CPU-GPU Response Time and Mapping Analysis for High-Performance Automotive Systems

Robert Höttger, Junhyung Ki, The Bao Bui, Burkhard Igel
IDiAL Institute
Dortmund University of Applied Sciences and Arts
Dortmund, Germany
{robert.hoettger, igel}@fh-dortmund.de,
{junhyung.ki001, the.bui003}@stud.fh-dortmund.de

Olaf Spinczyk
Computer Science Institute
Osnabrück University
Osnabrück, Germany
olaf.spinczyk@uos.de

***Abstract*—In accordance to the interest in autonomous driving, automotive software requires increasing computing power whilst formal verification methods form crucial requirements in order to cope with safety, reliability, real-time, or fault-tolerance demands. Image processing is a mandatory part for this trend which makes the use of GPUs reasonable. This paper outlines methods to address formal verification challenges in modern automotive environments and presents results along with an industrial model provided by the WATERS workshop community. Most of the challenges addressed in this paper are part of the WATERS industrial challenge 2019 [5].**

***Index Terms*—AMALTHEA, APP4MC, Mapping, Automotive, RTA, GPU**

I. Introduction

Recent development activities in the automotive domain have risen challenges when applying response time analysis (RTA) as well as memory contention and access latency estimation to AUTOSAR compliant models. Mapping tasks to processing units in order to optimize latencies of task chains and response times under the consideration of memory to GPU offloading costs and highly heterogeneous hardware architectures with different memory types, processing speeds, peripherals, accelerators, and more, as well as sophisticated memory contention models increases complexity and requires appropriate changes to conventional formal RTA methods [5]. This paper uses conventional RTA for fully-preemptive tasks under rate monotonic scheduling running on CPUs using the windowing technique [11] and combines it with weighted round robin (WRR) RTA for GPU tasks [13].

Additionally, we define a memory contention model for GPU copy engines as well as differences of asynchronous and synchronous GPU offloading mechanisms.

While memory accesses are already accounted within ticks of GPU tasks, CPU task response times must account memory contention caused by tasks running on different processing units that use different memory controller clients and/or the GPU's copy engine, and memory access latencies in addition to ticks and preemption.

Finally, we present results of task to CPU and GPU mapping exploration conducted with the help of a genetic algorithm. We repeat the algorithm for different optimization goals and measure various metrics such as response time sum, standard deviation of processing unit utilizations, as well as latencies of cumulated memory accesses, task chains, the copy engine operations, and contention over worst-, best-, or average execution times and synchronous and asynchronous offloading.

The remainder of this paper is structured as follows. The next Section II introduces related work, the context this paper's work is referring to, as well basics of the used system model. Afterwards, Sections III and IV form the main **contributions** of this paper and formulate the specific challenges and their solution approaches to **response time analysis and task mapping across CPUs and GPUs** in the respective automotive domain. The solutions **account data access costs, memory contention, Copy Engine (CE) operations, synchronous or asynchronous offloading, rate monotonic CPU and WRR GPU scheduling**, whereas the task mapping either optimizes (a) the sum across all task response times, (b) task chain latencies, of (c) load balancing. Finally, Section V presents measurements and results whereas Section VI concludes this paper.

II. Related Work, Assumptions, & System Model

Current research shows that compute and memory bandwidth isolation is an effective approach to reduce shared cache conflicts, bus and shared cache contention, as well as buffer conflicts or request reordering in the memory controller [9]. The WATERS community has been working on solving various automotive challenges since 2015 [10] such as worst-case end-to-end latencies along complex cause-effect chains [6], communication paradigms [4], WCET / WCRT for advanced shared memory architectures [14], optimized application mapping, and sophisticated models for multi-core execution platforms.

The WATERS2019 challenge [5] forms the basis of this paper via the following assumptions:

- Fixed priority (mixed) preemptive scheduling in form of rate monotonic scheduling ($D_i = T_i$)
- CPU contention given as:

$$\gamma_{i,k} = bl_{k,i} + (K_k \cdot \#C_i) + sGPU \cdot bGPU \quad (1)$$

with $\#C_i$ denoting the number of cores that run at least one task which accesses at least one label accessed by τ_i.

Baseline $bl_{i,k}$ is derived from a processing unit's access latency to memory as well as labels accessed by τ_i. K and $sGPU$ are constants derived from the memory contention model [2] in conjunction with information given in the forum[1].

- GPU contention is given as $\gamma_{i,GPU} = lb_{GPU} + 0.5 \cdot \#C$
- Copy operations of the CE are handled by the GPU
- The execution engine's memory accesses are already covered in GPU ticks
- Data is always transferred in form of an integer multiple of a complete cache line (i.e. 64 Bytes)

The challenge model is given as an AMALTHEA model[2] that can be accessed by the APP4MC[3] platform.

Table I collects all indexes and notations used throughout this paper which is based on the Burns Standard Notation [3].

TABLE I
INDEXES AND NOTATIONS

Entity	index	Entity	index
Task	i	Runnable	p
Processing Unit	m	Memory	l
Label	n	Global memory	g
Lower priority tasks	o	Higher priority tasks	h
Description	**Symbol**	**Description**	**Symbol**
Deadline	D_i	Period	T_i
WC execution time	C_i^+	WC exec. time on pu_m	$C_{i,m}^+$
Runnable exec. time	c_p	Task	τ_i
WC Response time	R_i^+	Busy period	W_i
Procesing unit	pu_m	Priority	P_i
Utilization	$U_{i,m}$	Runnable	r_p
Frequency in Hz	f_m	Latency	L

Given those indexes, we define the following different latencies:

- $L_{a,i}$ = A task's access latency derived from all its label accesses to memory
- $L_{c,i}$ = A task's contention latency derived from the challenge denoted in Eq. 1.
- A task's locking latency that is subdivided into:
  - The global latency defined by spin locks for resources shared across tasks running on different processing units
  - The local locking latency derived from the priority ceiling protocol (PCP)

  The locking should be considered for semaphores or other lock types that ensure deterministic system behavior as well as cooperative tasks that can only be preempted at runnable bounds.

Several suggestions to improve real time behavior and determinism for AUTOSAR applications running on a heterogeneous multi core system have been proposed [12] [7] but not yet included in the AUTOSAR specification. Considering locking latencies is planned for a later extension of this paper's work.

[1]WATERS forum thread https://bit.ly/2IlLXTe, accessed 05.2019
[2]AMALTHEA 0.9.3 documentation http://eclip.se/fy, accessed 06.2019
[3]APP4MC website http://eclip.se/eU, accessed 06.2019

III. Challenge I: RTA for CPU-GPU

A. CPU Response Time Analysis

1) Data Access Costs:
Before a tasks starts executing on a CPU, labels need to be read from memory. After a task finishes execution, its results (labels) must be written into memory. Eq. 2 describes the memory access latencies that is added to each CPU task's response time.

$$L_{a,i}^+ = \sum_{x \in rl_i} \left(\left\lceil \frac{ls_x}{64} \right\rceil \right) \cdot \frac{rl_{m,l}}{f_m} + \sum_{y \in wl_i} \left(\left\lceil \frac{ls_y}{64} \right\rceil \right) \cdot \frac{wl_{m,l}}{f_m} \tag{2}$$

The constant 64 is used here as the baseline derived from the challenge [5]. Here, ls denotes the label size and rl and wl define given read label and write label latencies specified in the given AMALTHEA model. Recent publications such as [8] have shown that memory mapping significantly influences task response times.

2) Memory Contention:
Memory contention latencies are added to task response times using Eq. 1.

3) CPU Response Time Analysis:
Based on [14], we consider the worst case response time (R_i^+) for rate monotonic scheduling within a level-i busy period window using recurrence relation as shown in Eq. 4. However, the previously outlined latencies must be added to task execution times in order to get more accurate analysis. While label access latencies always occur (also for BCET), worst case execution times take all described latencies into account as shown in Eq. 3.

$$C_i^+ = \sum_p c_p + L_{a,i} + L_{c,i} + L_{l,i} \text{ with } r_p \in \tau_i \tag{3}$$

$$\begin{aligned} W_i &= \sum_{he \in hep(i)} \left\lceil \frac{W_i}{T_{he}} \right\rceil \cdot C_{he} \text{ with } he \in (h \wedge i) \\ K_i &= \left\lceil \frac{W_i}{T_i} \right\rceil \\ f_i^{k,+} &= \sum_{h \in hp(i)} \left\lceil \frac{f_i^{k-1,+}}{T_h} \right\rceil \cdot C_h^+ + k \cdot C_i^+ \\ R_i^{+,\text{CPU}} &= \max_{k \in [1,K_i]} \left(f_i^k - (k-1)T_i \right) \end{aligned} \tag{4}$$

W_i is the busy period length (window) that has to be considered for task τ_i. K_i is the amount of task τ_i instances within the busy period that needs to be checked. f_i^k is the finish time of the k-th instance of τ_i and R_i is the worst case response time of τ_i which is the maximum among all k-th finish times minus the k-th task τ_i's release which is derived from the task's period. Equations 4 are entirely derived from [14], depend on the task to pu mapping, and consider only the set of tasks mapped to the same processor pu. This implementation allows arbitrary deadlines over rate monotonic derived deadlines.

4) Asynchronous and Synchronous Offloading:
The two offloading cases are expressed in Figure 1. Since

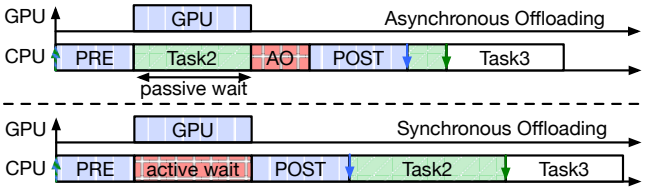

Fig. 1. Synchronous vs asynchronous GPU task offloading (without copy operations) in a Gantt chart

passive waiting allows other tasks to execute (cf. Task2 in Figure 1 asynchronous offloading), the overall throughput is higher for the asynchronous offloading. However, a penalty has to be added for the asynchronous offloading to represent the latency between the end of GPU kernel and the start of the post processing phase, which is denoted as `AO` for asynchronous offloading costs in Figure 1. Those additional costs `AO` require less processing resources compared with the relatively longer active waiting period during synchronous offloading. The synchronous offloading can be implemented using the conventional RTA from [11]. Consequently, for the synchronous case, the triggering task τ_i's execution time is:

$$Cs_i^+ = C_i^+ + CE_i^+ + R_{j,GPU}^+ \tag{5}$$

With CE_i^+ denoted in Eq. 7 for considering copy engine (CE) operations. This means, we take the normal execution time for a task and add the CE time for the triggered task and its response time at the GPU.

In order to calculate response times that consider the passive waiting for the asynchronous offloading situation, we split the triggering task into two parts, i.e., `pre` and `post` processing tasks. While the former simply receives the execution time of everything until the trigger event, the latter receives all execution time after the trigger event including `pre` processing as well as the `AO` penalty. Additionally, the latter task obtains an offset value which equals the `pre` task's length plus the triggered GPU task's response time. Consequently, we use another RTA to consider offsets, based on [15] for the asynchronous offloading approach, in which passive waiting can be utilized by other tasks. The offset consideration makes use of the "imposed interference" method since the critical instant derivation used for the synchronous offloading is not viable when having offsets for the asynchronous case. Therefore, task sets with the same periodic activation but different offsets are combined in transactions. Each transaction's (Γ_i) effectively imposed interference ($W_{db}(R_i, t)$) during an iteratively increasing time interval (t) is computed. The iteration ends via fix-point lookup for the response time calculation of the task under consideration, i.e. $R_i^{+,\text{offs}} = R_i^{+,\text{offs}(n)}$ with $R_i^{+,\text{offs}(n)} = R_i^{+,\text{offs}(n-1)}$.

$$\begin{aligned}
W_{db}(\tau_i, t) &= \sum_{j \in hp_d(\tau_i)} \left(\left(\left\lfloor \frac{t^*}{T_d} \right\rfloor + 1 \right) \cdot C_{dj} - x_{djb}(t) \right) \\
t^* &= t - phase(\tau_{dj}, \tau_{db}) \\
phase(\tau_{dj}, \tau_{db}) &= (T_d + (O_{dj} - O_{db})) \ \% \ T_d \\
x_{djb}(t) &= \begin{cases} 0 & \text{for } t^* < 0 \\ \max(0, C_{dj} - (t^* \% T_d)) & \text{otherwise} \end{cases} \\
W_d(\tau_i, t) &= \max_{b \in hp_d \tau_i} (W_{db}(\tau_{ua}, t)) \\
R_i^0 &= C_i^+ \\
R_i^{+,\text{offs};(n+1)} &= C_i^+ + \sum_{\Gamma_d \in \Gamma} \left(W_d \left(\tau_i, R_i^{+,\text{offs};(n)} \right) \right)
\end{aligned} \tag{6}$$

All equations 6 are derived from [15] whereas d is the transaction index. The part of task τ_{dj} that cannot be executed during interval t is denoted as x_{dj}.

B. GPU Response Time Analysis

Before a GPU task starts executing, the copy engine needs to copy all accessed labels into the dedicated GPU region.

1) Copy Engine:
The copy engine CE reads all labels accessed by a task from different memories, writes them into a dedicated GPU memory location of the global memory and after the GPU execution finished, all labels are written back into their original location. This copy engine access latency is described in Eq. 7.

$$CE_{a,i}^+ = \sum_{l_{n,i}} \left(\frac{ls_n \cdot (rl_{m,l} + wl_{m,l})}{f_m} \right) + \frac{lss_i \cdot (wl_{m,g} + rl_{m,g})}{f_m} \tag{7}$$

with $l_{n,i}$ being all labels l_n accessed by τ_i, ls_n denoting the label size in baseline, i.e. $ls_n = \left\lceil \frac{\#Bytes_n}{64} \right\rceil$, l being the index of the memory l_n is mapped to, f_m being the frequency in Hz of the GPU τ_i is mapped to, $rl_{m,l}$ is the read latency between GPU_m and memory mem_l, and $rl_{m,g}$ the read latency between GPU_m and global memory mem_g. The label size sum of a task τ_i : $lss_i = \sum_{l_{n,i}} ls_n$. Both read and write latencies to the label's original place (mem_l) are multiplied with the label sizes, since the copy engine reads during the copy-in phase and writes during the copy out phase. Since the labels are copied into the global memory region for the GPU, write and read latencies must be further multiplied with the label size sum lss_i for the copy in and copy out operations respectively. The resulting data flow is $CE_{read\_l} \Rightarrow CE_{write\_g} \Rightarrow GPU_{execution} \Rightarrow CE_{read\_g} \Rightarrow CE_{write\_l}$. For instance, the CE_a time for a task τ_1 accessing a single label of 128 Bytes from an A57 core which has 5 cycles read and 6 cycles write latency to the memory the label is mapped to, and 7 read and 8 write cycles latency to global memory, is $CE_{a,1} = \frac{\left(\frac{128}{64}\right) \cdot (5+6)}{2 \cdot 10^9} + \frac{2 \cdot (7+8)}{2 \cdot 10^9} = 26ns$. In addition to the actual copy operation time, contention $CE_{c,i}^+$ and queuing

delay $CE^+_{q,i}$ needs to be considered. The CE contention time is derived from the challenge description, i.e.:

$$CE^+_{c,i} = \sum_{i=0}^{\#Bytes/64} (bl + (K \cdot \#C)) \tag{8}$$

The CE queuing delay is derived from any higher priority copy operations among tasks mapped to the GPU and shown in Eq. 9. We assume FIFO-ordered CE queuing.

$$CE^+_{q,i} = \sum_{h \in hp(\tau_i) \setminus \tau_{max}} CE^+_{a,h} \tag{9}$$

Given a predefined label mapping as well as access latencies from each processing unit to each memory and the frequency of each core, we finally derive the total copy engine time that considers blocking and contention via Eq. 10.

$$CE^+_i = 2 \cdot (CE^+_{q,i} + CE^+_{c,i}) + CE^+_{a,i} \tag{10}$$

During the implementation of the copy engine lantecy calculation, situations were identified, to which labels were written back to their original place, but not changed during the GPU execution. Consequently, a small adjustment of the CE operation was implemented, which only accounts written (changed) labels to be chosen for being written back to their original location. This adjustment reduced the CE operation latency to a small extent as outlined in Section VI.

2) GPU RTA using Weighted Round Robin Scheduling: After the copy engine time has been calculated, we can analyze the response times of GPU tasks scheduled by a WRR scheduler. A task set is schedulable if Eq. 11 is true.

$$\sum_i \left(\frac{C^+_i \cdot T_{max}}{T_i} \right) \leq T_{max} \tag{11}$$

We implemented the response time analysis described in [13] apart from the specific burst stimulus consideration, which is not part of the model in scope. The RTA under round robin uses the windowing-technique proposed by Lehoczky in [11] to check for the worst case response time of tasks with arbitrary deadlines within the critical instant, i.e., the situation when all tasks are released at the same time. The implemented algorithm considers interference of others tasks ($\mathcal{I}_{k,j}$) within a round robin turn (k), task interference of previous round robin turns (pad), requested execution times until each time slice window, as well as periodic tasks with different execution times and time slices in order to derive accurate round robin timing behavior without much pessimism.

$$\begin{aligned}
R^{+,\text{GPU}}_i(q) &= q \cdot C^+_i + \mathcal{I}(q) - \delta^-_i(q) \\
\mathcal{I}(q) &= \sum_{k=1}^{K(q)} \mathcal{I}_k, \text{ where } K(q) = \left\lceil \frac{q \cdot C^+_i}{\theta_i} \right\rceil \\
\mathcal{I}_k &= \sum_{j=1}^{n-1} \mathcal{I}_{k,j} \\
\mathcal{I}_{k,j} &= \sum_{v=1}^{V} pad_{k,j}(v), V \text{ such that } pad_{k,j}(v) \neq 0 \\
pad_{k,j}(v) &= \min \left(l_{k,j}(v), C^{\max}_j \cdot \eta^+_j \left(t_{k,j} + \sum_{s=1}^{v-1} pad_{k,j}(s) \right) \right. \\
&\quad \left. - \sum_{p=1}^{k-1} \mathcal{I}_{p,j} - \sum_{s=1}^{v-1} pad_{k,j}(s) \right) \\
l_{k,j}(v) &= \theta_j - \sum_{s=1}^{v-1} pad_{k,j}(s) \\
t_{k,j} &= \sum_{p=1}^{k-1} \mathcal{I}_p + (k-1)\theta_i + \sum_{u=1}^{j-1} \mathcal{I}_{k,u} \\
pad_{k,j}(1) &= \begin{cases} \theta_i & \text{if } E_{k,j} \geq \theta_j \\ E_{k,j} & \text{if } E_{k,j} < \theta_j \end{cases} \\
E_{k,j} &= C^+_j \cdot \eta^+_j(t_{k,j}) - \sum_{p=1}^{k-1} \mathcal{I}_{p,j}
\end{aligned} \tag{12}$$

Here, q is the activation index, η^+_j is the upper-bound arrival function of task τ_i, δ^-_i is the minimal distance of τ_i events (set to T_i in this paper), θ_j is the time slot of τ_j, $K(q)$ is the total number of RR-turns required by q activation of τ_i to complete, $E_{k,j}$ is the remaining execution demand of task τ_j at the beginning of θ_j in the k-th RR-turn, and $l_{k,j}$ is the unused time in time slot θ_j at the beginning of $pad_{k,j}(v)$ as stated in [13].

C. Task Chain Latencies

We re-use an existing outline of task chain latency calculation derived from [1] that is shown in Eq. 13 with $\delta(TC)$ denoting the task chain length in time.

$$\begin{aligned}
TC &= \{\delta_0, ...\} \\
\delta_e &= R^+_{j=\delta_0} + \sum_{j=\delta_1}^{j<|TC|} (T_j + R^+_j)
\end{aligned} \tag{13}$$

Since no task chains are given in the existing model, we added the following two task chains manually:

1) TC1: {EKF → Localization → Planner}
2) TC2: {DASM → Pre_SFM_Post → Pre_Local._Post}

No differences across synchronous and asynchronous latency measurements were found for TC1 since no offloading task is included in that chain. However, TC1's latency measurement is still used for the task chain latency sum optimization outlined

in the next Section IV denoted as TCSO, to which results are also presented in Figure 4.

IV. Challenge II: Task Mapping

Before the actual task mapping is performed, the authors noticed, that the task `Planner` has a periodic activation of `12ms` whereas it's execution time is `>12ms` for any CPU. Consequently, we changed its period to 15ms to be schedulable.

Task mapping is encoded via a genetic algorithm using jenetics library [17]. In order to restrict the solution space for tasks available on either CPU cores only, GPU cores only, or both, instead of decoding a single chromosome with multiple integer genes, each task mapping is encoded within a dedicated chromosome consisting of a single integer gene. Consequently, genes can have different integer domains.

We assess results by summing up the following values along with the GA's fitness function:

I) **RTSO** = Response Time Sum Optimized
   a) Worst case response time sum over the CPUs and GPUs, i.e., $R^+_{tot} = \sum_i R^+_i$ that involves 4, 12, and 6)
   b) The total CPU memory access latency (cf. Eq. 2)
   c) The total CE latency (cf. Eq. 10)
   d) The total task contention ($TTC = \sum_i \gamma_i$; cf. Eq. 1)

II) **TCSO** = Task Chain latency sum Optimized
   The task chain sum of all task chain latencies via $TCS = \sum_e \delta_e$ that involves Eq. 13

III) **LBO** = Load Balancing Optimized
   The standard deviation along utilization values (given in Eq. 14) in %.

$$U_{m,\%} = \sum_i \left(\frac{C^+_{i,m} \cdot 100}{T_i} \right) \text{ with } \tau_i \text{ mapped to } pu_m \quad (14)$$

It is important to note here that a combination of any of the above calculations can easily be implemented. Additional analyses as well as further model entity analyses can be integrated within the genetic algorithm's fitness function.

V. Results

The following subsections outline results obtained by applying outlined challenge solutions described in Sections III and IV to the given WATERS model.

A. Synchronous vs Asynchronous Offloading

A CPU task, which offloads a GPU task and actively waits for its finalization (i.e. synchronous offloading), obtains significantly more execution time and response time correspondingly. Since the challenge model features already highly utilized processing units, setting the synchronous offloading to true is infeasible with the given model since no mapping could be found that is free from exceeding a processing unit's capacity across all processing units.

However, when considering best case execution times and increasing the period of the task '`PRE_SFM_gpu_POST`' from 33ms to 66ms, valid results were found (cf. '`syncBCET`' with diagonal lines in Figure 2). Having two tasks mapped to the GPU in this scenario, their triggering task's execution times increased to 911% and 1999% compared to their asynchronous execution times. This change results in an increase to 158% of the total response time sum fitness value.

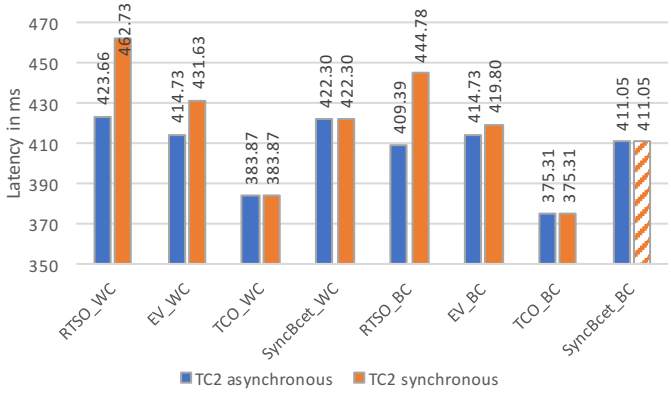

Fig. 2. Task chain latencies for different mappings as well as worst and best case execution times

B. Task Chain Latencies

Task chain latency results for `TC2` are shown in Figure 2. The `RTSO` mapping features the highest task chain latencies among the different mappings whereas the `TCO` mapping solution provides the lowest task chain latencies. Asynchronous and synchronous measurements are the same for the `TCO` mapping, since the respective offloaded tasks (*SFM* and *Localization*) are mapped to a CPU which, according to the challenge [5], results in 0 execution time for the `Pre` and `Post` processing runnables within the triggering task. The other mappings map at least one of those two tasks to the GPU which results in different latency measurements among the synchronous and asynchronous offloading approach.

Synchronous task chain latency measurements of Figure 2 violate at least one deadline except for the '`syncBCET`' solution due to the significant increase of wasted CPU cycles during acrivae waiting.

While task chain optimized mappings clearly feature the lowest task chain latencies, synchronous execution results in a task chain latency increase of up to 9.2%.

C. Various Metric Results along Different Mappings

Final utilization results are presented in Figure 3. The utilization already shows infeasibility of the given mapping model due the GPU being utilized by $> 141\%$. Even the `Denver0` processing unit will hardly meet the deadlines since the instructions only already fill $> 98\%$ of the processing unit's capacity. Utilization is computed via Eq. 14. In addition to Figure 3, Figure 4 presents measurements of different metrics along with the three optimized mappings as well as an early valid mapping. Due to infeasibility, some of those metrics could not be calculated for the given mapping model such that the given mapping is ommitted in Figure 4. Interestingly, the load balancing approach does not provide the best cumulated response times although it provides clearly the lowest

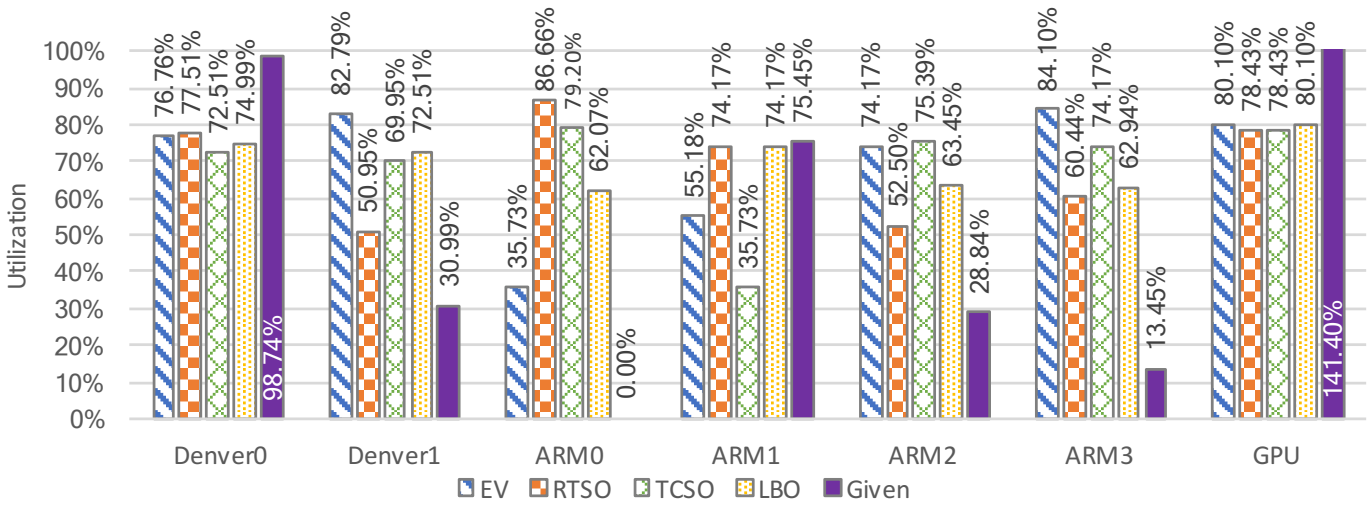

Fig. 3. Processing unit utilizations for an (a) EV = early valid mapping, (b) RTSO = response time sum optimized, (c) LBO = load balancing optimized, (d) TCO = task chain optimized mapping results, and (e) the given mapping

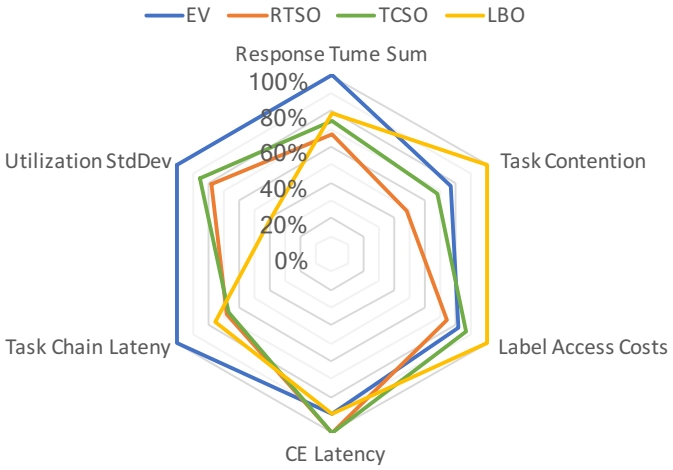

Fig. 4. Different metric measurements for calculated mapping results

utilization standard deviation with ∼40%. The configuration for the measurements of Figures 3 and 4 is considering WCET, asynchronous offloading, only written labels for the CE back to host process, and 1ms $\cdot P_i$ for the GPU time slice derivation.

D. Time Slice Derivation for GPU WRR Scheduling

During investigating time slice lengths and weights for the round robin scheduling on the GPU, ticks of the `Detection` task were reduced to $C^+_{Dectection}* = \frac{1}{10}$ in order to have three tasks running on the GPU without exceeding the GPU's capacity, i.e., having a feasible task set of three tasks on the GPU. Apart from `Detection`, `Localization` and `SFM` were mapped to the GPU.

$$\overline{sl} = \frac{\sum_i T_i - R_i^+}{nbTasks_{GPU}} \tag{15}$$

Figure 5 shows average task slack times derived from Eq. 15 along with different base time slices θ (x-axis) and weights, i.e., individual time slices that are derived from the base time slice θ. The figure compares equal weights (same θ), priority weights (derived from RMS whereas the highest priority has the highest value P_i), utilization weights (cf. Eq. 14) multiplied with the number of tasks mapped to the respective GPU, and the utilization weights only. For all θ values except 100ms, the priority-based time slice derivation shows the highest (best) slack times. In addition to the slack times presented in Figure 5, Figure 6 provides insights into each time slice weighting approach's standard deviation.

VI. Conclusions

This paper provides solutions towards typical response time analysis and task mapping challenges for high performance automotive systems. Therefore, formal CPU and GPU response time analyses are given that consider different scheduling paradigms (RMS and WRR), contention models, memory access and offloading patterns (synchronous vs asynchronous), task chains as well as locking, queuing, and blocking latencies. Results are presented along with applying the solutions to the WATERS 2019 challenge and its given AMALTHEA model.

The WRR scheduling has been investigated towards four different time slice derivation methods of which the default time slice times task priority derivation method has shown

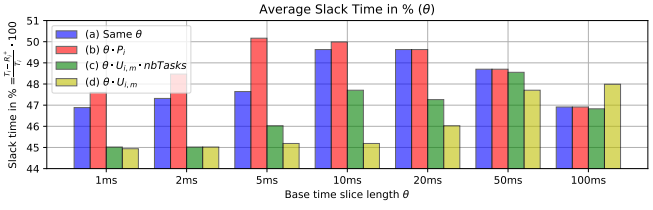

Fig. 5. Influence of time slice derivation methods and different base time slice lengths (θ) on slack times: (a) equal, (b) priority-, (c) utilization·nbTasks-, and (d) utilization-based time slices

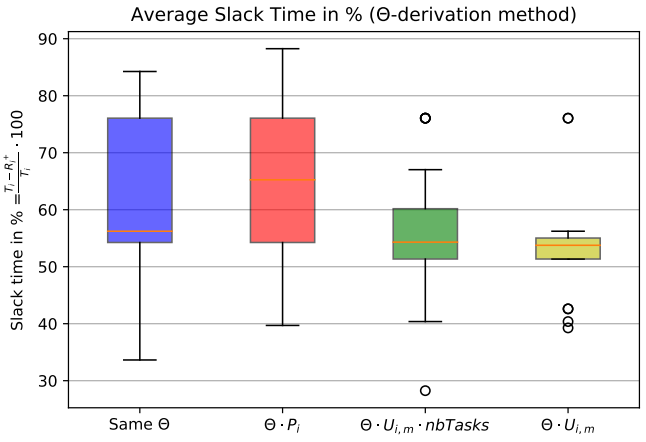

Fig. 6. Average slack time deviations of different time slice derivation methods

lower response times in average. Mappings have been calculated via a genetic algorithm and its results are presented along with various configurations such that BCET-WCET consideration, Synchronous-Asynchronous GPU offloading, time slice derivation method, as well as the Copy-Engine operation type.

Measurements of three different task to processing unit mappings, each optimized towards a different goal, and the given mapping have shown a rather sophisticated reasoning about the quality of results along various metrics such as memory contention, processing utilization standard deviation, cumulated label access costs, CE latency, task chain latency, and the sum of response times. However, each of the calculated mapping significantly outperforms the given mapping in most metrics.

One of our next steps is to further analyze blocking times on runnable level based on [16] as well as extending the current response time analysis to consider mixed-preemptive scheduling in terms of cooperative tasks, that can only be preempted at runnable bounds.

VII. Limitations & Remarks

It took us a couple of days and forum questions to fully understand the challenge model entities. We have been implementing our solutions since the publication of the AMALTHEA challenge model, not fully dedicating all resources to the challenges themselves. Most effort was spent on implementing concepts of [13] and [15].

Related progress, implementation, and information of this paper are intended to be collected in the WATERS forum corresponding thread[4].

References

[1] Jalil Boudjadar and Simin Nadjm-Tehrani. Schedulability and Memory Interference Analysis of Multicore Preemptive Real-time Systems. In *Proceedings of the Int. Conference on Performance Engineering*, ICPE, pages 263–274. ACM, 2017.

[2] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory Interference Characterization Between CPU Cores and Integrated GPUs in Mixed-Criticality Platforms. In *Proceedings of the Int. Conference on Emerging Technologies and Factory Automation*, ETFA, pages 1–10, 2017.

[3] R.I. Davis. Burns Standard Notation for Real Time Scheduling. In N. Audsley and S.K. Baruah, editors, *Real-Time Systems: The Past, The Present and The Future*, pages 38–41. Mar 2013.

[4] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. WATERS Industrial Challenge 2017. In *Int. Workshop an Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS, 2017.

[5] Arne Hamann, Dakshina Dasari, Falk Wurst, Ignacio Sanudo, Nicola Capodieci, Paolo Burgio, and Marko Bertogna. WATERS Industrial Challenge, 2019. Online: https://www.ecrts.org/forum/viewtopic.php?f=43&t=124&sid=1da5e37bde907477b2b991c411c03a03.

[6] Arne Hamann, Dirk Ziegenbein, Simon Kramer, and Martin Lukasiewycz. FMTV 2016 Verification Challenge. In *Real-Time and Embedded Technology and Applications Symposium*, RTAS, 2016.

[7] Robert Höttger, Burkhard Igel, and Olaf Spinczyk. On Reducing Busy Waiting in AUTOSAR via Task-Release-Delta-based Runnable Reordering. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*, DATE, pages 1510–1515. IEEE, March 2017.

[8] Robert Höttger, Lukas Krawczyk, Burkhard Igel, and Olaf Spinczyk. Memory Mapping Analysis for Automotive Systems. In *Work in Progress Paper, 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS, April 2019.

[9] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Raj Rajkumar. Fractional GPUs : Software-based Compute and Memory Bandwidth Reservation for GPUs. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, RTAS, pages 29–41, 2019.

[10] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real World Automotive Benchmarks For Free. In *Int. Workshop an Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS, 2015.

[11] John P Lehoczkyl. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the Real-Time Systems Symposium*, RTSS, pages 201–209, 1990.

[12] Renata Martins Gomes, Fabian Mauroner, and Marcel Baunach. Collaborative Resource Management for Multi-Core AUTOSAR OS. In Wolfgang A. Halang and Olaf Spinczyk, editors, *Betriebssysteme und Echtzeit*, pages 99–108. Springer Berlin Heidelberg, 2015.

[13] Razvan Racu, Li Li, Rafik Henia, Arne Hamann, and Rolf Ernst. Improved Response Time Analysis of Tasks Scheduled Under Preemptive Round-Robin. In *Proceedings of the Int. Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS, pages 179–184. ACM, 2007.

[14] Ignacio Sa, Paolo Burgio, and Marko Bertogna. Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System. In *Int. Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS, 2016.

[15] K. Traore, E. Grolleau, A. Rahni, and M. Richard. Response-time analysis of tasks with offsets. In *Proceedings of the Conference on Emerging Technologies and Factory Automation*, pages 1–8, Sep. 2006.

[16] Alexander Wieder and Björn Brandenburg. On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-ordered Spin Locks. In *Proceedings - Real-Time Systems Symposium*, pages 45–56, 2013.

[17] F. Wilhelmstötter. Jenetics is an advanced Genetic Algorithm, Evolutionary Algorithm and Genetic Programming library, written in modern day Java, 2019. Online available at http://jenetics.io/.

[4]WATERS forum thread for this paper: https://bit.ly/2IEJPpz, accessed 06.2019