An analytical approach for calculating end-to-end response times in autonomous driving applications

Lukas Krawczyk, Mahmoud Bazzal, Ram Prasath Govindarajan, Carsten Wolff
IDiAL Institute
Dortmund University of Applied Sciences and Arts
44227 Dortmund, Germany
lukas.krawczyk@fh-dortmund.de

***Abstract*—This work proposes a solution for the WATERS industrial challenge 2019. The first part addresses the response time analysis challenge and covers analyzing the end-to-end latency of the given application by evaluating the critical path from its sensor tasks to actuator tasks under the assumption of an (i) implicit and (ii) LET based communication paradigm. The second part discusses the optimization of the applications end-to-end latency using a genetic algorithm based approach by modifying (i) the application's allocation of tasks to cores as well as (ii) the time-slices for tasks allocated to the GPU.**

***Index Terms*—Real-time systems, response-time analysis, automotive, APP4MC**

I. Introduction

The demands on automotive computing platforms are continuously rising due to the increasing amount of software that is driven by new automotive functionalities. In order to cope with these needs, future E/E architectures will consists of a variety of heterogeneous applications with unique characteristics, mixed levels of criticality and different models of computation, such as classical periodic control, event-based planning, or stream-based perception applications will typically be co-existing on the same hardware platform [10]. Deploying these applications will introduce several challenges, such as maintaining freedom from interference in safety-critical application, as required by the ISO 26262 standard, or meeting constraints such as timing requirements. The later is especially challenging due to the varying computational models.

The WATERS industrial challenge 2019 describes the prototype of such an end-to-end automotive driving application and proposes two challenges typically encountered in designing such systems. The application consists of 10 Tasks with 4 tasks that may further be accelerated by offloading to an accelerator (GPU). The heterogeneous hardware platform is split into two processor islands, with the first consisting of a dual-core NVidia Denver 2 CPU and the second of a quad-core ARM Cortex-A57 CPU. Additionally, the platform also integrates a 256-core NVidia Pascal GPU grouped into two streaming multiprocessors.

The research leading to these results has received funding from the Federal Ministry for Education and Research (BMBF) under Grant 01IS18047D in the context of the ITEA3 EU-Project PANORAMA.

The overall goal of the first challenge is to determine the application's end-to-end response time, e.g. the data propagation path from its sensor tasks to its actuator tasks. The second challenge addresses the problem of minimizing the response time of task chains by optimizing the application's deployment to the underlying hardware platform (NVIDIA Jetson TX2 SoM).

This paper is organized as follows. Our assumptions for the remainder of this paper are introduced in Section II, followed by the system model along with its notation and terminology in Section III. Our proposed solution to the analysis challenge is addressed in Section IV and presents the approach for determining the application worst case response time. Section V presents our solution to the design space exploration (optimization) challenge, which is based on a genetic algorithm that optimizes the application's deployment towards a lower end-to-end response time by modifying the available degrees of freedom. Finally, VI discusses our results and concludes this paper.

II. Assumptions

The challenge states that all tasks, both for CPU and GPU, follow a read-execute-write semantic, which implies an e.g. implicit or LET communication paradigm. Accordingly, we will determine the resp. end-to-end latency for both communication paradigms, although we expect implicit communication to result in lower response times compared to LET as shown in [4]. Moreover, we assume that all tasks access the global memory (DRAM) exactly once at the beginning of their execution for a copy-in operation and also exactly once at the end of their communication for a copy-out operation, i.e. all cores (CPU, GPU) contain a local memory that stores local variable copies. Consequently, the time required for accessing these local variables is assumed to be included in a task's ticks and as such out of the scope of this paper.

Furthermore, we assume that all operations of the CPU as well as the GPUs Copy Engine (CE) and Execution Engine (EE) are fully preemptive and the resulting delays of concurrent accesses are covered by an additional contention overhead. All tasks are allocated to exactly one CPU core or the GPU. The allocation and duration of each kernels time slice is fixed at design time, i.e. tasks do not migrate at run-time. Finally, all tasks on the CPU follow a partitioned fixed priority scheduling policy whereas the GPU schedules its tasks using a weighted round-robin (WRR) policy with fixed time slices. Priorities for tasks are unique, i.e. two tasks with the same period have different fixed priorities.

Since this work considers tasks $\tau \in \mathcal{T}$ executed on heterogeneous processing units with different scheduling algorithms and different characteristics, we distinguish between tasks executed on the CPU and tasks offloaded to the GPU. Tasks that are executed on the CPU are further divided into tasks that perform *offloading* of some of its work to a GPU and *regular* tasks that do not perform offloading. An offloading task can be further subdivided into three phases: A (i) pre-processing phase, that prepares a given data set for its processing on the GPU, an (ii) offloading phase in which the task offloads some of its execution to the GPU and suspends itself until it receives a response from the GPU, and a (iii) post-processing phase that processes the resulting data set from the GPU. Moreover, offloading tasks can either perform synchronous or asynchronous offloading. In the *synchronous* case, the task will be actively blocking lower priority tasks while it is being executed. In the *asynchronous* case, lower priority tasks may be executed during its waiting phase at the cost of an asynchronous offloading overhead that increases the offloading tasks execution time. For the remainder of this work, we will apply asynchronous offloading if tasks with lower priority are executed on the same processing unit as the offloading task in order to allow lower priority tasks to be executing during the offloading tasks waiting phase. Synchronous offloading is applied if no lower priority tasks are co-scheduled on the same processing unit with the goal of minimizing the offloading tasks execution time by avoiding the asynchronous offloading overhead.

III. System Model

Each task that is executed on a **CPU** $\tau_i \in \mathcal{T}$ is described as tuple $\tau_i = (\{\tau_{i1}, \dots, \tau_{i|\tau_i|}\}, P_i, \pi_i)$ with its period P_i a unique priority π_i, and a list of $|\tau_i|$ sub-tasks τ_{ij}. A sub-task can either be executed on the CPU or offloaded to the GPU. In the former case, it is described as $\tau_{ij}^C = (C_{ij,\rho}, O_{ij}, J_{ij})$ with an offset O_{ij} and a jitter J_{ij}. The set $C_{ij,\rho} = \{C_{ij,\rho}^+, C_{ij,\rho}^-\}$ denotes a pair of execution times on processing unit ρ, with $C_{ij,\rho}^+$ being the best case execution time and $C_{ij,\rho}^-$ the worst case execution time.

A sub-task that will be offloaded to the **GPU** $\tau_{ij}^G \in \mathcal{T}^G$ is similarly described as tuple $\tau_{ij}^G = (C_{ij,\rho}, O_{ij}, J_{ij}, \phi_{ij})$, with ϕ_i being the length of its time slice for the WRR scheduling on the GPU. Although the challenge assumes the presence of a single GPU, some of the GPU tasks may be executed on the CPU. Accordingly, we maintain the concept of distinguishing between processing unit specific execution times for each processing unit ρ.

Our approach utilizes hyper periods of task sets, i.e. we address the k-th instance (job) of τ_i as $\tau_{i.k}$. The relative distance of a job's arrival to the beginning of the hyper period is represented by $r_{i.k}$, and the relative distance to its worst case response is denoted by $\mathcal{R}_i^-$. Data is passed between tasks using labels. A label $l \in \mathcal{L}_{ij}$ represents a variable that is accessed (i.e. either read or written) by a sub-task τ_{ij}.

Finally, a **task chain** $\sigma_i^n \in \mathcal{S}$ is denoted as a finite sequence of $n \geq 1$ tasks $(\tau_1, \dots, \tau_n)$ and represents the data propagation flow over the given task set, i.e. each path that can be constructed from a job $\tau_{m.\alpha}$ to a job $\tau_{m+1.\beta}$ needs to satisfy the criteria $r_{m+1.\beta} \geq r_{m.\alpha} + \mathcal{R}_m^-$, i.e. the absolute arrival of the successor needs to occur after the response of its predecessor.

IV. Proposed solution for analysis challenge

A. Task Chains

In order to determine the application's end-to-end response time, we have to derive longest data propagation path (*critical path*) from any of its sensor tasks (i.e. a task with no predecessor) to any of the actuator tasks (i.e. tasks with no successor).

A convenient approach for this is to analyze the communication graph spanned by runnables and their label accesses provided by the AMALTHEA model. However, the temporal behavior is ambiguous due to labels with multiple read- and write accesses from runnables executed by different tasks. For instance, the label *Cloud_map_host* is both read and written by runnables *Lidar_Function* (Task *Lidar_Grabber*) and *Localization_Preprocessing* (Task *PRE_Localization_gpu_POST*) (cf. Fig. 1a). Since both tasks can be executed in parallel, it is impossible to determine which task realizes a source of communication or a target. As a result, we decide to derive the communication flow based on the challenge's description in [3] and ensure data consistency by introducing transitive labels for task-to-task communication that ensure a proper data flow by limiting the number of writers per label to one (cf. Fig. 1b).

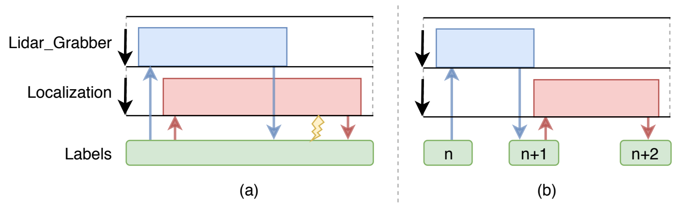

Fig. 1. Example of inconsistent data accesses due to multiple write sources to a label (a) and with consistent accesses by adding transitive labels (b)

Since mapping decisions may lead to an excessively or even indefinitely blocked task, each task within a data propagation path may increase the path response time. Accordingly, we consider all routes within the application from a sensor task to the actuator task and specify appropriate *task chains*. Possible sensor tasks for a critical path are identified as tasks without incoming edges, i.e. the tasks *Lidar_Grabber*, *Detection*, *CAN*, *SFM*, and *Lane_Detection*, whereas the sink is realized by *DASM*, which is the only task without outgoing edge.

The task chains, that become part of our analysis, are further specified in Tab. I.

TABLE I
IDENTIFIED TASK CHAINS

Task Chain	Tasks
σ_1	Lidar_Grabber $\Rightarrow$ Loc $\Rightarrow$ EKF $\Rightarrow$ Planner $\Rightarrow$ DASM
σ_2	CAN $\Rightarrow$ Loc $\Rightarrow$ EKF $\Rightarrow$ Planner $\Rightarrow$ DASM
σ_3	SFM $\Rightarrow$ Planner $\Rightarrow$ DASM
σ_4	Lane_detection $\Rightarrow$ Planner $\Rightarrow$ DASM
σ_5	Detection $\Rightarrow$ Planner $\Rightarrow$ DASM

B. End-to-end Latency

The applications end-to-end latency depends on the applied communication paradigm and the corresponding time at which data from one task is propagated to its follower in a task chain. In the following, we describe the approaches for determining a task-chains end-to-end latency for implicit and LET based communication paradigms that will be used for deriving the applications worst case end-to-end timing.

The worst case end-to-end latency of the application L^{E2E} can be determined by the highest latency among a set $\mathcal{S}$ of all task chains, formally denoted as $L^{E2E} = \max_{\sigma \in \mathcal{S}}(L^{TC}(\sigma))$ with L^{TC} being the function that determines the task chain worst case latency.

Implicit communication: In order to determine the applications end-to-end latency while assuming an implicit communication paradigm, we apply the approach from Kloda et al. in [5].

The worst case latency of a task chain σ^n is derived by analyzing the latency of each data propagation path that originates during the hyper period $H = lcm\{P_i | \tau_i \in \mathcal{T}\}$, which is sufficient due to the recurring execution and communication pattern.

$$L^{TC} = \max_{\forall k | kT_1 < H} L^{TC}(\sigma^n, kT_1) \tag{1}$$

As shown in [5] , an upper bound for the worst-case latency of task chains σ^n instance starting at time instant r_p is determined by Eq. 2, with the producer's release time r_p, the consumer's release time r_c, and a sub-chain σ^{n-1} that contains all tasks from σ^n except for the first element.

$$L^{TC}(\sigma^n, r_p) \leq \begin{cases} (r_c - r_p) + L^{TC}(\sigma^{n-1}, r_c) & n \geq 2 \\ \\ R_p^- & n = 1 \end{cases} \tag{2}$$

We illustrate this function in Fig. 2, which shows 3 tasks that form a task chain with a total latency of 14 time units. Naturally, data propagation is delayed due to different periods and arrival times of tasks. For instance, let us consider the first sub-chain consisting of a producer τ_p, represented by Task A with an release time $r_p = 54$, and a consumer τ_c, represented by Task B that arrives at $r_c = 56$. The latency due to different arrival times is $r_c - r_p = 2$ time units, and the task chains total latency becomes $2 + L^{TC}(\sigma^{n-1}, 56)$. In the second sub-chain, Task B becomes the producer τ_p and arrives at time instant $r_p = 56$, whereas the next consumer (Task C) capable of processing the producers output arrives at instant $r_c = 60$. Accordingly, the delay becomes $r_c - r_p = 4$ time units, plus the worst case response time of Task C (8 time units), leading to the chains total WCRT of 14 time units.

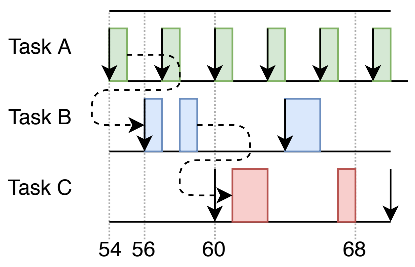

Fig. 2. Data propagation flow in a task chain with 3 tasks

The release time r_c of a consumer τ_c, is derived by Eq. 3. If both, the consumer as well as the producer are allocated to the same processing unit, and the producer has a higher priority compared to the consumer, any job of the consumer that satisfies $r_c \geq r_p$ will always be executed after the resp. producers job. Accordingly, a safe release time for the consumer that guarantees that it will read the most recent output of the producer is derived by the first case in Eq. 3. In all other cases, only a consumer that is released after the producer finishes its work, i.e. $r_c > r_p + \mathcal{R}_p^-$, will be capable of reading the producer's output.

$$r_c = \begin{cases} \left\lceil \dfrac{r_p}{T_c} \right\rceil T_c & \text{iff } \pi(\tau_p) > \pi(\tau_c) \text{ and } P(\tau_p) = P(\tau_c) \\ \\ \left\lceil \dfrac{r_p + \mathcal{R}_p^-}{T_c} \right\rceil T_c & \text{otherwise} \end{cases} \tag{3}$$

LET Communication: The applications end-to-end latency is determined using an adjusted variant from [5] that allowed an efficient implementation for this challenge. Due to the lack of space we omit a detailed description.

C. Response Time

Before we present an approach for determining the worst case response time in the given heterogeneous architecture, we need to analyze if and how tasks scheduled on different processing units impact each other. Therefore we illustrate the execution of a very simple heterogeneous example system with similar characteristics to the challenge in Fig. 3.

The system consists of four tasks $\tau_1 - \tau_4$ and three processing units Core1, Core2 and GPU. Tasks τ_2 and τ_3 denote regular tasks while τ_1 and τ_4 denote offloading tasks. An *offloading* task τ_x is further subdivided into

- τ_{x1}^C, being its **pre-processing** phase
- τ_{x2}^G, being its **offloading** or **suspension** phase that starts as soon as its sub-task is launched on the accelerator and lasts until the accelerator finished its execution, and
- τ_{x3}^C, being its **post-processing** phase that is instantly released once the task offloaded to the GPU finishes execution.

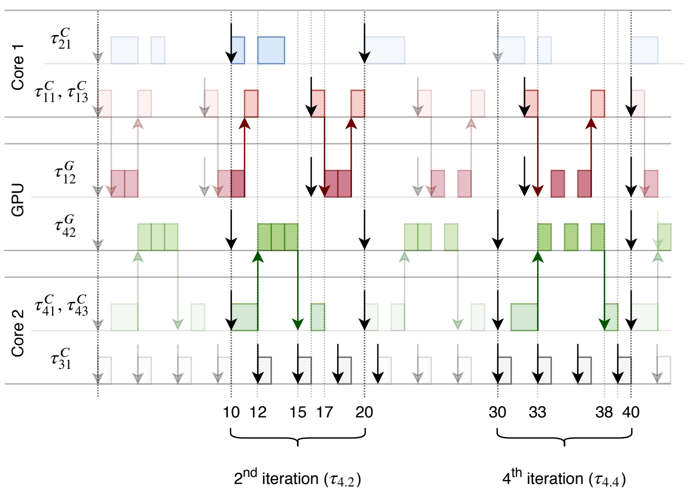

Fig. 3. Example of schedule with 3 tasks executed by 2 CPUs and 2 offloaded tasks to the GPU.

It becomes apparent that an offloading task follows the same periodic activation pattern on both, the CPU as well as the GPU. However, each subsequent phase of the offloading task is delayed by a minimum offset equal to the best case response time of the previous phase (cf. Fig. 3, 2nd iteration), and a maximum offset equal to the worst case response time (cf. Fig. 3, 4th iteration). In other words, the release of the suspension phase will occur no earlier then the best case response time of its pre-processing phase, and no later then the pre-processing phases worst case response time. In order to consider this behavior during our response time analysis, we need to adjust the values for offset and jitter for offloading and post-processing phases. Without loss of generality, we can describe this dependency by reformulating the notation from [7] for subsequently released sub-tasks τ_{ij} belonging to the same task τ_i as presented in Eq. 4.

$$\forall j \quad = \quad \{2, \dots, |\tau_i|\} \quad : \quad \begin{array}{l} O_{ij} = \mathcal{R}_{ij-1}^{+} \\ J_{ij} = \mathcal{R}_{ij-1}^{-} - \mathcal{R}_{ij-1}^{+} \end{array} \tag{4}$$

As both of our scheduling specific response time analysis approaches applied in the following sub-sections have a monotonic dependency of the response time on the jitter terms [7], an iterative algorithm will guarantee the jitter value to converge.

In the following subsections we will present the respective response time analysis approaches for tasks executed on the CPU resp. on the GPU.

*1) **Tasks execution on the CPU:*** Tasks executed on the CPU follow a fully preemptive fixed priority based scheduling strategy. As their total computation time needs to account times for memory access, we begin by determining the total execution time for each sub-task that is scheduled on the CPU. As presented in the previous section, we need to consider bounds for the best resp. worst cases. Since the calculation for both cases is generally the same and only differs in the values that are used for $\mathcal{C}_{ij,\rho}$ and $\mathcal{A}_{\rho}$, i.e. $\mathcal{C}_{ij,\rho}^{+}$ and $\mathcal{A}_{\rho}^{+}$ for the best case as well as $\mathcal{C}_{ij,\rho}^{-}$ and $\mathcal{A}_{\rho}^{-}$ for the worst case, we provide a general notation that can be applied for both cases.

The total execution time $\mathcal{W}_{ij}$ of a sub-task executed on the CPU consists of the raw processing time $\mathcal{C}_{ij,\rho}$ for the processing unit ρ, and the delays introduced by accessing labels and contention effects caused by other processing units (CPU, GPU) accessing shared resources such as e.g. DRAM. We formalize this behavior in Eq. 5 with $\mathcal{A}_{\rho}$ being the access time for accessing the shared memory from a processing unit ρ and λ_i being the numbers of memory accesses. For simplicity, we sum up the read and write accesses, since both have the same access time in the given challenge.

$$\mathcal{W}_{ij} = \mathcal{C}_{ij,\rho} + \lambda_{ij} \cdot \mathcal{A}_{\rho} \tag{5}$$

The number of memory accesses is trivially calculated by summing up the number of memory accesses per label l in Eq. 6, with $\mathcal{L}_{ij}$ being the set of labels accessed by sub-task τ_{ij}.

$$\lambda_{ij} = \sum_{l \in \mathcal{L}_{ij}} \left\lceil \frac{size(l)}{size(cacheline)} \right\rceil \tag{6}$$

In order to determine the worst case response time of any task τ_i, we need to identify the response time of its last sub-task $\tau_{i|\tau_i|}$ in Eq. 7.

$$\mathcal{R}_{i}^{-} = \mathcal{R}_{i|\tau_i|}^{-} \tag{7}$$

What remains now is deriving bounds on the best resp. worst case response times $\mathcal{R}_{ti}^{+}$ and $\mathcal{R}_{ti}^{-}$ for each of the sub-tasks. A good approximation for the best case response time can be obtained by summing up the total computation times of a tasks predecessors including itself as shown in Eq. 8 [7].

$$\mathcal{R}_{ij}^{+} = \sum_{k=1\dots j} \mathcal{W}_{ik}^{+} \tag{8}$$

The problem of finding the worst case response time in task sets with offsets has been shown to be NP complete by Tindell et al. [11] as it exponentially grows with the number of tasks. Although the challenge subject to this paper consists of a comparatively small example with 10 tasks, the analysis presented in this section will become part of the optimization challenge addressed in Sec. V, which again is a NP complete problem. Consequently, we find it desirable to apply an efficient approximation that will not only scale well with larger problem sizes, i.e. have a polynomial efficiency, but also provide solutions close to the exact response time analysis.

Therefore, we focus on the upper-bound approximation approach that has been developed by Tindell et al. [11] and later refined by Palencia et al. [7] for tasks with dynamic offsets.

The maximum response time $\mathcal{R}_{ij}^{-}$ for a given sub-task τ_{ij} is obtained by Eq. 9 [7] by checking every critical instant for an instance p that falls into the tasks busy period, with the function hp_i returning all higher priority sub-tasks that belong to task τ_i.

$$\mathcal{R}_{ij}^{-} = \max_{\forall c \in hp_i(\tau_{ij}) \cup \tau_{ij}} \left[\max_{p = p_{0,ijc}, \dots, p_{L,ijc}} (\mathcal{R}_{ijc}(p)) \right] \tag{9}$$

The response time for a given instance p when the critical instant coincides with the activation of task τ_{ic} is then derived by Eq. 10 [7] by subtracting the phase between both tasks and it's release time from the absolute response time, and adding it's offset.

$$\mathcal{R}_{ijc}(p) = w_{ijc}(p) - \Phi_{ijc} - (p-1)T_i + O_{ij} \tag{10}$$

Due to lack of space and since we did modify the original approach, we omit a further description and refer to the original source in [7] for a complete formulation of the remaining functions.

*2) **Tasks executed on the GPU:*** Sub-tasks being executed on GPU's (*kernels*) follow a weighted round robin scheduling policy. A context switch to the next queued kernel occurs (i) after the kernels predefined time slice elapses or (ii) the execution is finished, whichever is encountered first. Since kernels also follow a strict read-execute-write policy, their total execution time $\mathcal{W}_{ij}^{G}$ needs to account the copy engines *copy-in* $\mathcal{C}_{in}^{CE}$ resp. *copy-out* $\mathcal{C}_{out}^{CE}$ operations along with the execution engines execution time $\mathcal{C}^{EE}$ (Eq. 11).

$$\mathcal{W}_{ij}^{G} = \mathcal{C}_{in_{ij}}^{CE} + \mathcal{C}_{ij}^{EE} + \mathcal{C}_{out_{ij}}^{CE} \tag{11}$$

The delay introduced by the copy engines copy-in and copy-out operations is derived similarly to the memory access of CPU tasks. In order to create or copy back a local label, the memory has to be accessed twice. Due to the equal access times for read and write accesses, the duration of a copy engine can be determined equivalently to Eq. 5 by multiplying the number of accesses λ_{ij} by the access time $\mathcal{A}_\rho$ and the scale 2 to account the resp. write access for each read access in Eq. 12.

$$\mathcal{C}_{in_{ij}}^{CE} + \mathcal{C}_{out_{ij}}^{CE} = 2 \cdot \lambda_{ij} \cdot \mathcal{A}_\rho \tag{12}$$

Similar to tasks executed on the CPU, this approach can be used in determining both, worst as well as best case execution times by considering the resp. execution and access times C_{ij}^{+} and A_ρ^{+} and vice versa.

For the remainder of this section, we adapt the approach in [8] for determining the best and worst case response times in weighted round robin scheduled tasks by checking different time windows for a given busy period.

With a single stream, a sub-tasks τ_{ij} worst case response time $\mathcal{R}_{ij}^{G-}$ is derived (slightly reformulated compared to [8]) as presented in Eq. 13 by determining the maximum of all response times $\mathcal{R}_{ij}^{G-}(q)$ for the q-th time window of the busy period, with $n_{ij}^{+}(\Delta t)$ being an arrival function that returns the maximum number of activations for the given interval Δt [9].

$$\mathcal{R}_{ij}^{G-} = \max_{1 \leq q \leq n_{ij}^{+}(w_{ij}(q))} \left(\mathcal{R}_{ij}^{G-}(q)\right) \tag{13}$$

The execution time for the first q number of activations including all interferences $\mathcal{I}$ caused other tasks that are executed on the GPU is given by $w_{ij}(q)$ in Eq. 14.

$$w_{ij}(q) = q \cdot \mathcal{W}_{ij}^{G-} + \mathcal{I}_{ij}(q) \tag{14}$$

The response time for a given time window $\mathcal{R}_{ij}^{G-}(q)$ is then derived in Eq. 15 by subtracting the earliest absolute release time $\delta_{ij}^{-}(q)$ [9] of the q-th time window from the total execution time.

$$\mathcal{R}_{ij}^{G-}(q) = w_{ij}(q) - \delta_{ij}^{-}(q) \tag{15}$$

For determining the individual interference $\mathcal{I}_{ij}(q)$ for the q-th time window as well as the modifications required for determining the best case response time R_{ij}^{G+}, we refer to the original work in [8].

*3) **Memory Access Latency:*** As stated in the challenges description [3], the time for reading or writing to the memory $\mathcal{A}_\rho$ from processing unit ρ can be calculated by Eq. 16 for CPUs resp. Eq. 17 for GPUs, with CC_ρ being the number of cycles for accessing the memory, f_ρ the processing units frequency, and ζ the numbers of processing units concurrently accessing the memory. Moreover, the constant κ_ρ annotates the increase in latency for each interfering CPU, whereas γ_ρ represents the increase in latency if the GPU's copy engine is performing operations.

$$\mathcal{A}_\rho = \frac{CC_\rho}{f_\rho} + \kappa_\rho \cdot \zeta + \gamma_\rho \tag{16}$$

$$\mathcal{A}_\rho = \frac{CC_\rho}{f_\rho} + 0.5ns \cdot \zeta \tag{17}$$

In the following, we illustrate the worst case in which all $\zeta = 5$ neighboring cores as well as the GPU's copy engine cause contentions. Given the provided values for κ_ρ and γ_ρ in [3], the previous equations can be simplified into a worst case access time A_ρ^{-} in Eq. 18.

$$\mathcal{A}_\rho^{-} = \begin{cases} 220\,ns & \text{for CPUs (A57)} \\ 38\,ns & \text{for CPUs (Denver)} \\ 6\,ns & \text{for GPUs} \end{cases} \tag{18}$$

For the best case, we assume that none of the neighboring cores cause contention, which leads to the best case access time A_ρ^{+} in Eq. 19.

$$\mathcal{A}_\rho^{+} = \begin{cases} 20\,ns & \text{for CPUs (A57)} \\ 8\,ns & \text{for CPUs (Denver)} \\ 3\,ns & \text{for GPUs} \end{cases} \tag{19}$$

*4) **Results for analysis model:*** The results of our analysis are presented in Tab. II using the notation from Sec. III. Tasks denoted with an asterisk (*) represent offloading tasks that are executed on the CPU iff the offloaded task is being executed on the GPU. It becomes apparent that task *Planner* cannot be scheduled as it's total worst case execution time $\mathcal{W}_i = 12.4 + 0.8 = 13.2ms$ exceeds its period $T = 12$. Accordingly, we decide to reduce its number of ticks by 10% in order to make it schedulable on a Denver core while maintaining a tight bound.

As the remaining model is not schedulable due to it's allocations [3], we will apply the remainder of our analysis, i.e. the end-to-end analysis and the response time analysis, on the feasible solution that is generated in the design space exploration challenge.

TABLE II
WORST CASE EXECUTION-, AND COMMUNICATION TIMES FOR EACH TASK AND PROCESSING UNIT

		Denver				A57				Pascal			
Name	P	C^-	C^+	$\lambda \cdot \mathcal{A}^-$	$\lambda \cdot \mathcal{A}^+$	C^-	C^+	$\lambda \cdot \mathcal{A}^-$	$\lambda \cdot \mathcal{A}^+$	C^-	C^+	$\lambda\mathcal{A}^-$	$\lambda\mathcal{A}^+$
DASM	5	1.3	1.0	0.0	0.0	1.9	1.3	0.0	0.0	-	-	-	-
CANbus_polling	10	0.6	0.4	0.0	0.0	0.6	0.4	0.0	0.0	-	-	-	-
Planner	12	12.4	9.5	0.8	0.2	13.2	9.6	4.4	0.4	-	-	-	-
EKF	15	4.4	4.1	0.0	0.0	4.8	4.0	0.0	0.0	-	-	-	-
Lidar_Grabber	33	10.9	9.8	2.1	0.4	13.7	10.2	12.0	1.1	-	-	-	-
SFM	33	27.8	22.2	2.4	0.5	29.5	24.1	13.9	1.3	7.9	7.0	0.4	0.2
SFM*	33	6.7	5.4	3.6	0.8	7.9	6.3	20.8	1.9	-	-	-	-
Lane_detection	66	42.2	38.4	2.4	0.5	51.0	47.8	13.8	1.3	27.3	24.5	0.4	0.2
Lane_detection*	66	7.6	6.1	2.4	0.5	8.3	6.8	13.8	1.3	-	-	-	-
OS_Overhead	100	50.0	50.0	0.0	0.0	50.0	50.0	0.0	0.0	-	-	-	-
Detection	200	-	-	-	-	-	-	-	-	116.0	108.0	0.5	0.3
Detection*	200	4.1	3.0	3.3	0.7	4.7	4.0	19.0	1.8	-	-	-	-
Localization	400	294.8	276.7	1.8	0.4	387.4	366.5	10.3	0.9	124.0	117.0	0.3	0.1
Localization*	400	14.5	6.1	1.8	0.4	17.6	7.3	10.3	0.9	-	-	-	-

V. PROPOSED SOLUTION FOR OPTIMIZATION CHALLENGE

Our scope in the optimization challenge lies in minimizing the applications end-to-end latency, i.e. the term $L^{E2E} = \max_{\sigma \in \mathcal{S}}(L^{TC}(\sigma))$ as specified in the previous section. With regard to the communication paradigms, the only approach for reducing the end-to-end latency in LET communication would be in modifying the task's period P_i which is out of the scope of this paper, therefor we will focus on finding a feasible allocation for this case that ensures schedulability.

For the implicit case, the applications latency can be optimized by altering a task's worst case response times. Consequently we will use the tasks priority, the allocation from tasks to processing units, and time slices for tasks offloaded to the GPU as a degree of freedom in our approach.

For priority assignment, we apply Audsleys approach [1] that will determine priorities leading to a feasible schedule, and combine it with the recurrent-relation for assigning offsets and jitters for tasks performing offloading. The remainder of our optimization is performed by genetic algorithm based approach that aims at minimizing the applications end-to-end latency. It is implemented in Java using the open source library Jenetics [12] and extends the native DSE capability [6] of App4MC. The genetic algorithm has been executed on an Intel Core i5-3570K quad-core CPU operating 3.4 GHz with an initial population of 500 randomly initialized individuals and a termination criteria of 1000 iterations after a steady, i.e. non-improving, fitness value.

The applications end-to-end latency for each of the previously described task-chains is illustrated in Tab. III, with the deployment leading to these results in Tab. IV. The deployment was found after approx. 5 minutes, with the majority of cpu time (287 out of 291 seconds) taken by the fitness calculation as described in Sec. IV.

We can observe that implicit communication slightly ($3\% - 28\%$, avg. 10%) outperforms LET communication, which seems convenient considering that the response times of those tasks that form a task chain are very close to the resp. task's period. As expected, one of the Denver cores is forced to exclusive execute the task Planner due to a lack of alternatives.

TABLE III
WORST-CASE END-TO-END LATENCY FOR LET COMMUNICATION (LEFT) AND IMPLICIT COMMUNICATION (RIGHT)

Task Chain	LET end-to-end	Implicit end-to-end
σ_1	886	859.9
σ_2	865	836.9
σ_3	67	59.9
σ_4	100	71.9
σ_5	230	221.9

TABLE IV
WORST CASE EXECUTION-, COMMUNICATION, AND RESPONSE TIMES FOR TASKS

Name	P	π	C^-	$\lambda \cdot \mathcal{A}^-$	R^-	ϕ
Core 0 (Denver)						
Planner	12	9	11.2	0.8	12.0	–
Core 1 (Denver)						
SFM*	33	6	6.7	3.6	31.5	–
Lane_detection	66	2	42.2	1.2	53.6	–
Core 2 (A57)						
CANbus_polling	10	5	0.6	0.0	0.6	–
EKF	15	1	4.8	0	5.4	–
Core 3 (A57)						
Localization	400	4	387.4	5.2	392.6	–
Core 4 (A57)						
Lidar_Grabber	33	8	13.7	12.0	25.7	–
Detection*	200	7	4.7	1.8	198.0	–
Core 5 (A57)						
OS_Overhead	100	0	50	0.0	79.9	–
DASM	5	3	1.9	0.0	1.9	–
GP10B (GPU)						
Detection	200	–	116.0	0.5	166.2	375
SFM	33	–	7.9	0.0	19.9	11.6

VI. CONCLUSION AND OUTLOOK

This work presents an approach for analyzing the end-to-end latencies in applications for heterogeneous embedded systems.

It provides an detailed description on the challenges when (i) analyzing the application's end-to-end response time and (ii) minimizing it by optimizing the application's deployment. Since the initial system was not schedulable under our worst-case assumption (i.e. the total worst case execution time of task Planner exceeded its period), we reduced the number of ticks by 10% for the task planner only. We have presented our results for both challenges, consisting of the applications worst case response time (σ_1) for the implicit and LET communication paradigms as well as the worst case execution times, worst case communication overheads including contention, worst case response times, and the optimized and feasible deployment of the application.

Due to the complexity of the WATERS2019 challenge and the time required to fully comprehend the essential characteristics we had to narrow down the scope of our contribution, leaving room for future work we would like to address in the future.

While our current solution only covers a single streaming multiprocessor, it would be desirable to further exploit the hardware's capabilities by utilizing both SMs while considering any contention effects this would introduce. Moreover, we plan to consider other scheduling approaches that allow e.g. migrating tasks at run-time. Finally, we would like to back-up our findings with benchmarks of the prototypical application by executing these on a NVidia TX2 in order to increase the accuracy of our approaches.

References

[1] Neil C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. 2007.

[2] AUTOSAR. Specification of RTE 4.2.2, 2015. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SWS_RTE.pdf.

[3] Arne Hamann, Dakshina Dasari, , and Falk Wurst. *WATERS Industrial Challenge 2019*. 2019. URL: https://www.ecrts.org/waters/.

[4] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication Centric Design in Complex Automotive Embedded Systems. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:20, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7162, doi:10.4230/LIPIcs.ECRTS.2017.10.

[5] Tomasz Kloda, Antoine Bertout, and Yves Sorel. Latency analysis for data chains of real-time periodic tasks. In *23rd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2018, Torino, Italy, September 4-7, 2018*, pages 360–367. IEEE, 2018. URL: https://doi.org/10.1109/ETFA.2018.8502498, doi:10.1109/ETFA.2018.8502498.

[6] Lukas Krawczyk, Carsten Wolff, and Daniel Fruhner. Automated distribution of software to multi-core hardware in model based embedded systems development. In Giedre Dregvaite and Robertas Damasevicius, editors, *Information and Software Technologies - 21st International Conference, ICIST 2015, Druskininkai, Lithuania, October 15-16, 2015, Proceedings*, volume 538 of *Communications in Computer and Information Science*, pages 320–329. Springer, 2015. URL: https://doi.org/10.1007/978-3-319-24770-0_28, doi:10.1007/978-3-319-24770-0_28.

[7] J.C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. 2002. doi:10.1109/real.1998.739728.

[8] Razvan Racu, Li Li, Rafik Henia, Arne Hamann, and Rolf Ernst. Improved response time analysis of tasks scheduled under preemptive round-robin. In Soonhoi Ha, Kiyoung Choi, Nikil D. Dutt, and Jürgen Teich, editors, *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2007, Salzburg, Austria, September 30 - October 3, 2007*, pages 179–184. ACM, 2007. URL: https://doi.org/10.1145/1289816.1289861, doi:10.1145/1289816.1289861.

[9] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Dec 2004. URL: https://publikationsserver.tu-braunschweig.de/receive/dbbs_mods_00001765.

[10] Selma Saidi, Sebastian Steinhorst, Arne Hamann, Dirk Ziegenbein, and Marko Wolf. Special Session: Future Automotive Systems Design: Research Challenges and Opportunities. In *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–7. IEEE, sep 2018. URL: https://ieeexplore.ieee.org/document/8525873/, doi:10.1109/CODESISSS.2018.8525873.

[11] K.W. Tindell. Adding time-offsets to schedulability analysis. *Department of Computer Science, University of York,*, 1994.

[12] Franz Wilhelmsttter. Jenetics: Java genetic algorithm library, 2019. URL: http://jenetics.io/.