms} = 1/2$.

Scaling the original task periods with the factor $\mathcal{L}^{s,p}$ and the previously computed clock rate $f'^p_{\min}$ with the factor $1/\mathcal{L}^{s,p}$ guarantees the same FILO latency as in the reference platform.

Definition IV.3 (Adjustment for Robustness): *Let $\mathcal{L}^{s,p}$ be the latency impact for a parallel schedule $\mathcal{S}^p$ for the AUTOSAR application $\mathcal{A}$. Let $f'^p_{\min}$ be the clock rate for $\mathcal{S}^p$ after task-fitting. Let T'_i is the period of τ_i after task-fitting. The application is adjusted as follows:*

$$f^p_{\min} = f'^p_{\min} \cdot \frac{1}{\mathcal{L}^{s,p}} \quad \textit{and} \quad \forall \tau_i \in \mathcal{A} : T'_i = T_i \cdot \mathcal{L}^{s,p}$$

Such a transformation is valid, because all tasks are scaled with the same constant factor. Scaling the clock rate is necessary to finish all tasks before their deadlines. The resulting parallel schedule with adjusted task periods and frequency is shown in fig. 5.

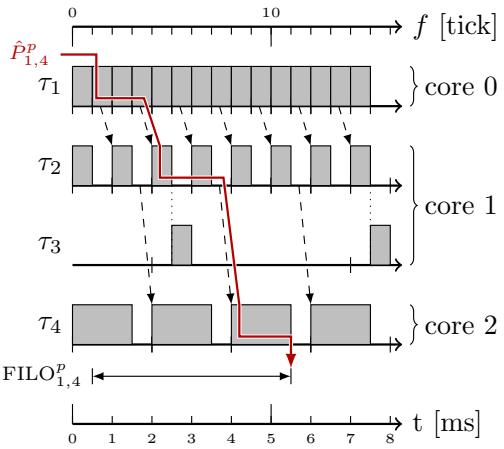


Figure 5: Schedule of fig. 4 adjusted for robustness.

C. Parallel Schedule Quality (PSQ)

Numerous parallel schedules are possible. Selecting a schedule from a set of possible solutions requires a measure that quantifies when one solution is superior to another. Therefore, the parallel schedule quality (PSQ) is introduced in this section, which quantifies the quality of a parallel schedule $\mathcal{S}^p$ in comparison to the former sequential execution of the same application with the schedule $\mathcal{S}^s$.

The first relevant quality criterion for a parallel schedule is the required clock rate, because the parallelization makes it possible to reduce the clock rate. The second criterion is the latency, which increases by task-level parallelization with TIC. Both are derived for a given schedule and considered as criteria. Quantifying whether the task-fitting and the adjustment for robustness result in an overall benefit for the parallelization is reflected by the PSQ.

Definition IV.4 (Parallel Schedule Quality): *Let $\mathcal{F}^{s,p}$ be the frequency impact and let $\mathcal{L}^{s,p}$ be the latency impact of the parallel schedule $\mathcal{S}^p$ over the sequential schedule $\mathcal{S}^s$ for the application $\mathcal{A}$. The parallel schedule quality (PSQ) is defined as*

$$PSQ_{s,p} = \mathcal{F}^{s,p} \cdot \mathcal{L}^{s,p} = \frac{f^s_{\min}}{f'^p_{\min}} \cdot \frac{\mathrm{FILO}^s_{i,j}}{\mathrm{FILO}'^p_{i,j}}. \tag{7}$$

Increasing the number of TIC communications in the parallel application changes the fractions $\mathcal{F}^{s,p}$ and $\mathcal{L}^{s,p}$ in opposite directions. The fraction $\mathcal{F}^{s,p}$ (definition IV.1) increases, because more tasks can execute in a parallel way and this can result in a reduction of $f^p_{\min}$. Contrarily, the fraction $\mathcal{L}^{s,p}$ (definition IV.2) decreases, because the latency $\mathrm{FILO}^p_{i,j}$ is increased. This criterion means to counterbalance the negative impact of the increased latency with the benefits of parallel execution. This is done by applying TIC to communication that affects the latency fewest and provides the best performance gain. Values larger than 1 represent schedules that benefit from the parallelization.

Applying the metric to the schedules in figs. 3 and 4 gives a PSQ of 0.66 and 2.5, respectively. Hence, the schedule with the reduced clock rate (fig. 4) is the superior schedule, when robustness is considered. However, the energy consumption of the former one is potentially lower due to the smaller clock rate. The next section describes how the PSQ can be used to explore the large set of possible schedule solutions.

V. Parallel Schedule Generation

Determining an optimal schedule is computational expensive, because many possible alternatives must be considered. Hence, meta-heuristics, such as simulated annealing [14, 15] or EAs [16] are popular as they scale well. Generally, the PSQ can be used as decision criterion in any heuristic. We decided to implement Parcus based on an EA for solving the *resource-constrained project scheduling problem (RCPSP)* [17], because it is one of the best performing methods [18] and the RCPSP is similar to static scheduling of AUTOSAR tasks for a hyperperiod. Moreover, we decided to use RunPar as representative method for runnable-level parallelism and TIC as representative for task-level parallelism, because there is a deep understanding of both methods. In addition, TIC has an impact on the latency, which Parcus will try to compensate.

Adapting an EA to use the PSQ as fitness criterion for a schedule is straightforward. The basic scheme [19] is:

1.) Generate initial population P
2.) Compute *fitness* for all *individuals* $I \in P$
3.) For $G = 1 \dots N$ or a time limit:
   1.) Produce a set of children $\mathcal{C}$ by *crossover* of P
   2.) Apply *mutation* to $\mathcal{C}$
   3.) Compute fitness for $\mathcal{C}$
   4.) $P = P \cup \mathcal{C}$
   5.) Reduce size of P by *selection*

The meta-heuristic $\mathbf{M}(\mathcal{J}, K^\rho, R^\rho_k, \mathcal{P}, r_{J,k}, p_J)$ generates a feasible hyperperiod schedule $\mathcal{S}^p$ for a job list $\mathcal{J}$, precedence relations $\mathcal{P}$, on the limited renewable resources K^ρ with per-period availability R^ρ_k, job resource request $r_{J,k}$, and job processing time p_J. That means $\mathcal{S}^p$ defines start times of tasks on a real-time axis in a way that each task instance τ^n_i is not scheduled before its release time o^n_i and it finishes before its relative deadline d^n_i.

The job list $\mathcal{J}$ is composed of all task *instances* that appear in one hyperperiod of the AUTOSAR application $\mathcal{A}$. The predecessors of P_J of the job J represent the periodic extended precedence constraints between the task instances. The target is an ECU with one processor with multiple cores that means $K^\rho = \{1\}$ and $R^1_1 = \texttt{procnum()}$. The request of job J for processor capacity is equal to the WCET of the corresponding task in processor cycles that means $p_i = \gamma_i$ for $J = \tau^n_i$.

The EA does not directly work on the solution of the scheduling. Instead, an individual $I = (\lambda, \mathrm{SGS}, \mu, w, \delta, u)$ encodes a unique schedule solution. The *schedule generation scheme (SGS)* [20] transforms the activity list $\lambda = (j_1, \dots, j_J)$ into a feasible schedule $\mathcal{S}^p = (s_1, \dots, s_J)$, which assigns a start time s_j to each activity $j \in J$.

The SGS starts from zero and builds a feasible schedule by stepwise extension of a partial schedule. In the partial schedule, only a subset of the activities have been scheduled. A distinction is made between *activity-* and *time-incrementation*. Two different SGSs are used. The *serial* SGS performs activity-incrementation, i.e. jobs are scheduled in the order as defined by the activity list. Each activity is assigned the earliest precedence and resource feasible start time possible. The *parallel* SGS performs time-incrementation, i.e. it computes a decision point at which an activity, to be scheduled, is started. At this point, a set of eligible activities is determined and successively scheduled until none is left. This process repeats until all activities are scheduled. Interested readers may consult [20] for further details.

The SGS does not refer to runnable- or task-level parallelism of activities. It specifies the way how task instances (activities) in the given task set J are scheduled and not whether the task itself is parallelized. The combination of activity list and SGS always produces the same schedule. Thus, shuffling the activity list produces another schedule. The representation of an individual has been extended by μ, w, δ, and u to express properties of parallelism.

The *communication list* $\mu = ((\tau_i, \tau_j), \dots)$ represents communication that is replaced by TIC and the *task list* $\delta = \{\tau_1, \cdots, \tau_J\}$ represents tasks that are parallelized with RunPar. A tuple $(\tau_i, \tau_j) \in \mu$ represents periodic extended precedence constraints between τ_i and τ_j. The order within these lists decides whether task- or runnable-level parallelism

is used. This is carried out in the following way. The parameter w represents the number of inter-task dependencies in μ that use TIC, starting from the left of the list. The parameter u represents the tasks in δ that are parallelized with RunPar, also starting from the left of the list. As a consequence, a task is performed in one out of two modes: *serial* or *parallel*. Executing a task in the serial mode means one core is allocated for the execution ($r_{i,1} = 1$). Contrarily, executing a task in the parallel mode results in the allocation of all cores ($r_{i,1} = R_1^1$), but with a shorter WCET (as defined by RunPar). μ and δ are instantiated with a random order during initialization of the population.

The EA uses a two-point crossover and mutation randomly changes the activity list or SGS. The crossover of two individuals takes place analogous to [17]. The communication and the task list are also merged with a two-point crossover. Afterwards, both lists are traversed from left to right and an element is shifted to the right with a probability $p_{\text{mutation}} = 0.05$, also recommended by [17].

Each iteration follows the basic scheme of an EA. The last step in each iteration is the selection of the fittest individuals for reuse in the next iteration. The PSQ (definition IV.4) is used as fitness function for an individual whose value must be maximized. Therefore, $f_{\min}^p$ and $\text{FILO}_{i,j}^p$ are calculated for every schedule in the population. Finally, the individuals (schedules) are ranked by their fitness and the ones with the highest value are selected for survival of the iteration.

In preliminary experiments, we found a *population size of 100* individuals and *six iterations* are enough to find good schedule solutions. The computation time for a large schedule (5285 jobs) is approx. 200 s on average for one iteration. A mechanism known as *clone detection* is used to avoid redundant calculations.

VI. Evaluation

First, we investigate Parcus' potential for decreasing the processor's clock rate. Therefore, an automotive case study is used. Subsequently, we investigate how the observed clock rate reduction translates in energy-saving on a real processor.

A. Case Study

Parcus is not meant to be used for safety-critical software, e.g. an anti-lock braking system. Hence, we selected a diesel EMS as use case, composed of more than one thousand runnables distributed among ten periodic tasks: τ_1, τ_4, τ_5, τ_8, τ_{16}, τ_{20}, τ_{32}, τ_{64}, τ_{96}, τ_{128} (index equals period) with frequent communication in between. The crank-angle task communicates asynchronously with periodic tasks and it is, thus, not considered here. We apply task-level parallelization (TIC) stepwise and observe the variation of the *gas pedal/injection* latency ($\mathcal{L}^{s,p}$) between τ_{16} (gas pedal sensor acquire) and τ_{32} (injection quantity and timing calculation): $\text{FILO}_{16,32}^s$. Furthermore, we observe the variation of the latency in the longest inter-task data-flow chain in a hyperperiod as a unique end-to-end latency, in order to evaluate Parcus with a long chain. Therefore, we identified the longest chain in the EMS ($\text{FILO}_{1,128}^s$) that ranges from τ_1 to τ_{128}:

$$\hat{P}_{1,128} = \tau_1^{n_0}\tau_4^{n_1}\tau_5^{n_2}\tau_8^{n_3}\tau_{16}^{n_4}\tau_{32}^{n_5}\tau_{64}^{n_6}\tau_{128}^{n_7}. \tag{8}$$

Thus, the latency impact $\mathcal{L}^{s,p}$ (definition IV.2) depends on the impact per path, i.e.

$$\mathcal{L}^{s,p} = \max\left(\text{FILO}_{16,32}^s/\text{FILO}_{16,32}^p, \text{FILO}_{1,128}^s/\text{FILO}_{1,128}^p\right). \tag{9}$$

The static timing analysis tool set OTAWA [21] is used to estimate the WCET of tasks. The approach presented in this paper is independent of the timing analysis method applied. Any other tool can be used to compute the WCET estimates of tasks.

B. Minimal Possible Processor Frequency

The processor frequency can be reduced proportionally with the supply voltage. That means the clock rate gives a first indication about the energy-saving potential.

1) Experiment Configuration: We consider a time-predictable quad-core processor [22] as the target platform, upon which the EMS is scheduled. 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)*, featuring a wormhole-based tree topology implementing three simple pipelined 2-to-1 routers. The maximum access latency to the NoC and the memory device, which are the main sources of interferences, is pre-computed. Such a processor is comparable to the AURIX.

The EA explores different combinations of runnable- and task-level parallelism and the schedule with highest PSQ is selected. The number of freely selectable RunPar tasks (runnable-level) is set to 0, 6, and 10. That means none, 6 or 10 tasks are allowed to distribute runnables to all available cores. The resulting graphs are labelled as $f_{\min}^0$, $f_{\min}^6$, and $f_{\min}^{10}$, respectively. For each $f_{\min}^i$, the number of TIC communications (task-level) increases over the x-axis, from left to right. That means complete tasks can run in parallel, in addition to other tasks that distribute runnables over all cores.

We conduct two experiments. An experiment A, we observe $f_{\min}^0$, $f_{\min}^6$, and $f_{\min}^{10}$ when task- and latency-fitting are applied, i.e. the parallel program is efficient and robust. That means the clock rate and task periods are adjusted according to section IV. In experiment B, we observe $f_{\min}^0$, $f_{\min}^6$, and $f_{\min}^{10}$ when only task-fitting is applied, i.e. the parallel program is optimized for efficiency. We observe in both experiments the processor frequency relative to execution in a serial way (100 %), on one core using a non-preemptive scheduler.

2) Results of Experiment A ($f_{\min}^p$ with Task- & Latency-Fitting): Figure 6 shows the minimal clock rate needed to guarantee all deadlines of the schedule relative to the frequency needed for execution in a serial way. The plot shows the result of the schedule with the highest PSQ for the given combination of runnable- and task-level parallelism. The sole use of runnable-level parallelism ($f_{\min}^{10}$) means no task can execute in parallel. Instead, all tasks are executed in a sequential order but with a shorter WCET due to parallel execution of

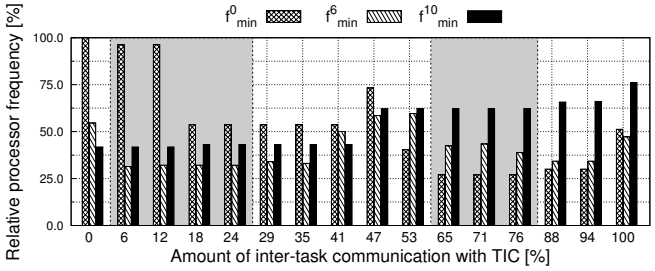


Figure 6: Results of experiment A. The minimal clock rate needed to guarantee all deadlines after adjustment for robustness by task- and latency-fitting.

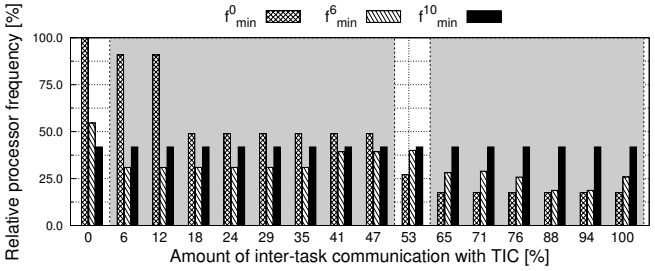


Figure 7: Results of experiment A. The minimal clock rate needed to guarantee all deadlines after task-fitting.

runnables. Combining this with task-level parallelism leads to a higher latency, because communication between tasks is buffered. As a consequence, the frequency needs to be increased as the amount of TIC communication grows. Such combinations do not make sense in practice, but they clearly illustrate the impact of TIC on the latency. Interestingly, the EA does find schedules that do not affect the latency (between $0\,\%$ to $12\,\%$), if some inter-task communication is on the critical path. The maximum possible frequency reduction of $f^{10}_{\min}$ is $58.5\,\%$.

Contrarily, only using task-level parallelism ($f^{0}_{\min}$) requires a large amount of connections to be replaced (at $53\,\%$) in order to have the same processor frequency as with runnable-level parallelism ($f^{10}_{\min}$). Further, increasing the amount of task-level parallelism even reduces the frequency down to $54\,\%$ (grey area from $65\,\%$ to $76\,\%$). Larger values again increase the frequency due to the adjustment for robustness. The maximum possible frequency reduction of $f^{0}_{\min}$ is $73\,\%$.

Combining runnable- and task-level parallelism ($f^{6}_{\min}$) leads to the envisioned improvement over the individual approach, when the amount of inter-task communication is between $6\,\%$ and $35\,\%$ (grey area on the left). However, the maximum possible frequency reduction of $f^{6}_{\min}$ is $69\,\%$. Thus, Parcus enables a reduction of the processor frequency, even if strict latency constraints are considered. The largest frequency reduction under strict constraints is $73\,\%$ (with $f^{0}_{\min}$).

3) Results of Experiment B ($f^{p}_{\min}$ with Task-Fitting Only): Figure 7 shows again the minimal clock rate needed to guarantee all deadlines of the schedule relative to the frequency needed for execution in a serial way, but this time only task-fitting is applied. In contrast to experiment A, the frequency of solely using runnable-level parallelism ($f^{10}_{\min}$) does not change as the amount of task-level parallelism increases, because latency-fitting is not applied. The latency increases, but there is no compensation. Hence, all values for the frequency are equal and maximum frequency reduction with $f^{10}_{\min}$ of $58.5\,\%$ is possible.

Again, the combination of both methods ($f^{6}_{\min}$) outperforms the individual approaches in the range from $6\,\%$ to $47\,\%$ (grey area on the left). This area is larger than in experiment A, because no adjustment is made. The maximum possible frequency reduction of $f^{6}_{\min}$ is $81.5\,\%$ and is shown outside this region when the amount of task-level parallelism is $88\,\%$ or $94\,\%$.

The overall best results are achieved, when only task-level parallelism is used. The needed processor frequency goes down to $17.5\,\%$ compared to the processor frequency of task execution in a serial. This is illustrated by the grey area on the right side, where the amount of task-level parallelism ranges from $65\,\%$ to $100\,\%$. Hence, the maximum possible frequency reduction of $f^{0}_{\min}$ is $82.5\,\%$. Such a parallelization would only be acceptable, if the application tolerates the latency increment.

On the one hand, these results promise possible energy-savings, because the supply voltage can be scaled down proportional to the frequency. The maximum possible frequency reduction under strict latency constraints is $73\,\%$ and increases to $82.5\,\%$, when latency constraints are relaxed. The large reduction results from a pessimistic baseline system with non-preemptive scheduling. Nevertheless, this gives a useful hint about the potential for reducing the frequency. As a result, one would expect a likewise large energy-saving on a target processor, but modern manufacturing techniques shrink the margin for energy-saving from voltage-frequency scaling. Hence, we investigate the available energy-saving margin of a real processor in the next section.

C. Energy-saving on the Infineon AURIX

The energy-saving depends on the processor architecture. To this end, we consider the Infineon AURIX [9] with three cores; a typical representative for an automotive embedded multicore processor. This microcontroller is fabricated in a $65\,\mathrm{nm}$ technology. Its core logic is qualified for an operating voltage of $1.3\,\mathrm{V}$ and the maximum clock rate is $200\,\mathrm{MHz}$.

1) Experiment Configuration: The AURIX microcontroller runs the same EMS application during all measurements, which is parallelized to run on its three cores. The test application is parallelized with the runnable-level approach RunPar. The AUTOSAR stack Elektrobit tresos AutoCore Generic [23] runs the application. Additionally, one of the cores runs an Ethernet stack, which is used to exchange data with a host PC. The data exchange is carried out to monitor the behaviour of the application, e.g. to check if all parameters are calculated correctly. Moreover, we set the voltage for the core logic (using the microcontroller embedded voltage regulator) and display the core voltage, which is measured by an internal ADC.

In experiment C, we measure the lowest supply voltage at a predefined frequency, for which the core logic is still operational. Therefore, the voltage is lowered until the microprocessor does not operate in a normal manner any more. Only the internal flash and SRAM are used as memories. The bus/NoC frequencies are scaled according to the core frequencies. Hence, the full system scales linearly.

Criteria for normal operation are correct results and a working connection to the PC via Ethernet. The microcontroller runs for several seconds on each voltage level. In cases of an insufficient voltage setting, the microcontroller stops working at once. Thus, stable operation can be assumed after several seconds of operation. We are well aware that the microprocessor is operated out of its specification and hence the failure rate is higher under these conditions. For all frequencies, the same relative safety margin of $30\,\%$ for the voltage is used; as observed at $200\,\mathrm{MHz}$. The lowest observed working voltages here is $1.0\,\mathrm{V}$, whereas the specification demands $1.3\,\mathrm{V}$.

In the subsequent experiments D and E, we are interested in energy-saving potential with Parcus on the AURIX. Therefore, the measured operating points of experiment C are interpolated over the full operational range of the processor. The resulting energy-saving for the processor frequencies got in experiments A and B are computed.

2) Results of Experiment C (Measurement of lowest working core voltage): The measured lowest working voltages are listed in column 2 of table I, for the frequency listed in column 1. The resulting supply voltage with safety margin is listed in column 3. We assume these values for further experiments. Column 4 lists the energy per operation relative to the $1.3\,\mathrm{V}$ operating point. The energy per operation is proportional to the square of the supply voltage. The results in table I show that lowering

Table I: Measured lowest working core voltage depending on frequency.

Frequency	Voltage	Voltage + margin	Relative energy
200 MHz	1.00 V	1.30 V	*1.00*
175 MHz	0.96 V	1.25 V	*0.92*
150 MHz	0.93 V	1.21 V	*0.86*
120 MHz	0.88 V	1.12 V	*0.77*
100 MHz	0.86 V	1.12 V	*0.74*
75 MHz	0.86 V	1.12 V	*0.74*
50 MHz	0.86 V	1.12 V	*0.74*

the frequency from $200\,\mathrm{MHz}$ to $100\,\mathrm{MHz}$ allows reducing the supply voltage for the core logic by about $14\,\%$. This reduction of the supply voltage leads to $26\,\%$ less energy per operation. However, further reducing the frequency, i.e. below $100\,\mathrm{MHz}$, does not give any additional headroom to further lower the core logic supply voltage and the energy per operation. Hence, the potential of energy-saving by voltage-frequency scaling is limited for this microcontroller to $26\,\%$.

3) Results of Experiment D (Energy-saving with Task- & Latency-Fitting): Figure 8 shows the cubic Hermite spline interpolation [24] of the voltage-frequency scaling of the AURIX based on the measurements conducted in experiment C, listed in table I. This curve is used to determine the energy-

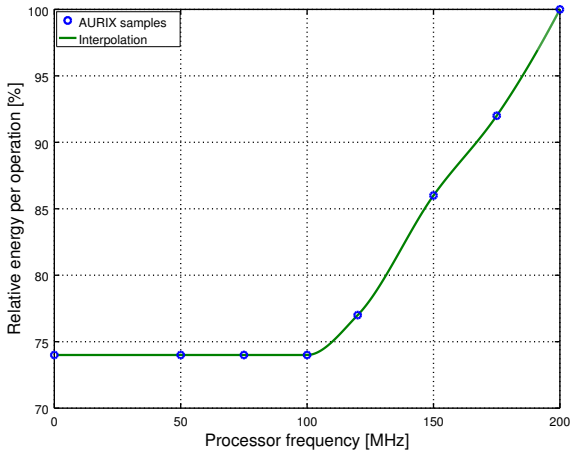


Figure 8: Interpolation of the relative energy consumption per instruction on the Infineon AURIX.

saving on a real platform. At this point, we are interested in the slope when one parallelization method results in a frequency reduction. Therefore, the values of $f^0_{\min}$, $f^6_{\min}$, and $f^{10}_{\min}$ from fig. 6 are used to compute the relative energy per instruction. Figure 9 shows the resulting curves that are named P_0, P_6, and P_{10}, respectively. Based on the results in fig. 6,

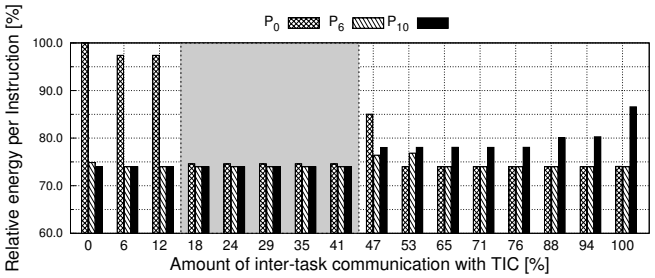


Figure 9: Relative energy consumption per instruction after adjustment for robustness by task- and latency-fitting.

one would expect that differences of the frequency reduction, between different combinations, manifest in an equally sized difference in the energy-saving. Contrarily, the differences are much smaller and the energy-saving is equal when $18\,\%$ to $41\,\%$ of the inter-task communication is replaced by TIC (grey area). In this area, all combinations of runnable- and task-level parallelism fully utilize the available energy-saving margin of the AURIX. Replacing more inter-task communication by TIC (i.e. introduce more task-level parallelism) shrinks the energy-saving only in the case of P_{10}, because the adjustment for robustness increases the latency. In contras, only relying in task-level parallelism (P_0) only provides minor energy-saving potential when applied rarely, because most of the tasks need to be executed in sequential order.

4) Results of Experiment E (Energy-saving with Task-Fitting Only): Therefore experiment, the values of $f^0_{\min}$, $f^6_{\min}$, and $f^{10}_{\min}$ from fig. 7 are used to compute the relative energy per instruction. Figure 10 shows the resulting curves that are again named P_0, P_6, and P_{10}, respectively. One can see that the

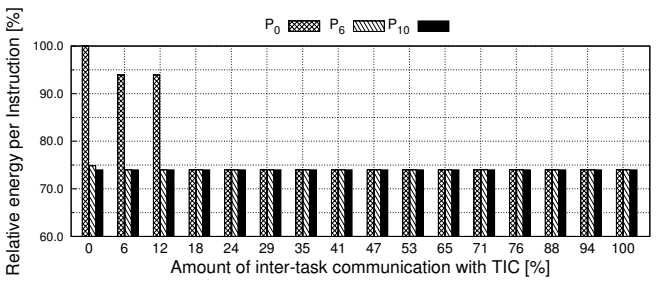


Figure 10: Relative energy consumption per instruction after task-fitting.

full energy-saving potential is exploited by all runnable- and task-level combinations, when at least $18\,\%$ of the inter-task communication uses TIC. Larger energy-saving, as one might expect from fig. 7, are not possible here. Additional measures must be taken to further reduce the energy consumption.

VII. Conclusion

This paper presented an energy- and latency-aware parallelization technique named *Parcus* that combines both runnable- and task-level parallelism. Parcus explicitly models the traversal of data from sensor to actuator through task instances, enabling to consider the latency imposed by parallelization techniques. This makes it possible to determine the latency and adjust the system to guarantee the worst-case. The success of the parallelization is quantified by the PSQ metric, which takes the latency and the processor frequency into account.

We demonstrated the applicability of Parcus with an automotive case study. Under ideal conditions, the processor frequency can be reduced by $73\,\%$ guaranteeing strict latency constraints. Furthermore, we measured the voltage-frequency curve of the Infineon AURIX multicore processor and found a maximal possible reduction of $26\,\%$ from voltage-scaling. When linked to the measured curve, we could observe that Parcus can fully utilize the processor's energy-saving potential.

Another crucial factor for parallelization is the mapping of labels to memory regions. Thus, our future work focuses on the memory mapping.

Acknowledgments

This research received funding from the EU FP7 no. 287519 (parMERASA), the ARTEMIS-JU no. 621429 (EMC^2), and the German Federal Ministry of Education and Research.

References

[1] Akihiro Hayashi, Yasutaka Wada, Takeshi Watanabe *et al.*, 'Parallelizing Compiler Framework and API for Power Reduction and Software Productivity of Real-time Heterogeneous Multicores', in *Languages and Compilers for Parallel Computing*, Springer, 2011.

[2] AUTOSAR GbR, 'AUTomotive Open System ARchitecture (AUTOSAR)', Standard v4.1, 2014.

[3] AC Rajeev, Swarup Mohalik, Manoj G Dixit *et al.*, 'Schedulability and End-to-end Latency in Distributed ECU Networks: Formal Modeling and Precise Estimation', in *10th ACM SIGBED EMSOFT*, ACM, 2010.

[4] Marco Di Natale, Wei Zheng, Claudio Pinello *et al.*, 'Optimizing End-to-end Latencies by Adaptation of the Activation Events in Distributed Automotive Systems', in *13th IEEE RTAS*, IEEE, 2007.

[5] Miloš Panić, Sebastian Kehr, Eduardo Quiñones *et al.*, 'RunPar: An Allocation Algorithm for Automotive Applications Exploiting Runnable Parallelism in Multicores', in *Proc. IEEE/ACM/IFIP CODES+ISSS*, New York: ACM Press, 2014.

[6] Martin Lowinski, Dirk Ziegenbein and Sabine Glesner, 'Splitting Tasks for Migrating Real-time Automotive Applications to Multi-core ECUs', in *Proc. IEEE SIES*, 2016.

[7] Sebastian Kehr, Eduardo Quiñones, Bert Boeddeker *et al.*, 'Parallel Execution of AUTOSAR Legacy Applications on Multicore ECUs with Timed Implicit Communication', in *52nd ACM/EDAC/IEEE DAC*, 2015.

[8] Julien Hennig, Hermann von Hasseln, Hassan Mohammad *et al.*, 'Towards Parallelizing Legacy Embedded Control Software Using the LET Programming Paradigm', in *2016 IEEE RTAS*, 2016.

[9] Infineon, *AURIX - TC27x B-Step, 32-bit Single-Chip Microcontroller, User's Manual, v14.1*.

[10] Julien Forget, Frédéric Boniol, Emmanuel Grolleau *et al.*, 'Scheduling Dependent Periodic Tasks Without Synchronization Mechanisms', in *Proc. 16th IEEE RTAS*, IEEE, 2010.

[11] Nico Feiertag, Kai Richter, Johan Nordlander *et al.*, 'A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems Under Different Path Semantics', in *CRTS Workshop*, 2008.

[12] Daniel Cordes, Peter Marwedel and Arindam Mallik, 'Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming', in *Proc. IEEE/ACM/IFIP CODES+ISSS*, New York: ACM Press, 2010.

[13] Sebastian Kehr, Miloš Panić, Eduardo Quiñones *et al.*, 'Supertask: Maximizing Runnable-level Parallelism in AUTOSAR Applications', in *DATE*, IEEE, 2016.

[14] Marco Di Natale and John A Stankovic, 'Applicability of Simulated Annealing Methods to Real-time Scheduling and Jitter Control', in *Proc. 16th IEEE RTSS*, IEEE, 1995.

[15] H.R. Faragardi, B. Lisper, K. Sandstrom *et al.*, 'An Efficient Scheduling of AUTOSAR Runnables to Minimize Communication Cost in Multi-core Systems', in *IST*, 2014.

[16] Ernest Wozniak, Asma Mehiaoui, Chokri Mraidha *et al.*, 'An Optimization Approach for the Synthesis of AUTOSAR Architectures', in *ETFA*, IEEE, 2013.

[17] Sönke Hartmann, 'A Self-adapting Genetic Algorithm for Project Scheduling Under Resource Constraints', *Naval Research Logistics (NRL)*, vol. 49, no. 5, 2002.

[18] Rainer Kolisch and Sönke Hartmann, 'Experimental Investigation of Heuristics for Resource-constrained Project Scheduling: an Update', *EJOR*, vol. 174, no. 1, 2006.

[19] John Henry Holland, *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[20] Rainer Kolisch and Sönke Hartmann, 'Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis', in *Project Scheduling: Recent Models, Algorithms and Applications*, Jan Węglarz, Ed. Boston, MA: Springer US, 1999.

[21] Haluk Ozaktas, Christine Rochange and Pascal Sainrat, 'Automatic WCET Analysis of Real-Time Parallel Applications', in *WCET*, 2013.

[22] Theo Ungerer, Christian Bradatsch, M Gerdes *et al.*, 'parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability', in *Euromicro DSD*, 2013.

[23] Elektrobit Automotive GmbH. (12th Oct. 2016). EB tresos AutoCore - Elektrobit Automotive, [Online]. Available: https://www.elektrobit.com/products/ecu/eb-tresos/autocore/.

[24] Carl De Boor, 'A practical guide to splines', in, ser. Mathematics of Computation 149. 1978, vol. 27, ch. IV Piecewise Cubic Interpolation.