Partitioning Embedded Real-Time Control Software based on Communication Dependencies

Martin Lowinski[1,2], Dirk Ziegenbein[1], and Sabine Glesner[2]

[1] Robert Bosch GmbH, Corporate Research, Robert-Bosch-Campus 1, 71272 Renningen
[2] Technische Universität Berlin, Ernst-Reuter-Platz 7, 10587 Berlin, Germany

Abstract. Electronic Control Units (ECUs), such as for automotive engine control, execute highly interdependent software units. These software units and their interaction are optimized for single-cores and need to be parallelized for upcoming multi-core processors. In this paper we investigate how to leverage the parallelism of the physical environment for the parallelization of legacy control software. Key for efficient parallelization is the knowledge of the physically required data flow timing which is often more relaxed than the timing of the single-core implementation. As this knowledge is often not documented, a domain expert needs to be involved. We propose an iterative model-based approach that minimizes the evaluation effort of the domain expert when parallelizing. In our case study, using a real-world automotive engine control software, we show that the presented approach can exploit parallelism while guaranteeing a correct data flow timing.

Keywords: Real-Time, Control Software, Parallelization, Automotive

1 Introduction

Future automotive control algorithms become more and more sophisticated due to increasing comfort, safety, and power-train functionalities. The resulting complex functionality is implemented in embedded real-time control software with an increasing demand for computing power. Multi-core processors are the most promising solution to cope with the computational demand. Though, decades of ECU software development for single-core processors have left a huge amount of legacy software, structured in cyclically activated tasks. Using the concept of Logical Execution Time (LET) [7], tasks can run in parallel with a deterministic behavior independent of their distribution to cores. However, even single tasks which consist of a growing number of software units may exceed the computational power of a single core. Thus, such a task has to be parallelized by reassigning the software units to multiple parallel tasks. But control software is very sensitive to timing-relevant changes like these, as they can cause an incorrect system behavior. Also, control software in the automotive domain is composed of highly interdependent software units, cf. Fig. 1 for a real-world example. When partitioning, dependencies have to be maintained by synchronization to ensure the legacy data flow. This limits the parallizability and is prohibitively expensive in terms of synchronization overhead.

Our solution approach thus suggests to leverage domain knowledge of experts to identify where a dependency can be relaxed in terms of its timing. This may lead to a behavior which is different from the legacy implementation. A domain expert has

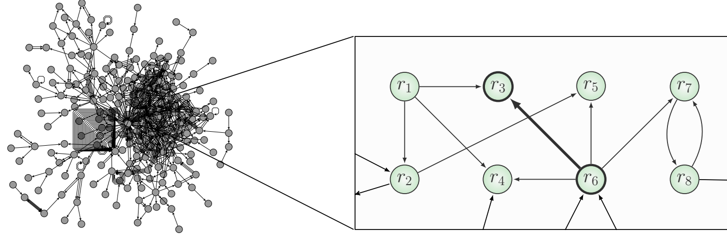


Fig. 1: A typical task in an industrial embedded real-time control software.

to evaluate if this behavior is still correct. Though, evaluating *all* dependencies is unfeasible due to time and cost reasons. In contrast, a small number of dependencies, as depicted in Fig. 1 on the right side, can be evaluated. The main thesis of this paper is that there are dependencies which are more relevant and probable to ease the partitioning challenge and thus more promising candidates for evaluation than others. The main contribution is a model-based workflow to determine these evaluation candidates through a machine-assisted iterative approach. In our case study, we have evaluated a real-world gasoline Engine Management System (EMS), which is one of the most complex control software applications in the automotive domain. Our approach allows to find a correct and suitable degree of parallelism with reduced evaluation effort.

The rest of this paper is organized as follows: In the next section we present the background related to the used models in our approach. Section 3 explains our workflow and its algorithms in detail. Experimental results are presented in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Background

In this section we introduce the system model and its graph representation used for our approach. We use Amalthea [9] as an example model in the presented approach because of its focus on parallel applications. In general, other models with the same semantics, e.g. Autosar [2] can be used as well.

2.1 System Model

Amalthea is a model and tool platform for automotive embedded-system engineering and provides integration into established industrial development processes. In Amalthea, a software unit is called runnable and a set of communicating runnables provides a desired functionality. A runnable $r \in \mathbf{R}$ is an encapsulated portion of sequential code, i.e., a void-void function. The runnables communicate indirectly via labels. A label $d \in \mathbf{D}$ can be of different data type, e.g. bit, array, or characteristic map and is stored in shared memory. Large data types such as a map are typically read-only and only segments are accessed. Each read-write and write-read relation between two runnables via a label is a dependency $q \in \mathbf{Q} \subseteq (\mathbf{R} \times \mathbf{R})$. A dependency between two runnables $r_i, r_j \in \mathbf{R}$ is denoted as $q_{i,j}$ and distinct through the

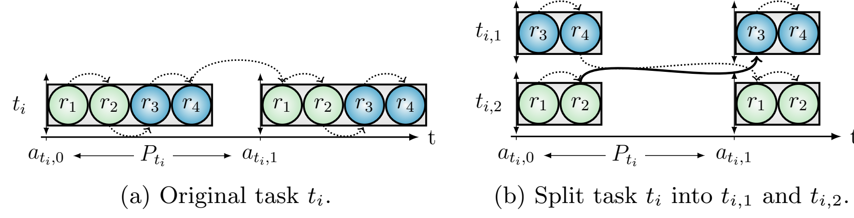
corresponding label d_k. The unit of scheduling for the underlying operating system is a task $t_i \in \mathbf{T}$ that manages the execution of runnables. Runnables are mapped to a task by the function $m : \mathbf{R} \rightarrow \mathbf{T}$. Let t_i be a task, then the order of execution of runnables is defined as the totally ordered set $\mathcal{P}(\mathbf{R}_{t_i}, \prec)$. If $r_a \prec r_b$ (r_a precedes r_b), then runnable r_a terminates its execution before r_b is executed. A task is cyclically activated with a period P_{t_i}, has a relative deadline D_{t_i} that is typically equal to P_{t_i}, and releases an infinite number of jobs $J_{t_i,k}, k \in \mathbb{N}$. Each job $J_{t_i,k}$ of task t_i is activated at $a_{t_i,k+1} = a_{t_i,k} + P_{t_i}$ with $a_{t_i,0} = 0$. The execution time of a task t_i is denoted as I_{t_i}. It is the sum of the execution time[3] of all runnables managed by task t_i. A task t_i can be split into multiple *task partitions* $t_{i,k}, k \in \mathbb{N}$. $N_{t_i,k}$ is hereby the ratio of execution time of partition $I_{t_i,k}$ to execution time of task I_{t_i}.

We represent the AMALTHEA model as a graph. The runnable graph is a cyclic multigraph $G_R = (\mathbf{V}, \mathbf{E})$ that describes the communication dependencies between runnables. It consists of nodes $\mathbf{V} = \mathbf{R}$ and the multiset of edges $\mathbf{E} = \mathbf{Q}$.

2.2 Communication

For the communication between tasks, we use the concept of Logical Execution Time (LET) [7]. For a job $J_{t_i,k}$, the LET starts with its activation $a_{t_i,k}$ and ends with the next activation $a_{t_i,k+1}$. The communication between jobs happens logically instantaneous at fixed points in time: at the beginning and end of each LET. Thus, on the task-level LET provides determinism such that the same output is produced from the same input independent of distribution, workload, or exact task execution timing. To express timing of the communication between runnables, we annotate dependencies with the communication behavior of LET. For runnables of one task that communicate with runnables of another task, LET introduces a communication latency. This latency is the result of the delayed publication of the produced data. Let m be a mapping, such that runnable r_a is mapped to producer task $m(r_a) = t_p$ and runnable r_b to consumer task $m(r_b) = t_c$. The runnables communicate via a dependency $q_{a,b}$. Then, the publication $pub(q_{a,b})$ of the data in job $J_{t_p,k}$ is delayed until $a_{t_i,k+1}$. Likewise, the communication is delayed for two runnables that are mapped to the same task $m(r_a) = t_i$ and $m(r_b) = t_i$ but are executed in opposite order $r_b \prec r_a$. The forward communication dependencies inside a job are not affected by LET since they are typically realized as shared variables and are read/written inside the runnable context. Thus, the produced data of these dependencies is published instantly. The example in Fig. 2 shows two consecutive job instances of the task t_i split into two task partitions $t_{i,1}$ and $t_{i,2}$. For all communication dependencies (illustrated as dotted lines), except between runnable r_2 and r_3 (illustrated as solid line), the timing does not change when the task is split. In this case, the overall communication latency from r_1 via r_2 and r_3 to r_4 is doubled from P_{t_i} to $2 * P_{t_i}$ as a result of the task splitting. An expert has to evaluate if this additional latency maintains a correct behavior. We define that a dependency $q_{a,b} \in \mathbf{Q}$ has an associated criticality $crit : \mathbf{Q} \rightarrow \{\text{critical}, \text{uncritical}\}$.

[3] Depending on the application characteristic, AMALTHEA provides the best, mean (e.g. for load balancing), and worst case (e.g. for hard real-time) execution times.



(a) Original task t_i.

(b) Split task t_i into $t_{i,1}$ and $t_{i,2}$.

Fig. 2: Communication based on LET when a task is split.

It represents the timing constraint for the communication between the producer r_a and consumer r_b when data is produced in job $J_{t_i,k}$ for a mapping m, such that $m(r_a) = t_i$ and $m(r_b) = t_j$. A dependency $q_{a,b}$ is considered *uncritical* if the communication *allows* additional latencies through LET, i.e., $pub(q_{a,b}) = a_{t_i,k+1}$. Otherwise they are considered *critical*, i.e., $pub(q_{a,b})$ is instantly. Relaxing the criticality of a dependency is the verified transformation from critical to uncritical.

The cluster graph G_C is the transformed graph of G_R such that all uncritical dependencies are removed. Each subgraph in G_C represents a *task cluster* $c \in \mathbf{C}$. Task clusters are the graph representation of task partitions. All runnables inside a cluster have critical dependencies and are thus functionally (in respect to timing) dependent on each other.

3 Machine-assisted Dependency Analysis

Our *expert-in-the-loop workflow* determines the criticality of dependencies in which the expert and the machine are in an interactive loop. The input is an Amalthea task with communication based on LET and a parallel target scenario. Initially, all dependencies of this task are critical. During the workflow, the expert is guided toward a parallel target scenario by evaluating one dependency for its criticality on every iteration. One iteration in our workflow consists of five steps, cf. Fig. 3:

1. Every unevaluated dependency is analyzed according to a number of criteria.
2. The criteria results of each dependency are aggregated. A dependency becomes a candidate if it has a potential impact toward the given parallel target scenario and has a high probability to be uncritical.
3. The analysis proposes the dependency candidate that is most suitable and probable for evaluation to the domain expert.
4. The domain expert evaluates whether the criticality of the dependency candidate can be relaxed.
5. The resulting, potentially relaxed, criticality is refined in the model.

The parallel scenario is computed via a bin-packing approach on every workflow iteration. A parallel scenario $s \in \mathbf{S}$ is defined as the execution time ratios of all task partitions $N_{t_i,k}$. E.g. a scenario with two equally sized task partitions is denoted as $s = (N_{t_i,1}, N_{t_i,2}) = (0.5, 0.5)$. The parallel *target* scenario s_t allows a variation V_{t_i} of the task partitions to the target. For example, if two equally sized task

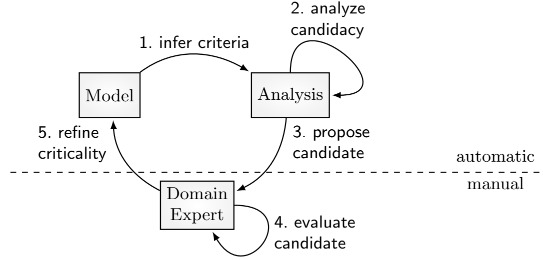


Fig. 3: Iterative, semi-automatic workflow.

partitions with a variation of 10 % are allowed, we denote the target scenario as $s_t = ((N_{t_{i,1}}, N_{t_{i,2}}), V_{t_i}) = ((0.5, 0.5), 10\,\%)$. The workflow is finished as soon as the s_t is reached. The workflow also respects more fine-grained partitions because partitions can always be merged. E.g. the workflow halts if it finds a scenario with $(0.45, 0.24, 0.31)$. In the following, we present steps 1–5 of our workflow.

3.1 Criteria

In the first step of our workflow, the dependencies are quantified according to certain criteria which evaluate these properties. There are two types of criteria: 1) An *impact* criterion indicates the benefit toward the parallel target scenario if the criticality is relaxed. It is the normalized function $impact : \mathbf{Q} \rightarrow [0, 1] \in \mathbb{R}$. The benefit is relative to the impact: an impact of $]0, 1]$ is beneficial, an impact of 0 indicates no benefit. 2) A *probability* criterion indicates the probability of the criticality. It is the function $prob : \mathbf{Q} \rightarrow [0, 1] \in \mathbb{R}$, i.e., if a dependency is likely critical it is in $[0, 0.5[$ or if it is likely uncritical it is in $]0.5, 1]$. If the criterion cannot be decided to either criticality, it is 0.5. We present five criteria in the following subsections.

Minimum Feedback Arc Set (MFAS). For a correct execution, dependency cycles of the task have to be resolved. Such cycles can be resolved by relaxing the criticality of at least one dependency in the cycle. To reduce the evaluation effort, it is important to find the smallest set of dependencies (MFAS) to resolve. As shown in (1), the criterion $impact_{\mathrm{MFAS}}$ is 1 for dependencies that are part of this set as they reduce the evaluation effort.

$$impact_{\mathrm{MFAS}}\,(q_{i,j}) = \begin{cases} 1, & \text{if } q_{i,j} \in \mathbf{MFAS} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Strongly Connected Component (SCC). Even if a dependency is not part of MFAS, it can be a member of a cycle. To find cycles, i.e., SCCs in our graph we use the algorithm by K. A. Hawick and H. A. James [6]. For each SCC found by the algorithm, every containing runnable is reachable from every other runnable via

one or more dependencies. Relaxing the criticality of such a dependency may resolve many cycles at once and reduces the evaluation effort. The number of cycles a dependency can resolve $\#cycles(q_{i,j})$ is determined by the number of SCCs the dependency is a member of. Thus, the criterion $impact_{\mathrm{SCC}}$ is the ratio of the number of cycles the dependency would resolve to the total number of cycles in the graph $\#cycles(G_R)$:

$$impact_{\mathrm{SCC}}\,(q_{i,j}) = \frac{\#cycles(q_{i,j})}{\#cycles(G_R)},\;\; \#cycles(G_R) > 0 \tag{2}$$

Forward Evaluation (FE). Relaxing the criticality of a dependency can create new task clusters. Therefore, we can evaluate the potential benefit for the number of task clusters if the criticality of the dependency is uncritical. The Forward Evaluation (FE) impact assumes that the dependency candidate is uncritical. It measures the number of task clusters with and without this assumption. The resulting boolean variable $N_{q_{i,j}}$ is 1 if the dependency creates an additional task cluster, otherwise 0. One uncritical dependency can create one additional cluster at most. Therefore, we can differentiate further. The impact is relative to the balance of the created clusters and its consisting runnables. We define the balance of the clusters based on the sum of the runnables' execution time per cluster $I_{t_i,k}$. The clusters are equally balanced if the standard deviation of the execution time per cluster $\sigma(I_{t_i,k})$ is zero. The aggregated criterion $impact_{\mathrm{FE}}$ is

$$impact_{\mathrm{FE}}\,(q_{i,j}) = \begin{cases} 1/\sigma(I_k), & \text{if } N_{q_{i,j}} = 1 \text{ and } \sigma(I_k) \neq 0\;\;. \\ 0, & \text{otherwise}\;\;. \end{cases} \tag{3}$$

Dependency Classification (DC). The values communicated between runnables via dependencies have different dynamics. For example, crank-angle based values change with higher dynamics while temperature values are steadier over time. Hence, relaxing the criticality of a dependency that communicates slowly altering values may have only a neglecting effect on the behavior. Thus, such a dependency has a high probability to be uncritical. Based on this domain-specific knowledge, control-engineers can create classes $c_i \in C$ prior to the analysis. A class features dependencies of the same dynamics and is assigned a specific probability P that a member is uncritical. This also shows how probable it is that a manual evaluation of such a dependency by a domain expert confirms this assumption. Thus, dependency classes with a high probability that members are uncritical reduce the evaluation effort. The classes are created from the associated label of a dependency which represents an ECU variable. ECU variables are described using A2L [1], the standard description format for measurement and calibration data in the automotive industry. It describes e.g. the data type, format, and computation method is used for measurement and calibration purposes. A control engineer interprets this information according to its dynamics and creates the classes prior to the analysis. The key to each class is the A2L information such that the function $class : \mathbf{Q} \rightarrow C$ can identify the corresponding class of a dependency during the

analysis. The Dependency Classification (DC) criterion $prob_{\mathrm{DC}}$ of a dependency is the probability of the associated class:

$$prob_{\mathrm{DC}}\left(q_{i,j}\right) = P\left(class\left(q_{i,j}\right)\right) \tag{4}$$

Reference Implementation (RI). The criterion RI is a special kind of probability criterion as it can propose reference criticalities of a verified single- or multi-core implementation with absolute certainty. This is useful for two scenarios: 1) To create an initial model with verified criticalities for further analysis. 2) As fallback and reference of the legacy control software for the domain expert. The result of this criterion is safe because AMALTHEA models provide a verified order of execution that has been integrated and tested on real ECUs. According to the fixed order of execution, the following information can be inferred for a dependency $q_{a,b}$: If $r_b \prec r_a$, then the dependency $q_{a,b}$ was considered uncritical in the legacy model, otherwise critical. This information is proposed to the domain expert in the form of $prob_{\mathrm{RI}}$: If $r_b \prec r_a$, then the result for this dependency is 1 (uncritical), else it is 0 (critical)—cf. (5). Via the Reference Implementation, all dependencies can be evaluated for their criticality. Cyclic dependencies as well as other uncritical dependencies that are based on domain knowledge can be resolved.

$$prob_{\mathrm{RI}}\left(q_{i,j}\right) = \begin{cases} 1, & \text{if } r_i \succ r_j \ . \\ 0, & \text{if } r_i \prec r_j \ . \end{cases} \tag{5}$$

3.2 Analysis

After evaluating the criteria of each dependency, the results are aggregated in this second step. Each dependency is hereby analyzed for candidacy which is represented by a score. The $score(q_{a,b})$ of a dependency is the product of the total impact times the total probability:

$$score\left(q_{a,b}\right) = \prod_{i=0}^{n} prob_i(q_{a,b}) * \sum_{i=0}^{n} w_i * impact_i(q_{a,b}) \tag{6}$$

To evaluate the total impact of a dependency, the impact criteria are weighted. During the workflow these weights change and thus control the analysis. For example, in the beginning of the workflow the main objective besides the target scenario is to resolve cycles. For that purpose, the analysis adjusts the weights w_i to favor the MFAS and SCC criteria. As soon as all cycles are resolved, the two criteria will yield no further impact. The product of all probability criteria estimates how likely it is that a dependency is evaluated as uncritical. For example, if a dependency originally was critical based on the RI criterion, but the DC criterion yields a probability of 0.8, the dependency is a more probable candidate. Depending on the score, a criticality is proposed. The score and the dependency create an evaluation candidate. These candidates are inserted into a prioritized queue. The evaluation candidate with the highest score is proposed to the domain expert in step three of our workflow. For the current iteration, the candidate with the highest score represents the most suitable and probable dependency for an evaluation.

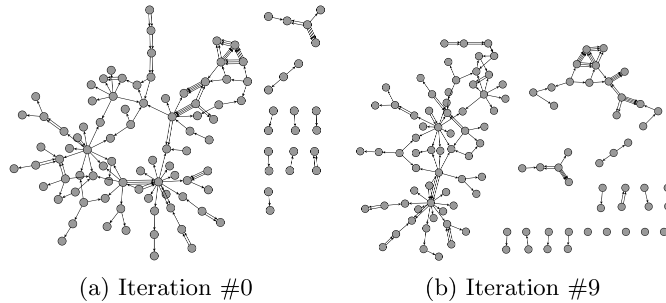
3.3 Expert Evaluation and Refinement

In step four, the domain expert evaluates if the proposed dependency candidate allows a relaxed timing. The dependency candidate is part of a controller with a certain functionality. When relaxing the timing of this dependency, the communicated data is delayed (as shown in Section 2.2). This delay increases the reaction time of the controller that potentially alters the functionality: Depending on the controller design, the delay may violate latency or stability requirements, and other performance criteria. But a complete specification of these requirements for each single controller and their interactions is not available, such that a manual evaluation by an expert in this domain is mandatory. For the evaluation, our workflow presents the dependency candidate together with the criteria values to the expert. The criteria values give hints why this dependency candidate is the most suitable and probable one among the others. This may influence how the expert evaluates the dependency candidate. There are various techniques to evaluate the execution timing impact on the functionality of the controller such as formal verification or simulation [13]. Based on this decision, the resulting potentially relaxed criticality is added to the model and finishes the current workflow iteration.

4 Case Study

We have implemented the presented approach as an AMALTHEA [9] toolchain element and evaluate it on a real-world Engine Management System (EMS) which is a complex example of embedded real-time control software. For this case study we have selected a heavy task t_e with 234 runnables and 248 communication dependencies. For simplicity reasons, all cycles are resolved based on the reference implementation. Figure 4a shows the initial task cluster distribution. The biggest cluster has 65 % of the task's execution time and is thus the main focus for the cluster impact factors; all other clusters are smaller than 4.9 %. The parallel target scenario we want to achieve in this study consists of two equally sized task clusters with an allowed variation of 2 %, so $s_t = ((0.5, 0.5), 2\,\%)$. Due to the complexity of this EMS, we would need to involve many different experts to evaluate the functional impact on each controller. To efficiently assess our approach, we therefore emulate the experts based on the Dependency Classification criterion. For each proposed dependency, the emulated expert decides randomly with the probability $prob_{DC}$ if the dependency is uncritical. This follows our argument that $prob_{DC}$ indicates how the expert would decide. In the future we plan to test our approach with different domain experts and compare the results. To assess the benefit of our approach, the key metric is the number of workflow iterations. Assuming that it takes a fixed amount of work for the expert to evaluate a dependency, this metric represents the evaluation effort. For comparison, we executed the same workflow but selected dependencies for evaluation at random. Both approaches were executed 100 times.

The results in Fig. 4a and 4b show how our approach partitioned the task t_e. Note that 128 clusters with a size of one runnable are left out for illustration reasons. On average, after 9.4 (min: 8, max: 24) iterations, our approach finds a partitioning that satisfies the parallel target scenario. In comparison, when



(a) Iteration #0 (b) Iteration #9

Fig. 4: Task t_e in comparison from iteration #0 to iteration #9.

dependencies were selected randomly, it took approximately 72 (min: 17, max: 123) iterations with the expert on average to reach the target scenario. Note that one dependency is evaluated by the expert in one iteration. This shows that our approach successfully guides the selection of dependencies for evaluation and significantly reduces the number of iterations and thus the amount of manual work for reaching the parallelization goal.

5 Related Work

Based on the abstraction level of tasks, the problem of partitioning has been studied in the following contributions. Using the Amalthea platform, Höttger et al. [8] describe and evaluate partitioning of weighted directed acyclic graphs (WDAGs) considering the system's critical path (CP) or by applying earliest start scheduling (ESS). For the Autosar ecosystem, Faragardi et al. [5] present partitioning and mapping techniques with the goal to minimize the inter-runnable communication time through evolutionary algorithms. In contrast, Panić et al. [11] propose the idea to allocate runnables based on a variant of the worst-fit decreasing heuristic directly to cores. Using Integer Linear Programming (ILP), the work of Saidi et al. [12] also maps runnables from Autosar applications directly to cores, optimizing the load-balance and minimize the communication effort. With the Hierarchical Task Graph (HLT) described by Cordes et al. [4], the execution time and energy consumption is optimized at the instruction level with ILP and Genetic Algorithms (GAs). The semi-automatic approach of the MPSoC Application Programming Studio (MAPS) [3] assists a programmer in developing parallel C applications by combining machine analysis and domain knowledge to suggest partitions to the programmer. The approach by Jahr et al. [10], also semi-automatic, features a model-based way to parallelize existing legacy software with Activity and Pattern Diagrams (APD) by searching for Parallel Design Patterns (PDP) in the sequential code.

In contrast to the above related approaches that maintain dependencies through synchronization, our approach relaxes the timing of dependencies to increase parallelizability. The relaxed dependencies could then be used as input for the discussed approaches and yield improved results.

6 Conclusion

In this paper, we have presented an approach to ease the partitioning challenge of control software by relaxing the timing constraints of dependencies. Relaxed timing constraints increase the efficiency of partitioning techniques and reduce synchronization overhead. Our main contribution is the semi-automatic and iterative workflow to find the most beneficial and probable dependencies for evaluation by a domain expert. Our analysis leverages domain knowledge and thus the parallelism of the physical environment. We have demonstrated our approach by partitioning a heavy task of a real-world Engine Management System (EMS) by selecting and evaluating only 3.6 % of the dependencies through an expert.

As a future work, we plan to include additional criteria such as end-to-end latency constraints over chains of runnables. We also consider a possible propagation of criticality along dependency chains based on static code analysis.

References

1. ASAM MCD-2 MC V1.7.0, `http://www.asam.net/nc/home/standards/standard-detail.html?tx_rbwbmasamstandards_pi1[showUid]=3078`.
2. AUTomotive Open System ARchitecture, `http://www.autosar.org`.
3. J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An integrated framework for MPSoC application parallelization. In *DAC*, pages 754–759, 2008.
4. D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *CODES+ISSS*, pages 267–276, 2010.
5. H. R. Faragardi, B. Lisper, K. Sandström, and T. Nolte. A communication-aware solution framework for mapping autosar runnables on multi-core systems. In *ETFA*, pages 1–9, Sept 2014.
6. K. A. Hawick and H. A. James. Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In *FCS*, pages 14–20, Las Vegas, USA, 2008. CSREA.
7. T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software*, pages 166–184, London, UK, 2001. Springer-Verlag.
8. R. Höttger, L. Krawczyk, and B. Igel. Model-based automotive partitioning and mapping for embedded multicore systems. *International Journal of Computer, Control, Quantum and Information Engineering*, 9(1):268–274, 2015.
9. ITEA2 Project: AMALTHEA (ITEA2-09013), `http://amalthea-project.org`.
10. R. Jahr, M. Gerdes, and T. Ungerer. A pattern-supported parallelization approach. In *PMAM*, pages 53–62, New York, NY, USA, 2013. ACM.
11. M. Panić, S. Kehr, E. Quiñones, B. Boddecker, J. Abella, and F. J. Cazorla. RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. In *CODES+ISSS*, pages 29:1–29:10, New York, NY, USA, 2014. ACM.
12. S. E. Saidi, S. Cotard, K. Chaaban, and K. Marteil. An ILP approach for mapping AUTOSAR runnables on multi-core architectures. In *RAPIDO*, pages 6:1–6:8, New York, NY, USA, 2015. ACM.
13. D. Ziegenbein and A. Hamann. Timing-aware control software design for automotive systems. In *DAC*, pages 56:1–56:6, New York, NY, USA, 2015. ACM.