Optimized Partitioning and Priority Assignment of Real-Time Applications on Heterogeneous Platforms with Hardware Acceleration

Daniel Casini[a,b,*], Paolo Pazzaglia[c], Alessandro Biondi[a,b] and Marco Di Natale[a,b]

[a]*TeCIP Institute, Scuola Superiore Sant'Anna, Via G. Moruzzi 1, 56124 Pisa (PI), Italy*
[b]*Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Via G. Moruzzi 1, 56124 Pisa (PI), Italy*
[c]*Saarland University, Saarland Informatics Campus, E1.3, 66123 Saarbrücken, Germany*

ARTICLE INFO

Abstract

Hardware accelerators, such as those based on GPUs and FPGAs, offer an excellent opportunity to efficiently parallelize functionalities. Recently, modern embedded platforms started being equipped with such accelerators, resulting in a compelling choice for emerging, highly computational intensive workloads, like those required by next-generation autonomous driving systems. Alongside the need for computational efficiency, such workloads are commonly characterized by real-time requirements, which need to be satisfied to guarantee the safe and correct behavior of the system. To this end, this paper proposes a holistic framework to help designers partition real-time applications on heterogeneous platforms with hardware accelerators. The proposed model is inspired by a realistic setup of an advanced driving assistance system presented in the WATERS 2019 Challenge by Bosch, further generalized to encompass a broader range of heterogeneous architectures. The resulting analysis is linearized and used to encode an optimization problem that jointly **(i)** guarantees timing constraints, **(ii)** finds a suitable task-to-core mapping, **(iii)** assigns a priority to each task, and **(iv)** selects which computations to accelerate, seeking for the most convenient trade-off between the smaller worst-case execution time provided by accelerators and synchronization and queuing delays.

1. Introduction

Embedded real-time systems have been subject to considerable changes over the last two decades. First, the advent of multi-core platforms introduced new allocation and scheduling problems [28] and the consideration of contention delays on shared resources such as memories [34, 48] and I/O devices [17]. More recently, the race towards feature-rich, predictable, safe, and secure autonomous vehicles shifted the attention to devices capable of performing a huge amount of parallel computation in an efficient way: *heterogeneous platforms*.

Heterogeneous platforms are composed of multiple cores, possibly with different characteristics. Often, they are also provided with *hardware accelerators*, such as graphic processing units (GPUs), field-programmable gate arrays (FPGAs), or digital signal processors (DSPs). The accelerators have proven to be an essential means to feasibly implement the perception and prediction software required by autonomous cars [8]. Indeed, such functionalities commonly require the usage of deep neural networks and computer vision algorithms that cannot be efficiently executed by processor cores.

However, improving the timing efficiency by means of hardware accelerators is just a piece of the puzzle. The new software introduced for autonomous driving is subject to real-time constraints, and it is required to interact and communicate *in a predictable way* with all the other time critical (and often legacy) software for the control of the vehicle [72]. This aspect calls for action on the side of the real-time scheduling analysis. For example, a computation may start on a processor core, continue on a hardware accelerator, and complete again on a core. Clearly, these behaviors require a richer modeling and more complex analysis strategies.

Furthermore, the engineers are left with several design choices, e.g., deciding the best task-to-core mapping, and deciding whether to use hardware accelerators when multiple implementations of the same functionality are available. This process is not trivial, and becomes even more complicated when it is required to guarantee that the timing constraints on the whole application are respected. For example, is it best to execute a computation slowly on a fairly empty CPU, or faster in a congested GPU? Such choices are critical and can heavily affect the performance of the system.

A possible approach to solve this issue may involve experimenting with different configurations, and empirically observing the results, seeking the best performing one. However, this exhibits the following shortcomings: **(i)** it may be highly time-consuming, **(ii)** it only allows to check a small number of configurations, without reaching holistic conclusions, **(iii)** it provides no real-time guarantees and predictability, and **(iv)** it is very unlikely to be optimal or near-optimal.

These drawbacks give rise to the need for off-line design and analysis strategies, to guarantee that all the tasks fulfill their timing constraints while taking the best decision on multiple design choices, e.g., whether to accelerate a task or not, how to allocate tasks to cores, and how to assign priorities.

While pursuing this goal, one may be tempted to rely

---

*Corresponding author
✉ daniel.casini@santannapisa.it (D. Casini);
pazzaglia@cs.uni-saarland.de (P. Pazzaglia);
alessandro.biondi@santannapisa.it (A. Biondi);
marco.dinatale@santannapisa.it (M.D. Natale)
ORCID(s): 0000-0003-4719-3631 (D. Casini); 0000-0003-0377-3327 (P. Pazzaglia); 0000-0002-6625-9336 (A. Biondi); 0000-0002-4480-8808 (M.D. Natale)

on overly simplistic models, leading to analysis and design strategies useful from a theoretical point of view but possibly far from reality or, on the opposite side, to solve a specific problem on a specific computing platform, achieving a result that does not generalize to other cases, thus limiting its usefulness for the research community.

This Paper. To avoid falling into the two aforementioned issues, this paper tries to balance between generality and specificity. To this end, we start by looking at a specific and realistic problem, the WATERS 2019 Challenge proposed by Bosch [37], which provides data (execution times, communication relations, etc.) for an Advanced Driver-Assistance System (ADAS) application running on a *NVIDIA Jetson TX-2*, a heterogeneous platform with a GPU accelerator. Based on this system configuration, we build a model to analyze its real-time behavior while generalizing to other processing platforms with other kinds of accelerators. Then, we show how to build optimization strategies to determine suitable acceleration decisions, task allocations, and priority assignments, which are then evaluated on the WATERS Challenge model in our experimentation.

This paper builds upon the workshop paper from the same authors [20] presented at the 10th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2019), as a solution to the WATERS 2019 Challenge. The original contribution [20] is extended in several directions: **(i)** the framework used in this paper now supports tasks performing acceleration in multiple separate segments of their execution; **(ii)** we study two different scheduling policies for the accelerator, i.e., round-robin and non-preemptive fixed priority, and provide the corresponding analyses; **(iii)** we propose a response time analysis for self-suspending tasks that is suitable for linear optimization, inspired by (and extending) the approach presented in [59]; **(iv)** we provide a comprehensive optimization problem, extensively discussing all the constraints and presenting a corresponding proof for each of them; **(v)** we provide a comprehensive evaluation based on the WATERS Challenge model that studies different objective functions and scheduling policies.

Paper Structure. The remainder of the paper is organized as follows. Section 2 introduces the WATERS 2019 Challenge. Section 3 presents our modeling solution to describe an application running on a heterogeneous platform with a hardware accelerator. Section 4 presents methods to analyze heterogeneous applications using self-suspending task theory. Section 5 illustrates an optimization problem to take several design-level decisions in an optimal way. Section 6 presents the results of our experimental evaluation. Section 7 discusses the related work. Finally, Section 8 concludes the paper.

2. The WATERS 2019 Challenge

The WATERS 2019 Challenge [37] by Bosch represents an interesting opportunity to explore a realistic design of a modern ADAS application, implemented on a heterogeneous platform. The Challenge provides an Amalthea APP4MC
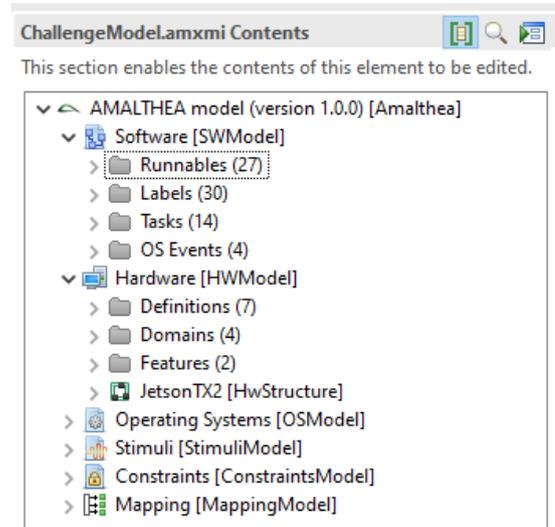

AMALTHEA Contents Tree

Figure 1: Amalthea model provided with the WATERS 2019 Challenge.

model (see Figure 1) representing an ADAS prototype. The application is composed of nine tasks performing computations from the sensors input to the steering command. Tasks have communication dependencies as shown in Figure 2. The model uses the Jetson TX-2 as a reference platform, with six cores organized in two processor islands. The first island includes four ARMv8 A57 cores running at 1.9 GHz, while the latter contains two 2 Ghz ARMv8 Denver cores. The platform is also provided with an iGPU (integrated GPU), which allows accelerating some strongly-parallel computations.

The Amalthea APP4MC model includes additional information about the structure of the tasks. For each task, periods and deadlines are specified. Each task is then composed of multiple segments of computation, called *runnables*, according to the AUTOSAR standard [1]. Each runnable implements a specific function and is characterized by a set of possible execution time values, depending on where it is implemented. If the runnable is designed to run on a CPU, it presents the execution time information for the A57 cores and for the Denver cores; if it is designed to be accelerated, it includes the execution time information computed for the GPU. In both cases, the minimum, average, and maximum execution times are reported. The model also specifies which *labels* (i.e., shared memory locations) a runnable is reading or writing. This information is particularly useful for deriving the communication dependencies.

For each task that can be accelerated, the Amalthea model provides two task objects. The first one contains the preprocessing and postprocessing runnables that are executed on a CPU when the main activity is executed on the GPU: such additional runnables are required for passing the inputs to the GPU function and getting back the results. The second object involves the runnables executing on the GPU and performing

---

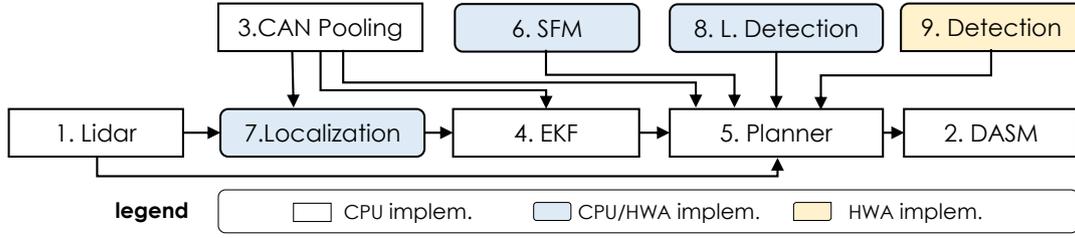[1] The AUTOSAR standard, version 4.3. http://www.autosar.org

Figure 2: Processing chains of the WATERS 2019 Challenge. The number on the side of each task represent the task ID, imported from the challenge data.

the computation. The WATERS 2019 challenge also provides (in another model file) an alternative version of some tasks, with their parameters in the case in which they are executed on a CPU (when allowed).

Building upon this realistic case study, we derive a more general model that is used to analyze and optimize a generic real-time system running on a heterogeneous platform. The NVIDIA GPU device considered in the WATERS 2019 Challenge involves scheduling policies that are not publicly disclosed by the hardware vendor [3], for which details may only be experimentally inferred through reverse engineering [3, 54] or approximated [37]. Conversely, we consider two predictable scheduling policies, round-robin and non-preemptive fixed-priority, which may be either directly adopted in the hardware accelerator or be enforced by an application-level scheduler that handles acceleration requests on the CPU side [9, 31].

3. System Model

The system model analyzed in the following of the paper builds upon the proposal the WATERS 2019 Challenge, and it is extended to make it representative of processing platforms with arbitrary heterogeneous cores and hardware accelerators.

Table 1 summarizes the main symbols used in this paper.

3.1. Platform Model

This paper considers a heterogeneous embedded real-time platform composed of a set $\mathcal{P} = \{p_1, \dots, p_m\}$ of processor cores. Each core $p_k \in \mathcal{P}$ is assigned a *type* that determines the execution-time profile of the functionality implemented in that core. The type can be easily extended to cover other aspects of interest, such as power consumption and cache memory size, which however are out of the scope of this work.

For the sake of simplicity, we consider that the platform provides a single hardware accelerator, which is referred to as $\mathcal{H}$. Nonetheless, the present analysis can be easily extended for the case of multiple independent accelerators at the expense of a more complex notation.

3.2. Task Model

The application implemented in the platform comprises a set $\Gamma = \{\tau_1, \dots, \tau_n\}$ of periodic real-time tasks. Tasks are executed on cores according to a *partitioned fixed-priority*

preemptive scheduling, where each task τ_i is statically assigned to a processor p_k and a unique priority π_i. This configuration guarantees a high predictability while being representative of systems capable to run on hardware accelerators, i.e., those based on Linux, where partitioned fixed-priority scheduling can be achieved by assigning tasks to the SCHED_FIFO scheduling class and specifying affinities with the sched_setaffinity() system call.

We denote with $\Gamma_k \subseteq \Gamma$ the subset of tasks mapped on core p_k, with $\bigcup_k \Gamma_k = \Gamma$ and $\bigcap_k \Gamma_k = \emptyset$. The set of all tasks with priority higher (resp. lower) than π_i is denoted with $hp(\tau_i)$ (resp. $lp(\tau_i)$). Similarly, the set of all tasks mapped on core p_k and with priority higher (resp., lower) than π_i is denoted with $hp_k(\tau_i)$ (resp., $lp_k(\tau_i)$). Each task τ_i releases a potentially infinite sequence of instances called *jobs*, each separated by T_i time units. Each job needs to complete within its relative deadline $D_i \le T_i$, i.e., within D_i units of time from its release. A task is said to be *schedulable* if all of its jobs always complete within D_i time units from its release.

Furthermore, each task $\tau_i \in \Gamma$ is composed of a sequence of code *segments* executed sequentially, with $\rho_i = \{\rho_{i,1}, \dots, \rho_{i,w}\}$ denoting the set of all segments $\rho_{i,j} \in \rho_i$. A job of task τ_i starts with the execution of segment $\rho_{i,1}$, and any other segment $\rho_{i,j}$ with $1 < j \le w$ starts executing only after the completion of $\rho_{i,j-1}$. Each segment represents a functionally distinct fragment of code, and its implementation can be either provided for execution on processor cores, on the hardware accelerator, or both. To this end, each segment is characterized by an implementation type $\iota_{i,j} \in \{\text{CPU}, \text{HWA}, \text{CPU-HWA}\}$, where $\iota_{i,j} = \text{CPU}$ if the segment can only be executed on a core; $\iota_{i,j} = \text{HWA}$ if the segment can only be executed on the hardware accelerator $\mathcal{H}$, and $\iota_{i,j} = \text{CPU-HWA}$ if two implementations are provided and hence the segment can be executed either on a core or on the accelerator.

When both implementations on cores and the hardware accelerator are provided, it necessary to determine where the segment actually executes. Hence, each segment is further characterized by a parameter $a_{i,j} \in \{\text{T}, \text{F}\}$ denoting if τ_i offloads its computations to the accelerator ($a_{i,j} = \text{T}$) or not ($a_{i,j} = \text{F}$). Clearly, if $\iota_{i,j} = \text{CPU}$, then $a_{i,j} = \text{F}$, and if $\iota_{i,j} = \text{HWA}$, $a_{i,j} = \text{T}$. The set of accelerated segments of task τ_i is denoted with $\rho_i^A = \{\rho_{i,j} \mid a_{i,j} = \text{T}\}$), while the set of segments that *may be accelerated* is denoted with $\rho_i^H = \{\rho_{i,j} \mid \iota_{i,j} \in \{\text{CPU-HWA}, \text{HWA}\}\}$.
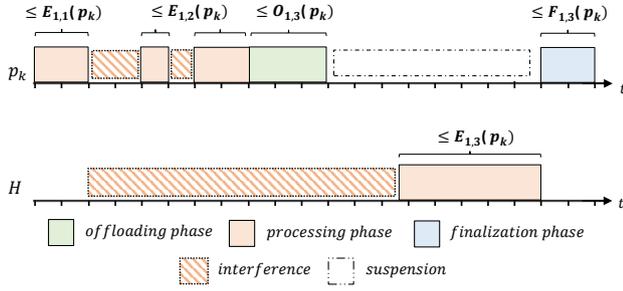
Figure 3: Example of schedule under the proposed model.

3.3. Offloading Mechanism

We consider a *synchronous*, *suspension-based* offloading mechanisms to the hardware accelerator. Namely, when a segment $\rho_{i,j}$ of $\tau_i \in \Gamma_k$ is accelerated (i.e., $a_{i,j} = \top$), it first executes the *offloading phase*, i.e., it executes a first chunk of code on its processor p_k to perform the initial operations and to prepare the data to offload to the accelerator. When it completes, the task suspends on p_k, and its execution continues on $\mathcal{H}$ in the *processing phase*. Upon completion, the task is awakened on p_k, and retrieves the outputs produced by the accelerator, possibly executing further processing in the *finalization phase*, which terminates the execution of $\rho_{i,j}$. This behavior is representative of most hardware accelerators, e.g., those based on GPUs [37] and FPGAs [9]. Conversely, when a segment is not accelerated, i.e., $a_{i,j} = \bot$, it performs only the processing phase on p_k.

For such three phases of an arbitrary segment $\rho_{i,j}$, we introduce $O_{i,j}(p_k)$, $E_{i,j}(p_k)$, and $F_{i,j}(p_k)$ to denote the worst-case execution times (WCETs) of the offloading, processing, and finalization phase, respectively. Note that, due to the platform heterogeneity, the WCETs depend on the type of the core p_k. The offloading and finalization phases of $\rho_{i,j}$ have a positive WCET only when $a_{i,j} = \top$. In this case, the offloading and finalization phase run on the core p_k where the corresponding task is allocated, with WCETs $O_{i,j}(p_k)$ and $F_{i,j}(p_k)$, respectively. The execution phase instead runs on the hardware accelerator $\mathcal{H}$, and its WCET is denoted by $E_{i,j}(\mathcal{H})$. When $a_{i,j} = \bot$, the segment is not accelerated and composed of the execution phase only, which occurs on its core p_k.

Figure 3 shows an example of a possible schedule under the execution model of this paper. Task τ_1 is composed of three segments. The first two are not accelerated, and hence they are only composed of a single processing phase each, executing on a core p_k, with durations bounded by the parameters $E_{1,1}(p_k)$ and $E_{1,2}(p_k)$, respectively. The third segment is accelerated. Therefore, it is composed of an offloading and finalization phase running on p_k for at most $O_{1,3}(p_k)$ and $F_{1,3}(p_k)$, respectively, and a processing phase running on the accelerator for at most $E_{1,3}(\mathcal{H})$.

For brevity, the worst-case execution time of a segment $\rho_{i,j}$ on a core p_k is denoted as follows:

$$C_{i,j}(p_k, a_{i,j}) = \begin{cases} E_{i,j}(p_k) & \text{if } a_{i,j} = \bot \\ O_{i,j}(p_k) + F_{i,j}(p_k) & \text{if } a_{i,j} = \top \end{cases}$$

Table 1
Table of symbols

Symbol	Description
p_k	k-th processor core
$\mathcal{H}$	hardware accelerator
τ_i	i-th task
Γ_k	tasks assigned to p_k
T_i	i-th task period
D_i	i-th task deadline
R_i	WCRT bound of τ_i
$\rho_{i,j}$	j-th segment of τ_i
ρ_i	set of τ_i's segments
$\iota_{i,j}$	implementation type of $\rho_{i,j}$
$a_{i,j}$	equal to $\top$ iff $\rho_{i,j}$ executes on $\mathcal{H}$
ρ_i^A	set of accelerated segments
ρ_i^H	segments that *may* be accelerated
$O_{i,j}(p_k)$	WCET of the offloading phase of $\rho_{i,j}$
$E_{i,j}(p_k)$	WCET of the processing phase of $\rho_{i,j}$
$F_{i,j}(p_k)$	WCET of the finalization phase of $\rho_{i,j}$
$C_{i,j}(p_k, a_{i,j})$	WCET of $\rho_{i,j}$ on a core p_k
ω_x	x-th chain
Ω	set of all chains

and the WCET of the whole task τ_i, running on a core p_k, is hereafter referred to in compact notation as C_i, defined as follows:

$$C_i = \sum_{\rho_{i,j} \in \rho_i} C_{i,j}(p_k, a_{i,j}).$$

Each task τ_i is also characterized by a worst-case response time (WCRT), which is the longest time span elapsed between the release and the completion of any of its jobs. Usually, the exact WCRT is difficult to derive, but an upper-bound R_i can be found with a suitable response-time analysis. Then, if $R_i \leq D_i$ for each task $\tau_i \in \Gamma$, the system is said to be schedulable. Methods to bound the WCRT under the configuration proposed in this paper are reported in Section 4.

3.4. Task Chains

As discussed in Section 2, tasks may have functional dependencies, e.g., producer-consumer relationships. This is modeled by denoting a sequence of communicating tasks with a *processing chain* ω_x, where ω_x represents an ordered list of tasks in a two-by-two producer-consumer relationship, i.e., $\omega_x = (\tau_{x_1}, \tau_{x_2}, \dots)$. As shown in Figure 2, each task may belong to multiple chains, thus forming a graph of dependencies. The set of all the chains is denoted as Ω. In this paper, task chains are assumed to be time-triggered, i.e., each task $\tau_i \in \omega_x$ in the chain is periodically released according to its period T_i.

In the next section, we discuss how to compute the end-to-end latency of such chains, and we introduce a response-time analysis for the tasks running on both the CPUs and the accelerator.

4. End-to-End Latency Analysis

This section shows how to bound the end-to-end latency of task chains running on top of a heterogeneous platform,

where each task is statically mapped in one of the CPU cores, but some of its computation can be offloaded to a hardware accelerator multiple times during execution.

We recall from prior work [27] that the end-to-end latency L_x of a (time-triggered) processing chain ω_x is bounded by:

$$L_x = \sum_{\tau_i \in \omega_x} (R_i + T_i) - T_{\text{first}}, \qquad (1)$$

considering that the external event triggering the chain arrives synchronously with the release of the first task τ_{first} of the chain. To apply Equation (1), the worst-case response-time R_i of each task in the chain needs to be bounded. Next, we show how to leverage existing results on *self-suspending tasks* theory [25] to analyze the behavior of a task performing acceleration on a core $p_k \in \mathcal{P}$.

4.1. Self-Suspending Tasks

To enable the presentation of the adopted analysis techniques, the segmented self-suspending task model is here briefly introduced.

A segmented self-suspending task – hereafter denoted with the symbol τ_i^{ss} to better differentiate from the task model introduced in Section 3 – is characterized by an ordered sequence of N_i^S regions $(l_{i,1}, \dots, l_{i,j}, \dots, l_{i,N_i^S})$, representing alternating code executions and self-suspensions. Here we intentionally use the new concept of regions instead of segments, since they serve a different purpose, and will be used later to map the original task model of the paper to the current self-suspended tasks. Both execution and suspension regions are characterized by a bounded worst-case duration. If $l_{i,j}$ is an execution region, its WCET is denoted by $C_{i,j}$; otherwise, if $l_{i,j}$ is a suspension region, its duration is bounded by $S_{i,j}$. Overall, the duration bounds of execution and suspension regions is represented by the tuple $\langle C_{i,1}, S_{i,1}, \dots, S_{i,N_i^S-1}, C_{i,N_i^S} \rangle$. Analogously to Section 3, a self-suspending task τ_i^{ss} is periodically released with period T_i. Furthermore, each of them is characterized by a relative deadline $D_i \leq T_i$ and fixed priority π_i. Once a self-suspending task is released, the first execution region is also released. If the $(j-1)$-th execution region of τ_i^{ss} completes at time t, the $(j+1)$-th execution region is released no later than time $t + S_{i,j}$. It is worth noting that a task τ_i^{ss} may have no suspension regions but still be modeled as a self-suspending task: in that case, it will only consist of one execution region $l_{i,1}$ with execution time $C_{i,1}$.

4.1.1. Mapping Accelerations to Self-Suspensions

Γ_k^{ss} indicates the set of self-suspending tasks running on core p_k. A task $\tau_i \in \Gamma_k$ making use of hardware acceleration can be analyzed as a corresponding self-suspending task $\tau_i^{\text{ss}} \in \Gamma_k^{\text{ss}}$ by establishing the following mapping.

Given an arbitrary task $\tau_i \in \Gamma_k$, each non-accelerated segment $\rho_{i,j}$ (i.e., with $a_{i,j} = \text{F}$) is firstly mapped to an execution region of $\tau_i^{\text{ss}} \in \Gamma_k^{\text{ss}}$ with WCET equal to $E_{i,j}(p_k)$. Conversely, an accelerated segment $\rho_{i,j}$ ($a_{i,j} = \text{T}$) is mapped as follows:
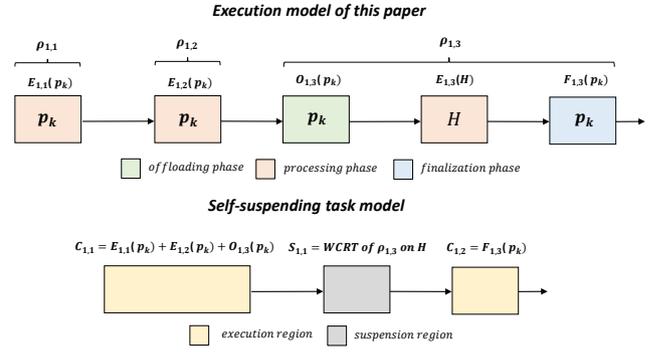

Figure 4: Example mapping of the model proposed in this paper to the self-suspending task model for analysis purpose.

1. the offloading phase of $\rho_{i,j}$ is mapped to an execution region of $\tau_i^{\text{ss}} \in \Gamma_k^{\text{ss}}$ with WCET equal to $O_{i,j}(p_k)$;

2. the processing phase is mapped to a suspension region of $\tau_i^{\text{ss}} \in \Gamma_k^{\text{ss}}$;

3. the finalization phase is mapped to an execution region of $\tau_i^{\text{ss}} \in \Gamma_k^{\text{ss}}$ with WCET equal to $F_{i,j}(p_k)$.

Finally, consecutive execution regions (i.e., not separated by a suspension region) are merged into a single region, and their WCETs are summed (as they all sequentially execute on the same core without suspensions). We denote by $\mathcal{Q}(\rho_{i,j}) = l_{i,x}$ the suspension region that corresponds to segment $\rho_{i,j}$, if any.

While the WCET of each execution segment of τ_i^{ss} is thus known from the task parameters of τ_i, the duration of each suspension region depends on the response time of the corresponding task executing on the hardware accelerator, considering that other segments may be contending the same computational resource. Consequently, bounding the maximum duration of a self-suspension regions involves bounding the response-time of each accelerated segment in the hardware accelerator. Clearly, this requires knowing the scheduling policy adopted on the accelerator.

Figure 4 shows an example of mapping between the model proposed in this paper and the self-suspending task model. Task τ_1 in the example is composed of three segments, $\rho_{1,1}$, $\rho_{1,2}$, and $\rho_{1,3}$. Only the third one is accelerated. Hence, the corresponding self-suspending task τ_1^{ss} running on p_k is composed of an execution region with WCET $C_{i,1} = E_{1,1}(p_k) + E_{1,2}(p_k) + O_{1,3}(p_k)$, a suspension region with length bounded by $S_{i,1}$, and an execution region with WCET $F_{1,3}(p_k)$. The parameter $S_{i,1}$ is unknown beforehand and needs to be bounded by analyzing the worst-case response time of the accelerated segment running on $\mathcal{H}$.

Before proceeding with the details of the accelerator scheduling, we discuss the response-time analysis for self-suspending tasks.

4.2. Response-Time Analysis with Self Suspensions

Several analysis techniques for self-suspending tasks are available in the literature (please refer to the work by Chen et al. [25] for a detailed review). Next, we explore a method for

analyzing a self-suspending task which is particularly suited to be applied to our optimal mapping problem while still providing good schedulability performance.

In the following paragraph the analysis is performed considering a task set of self-suspending tasks Γ^{ss} obtained using the mapping of Section 4.1.1 from the task set under analysis.

4.2.1. Jitter-based Analysis

A quite popular analysis technique for self-suspending tasks considers the timing effects of suspensions of interfering tasks as *release jitter*, while the suspension of the task under analysis is modeled as an inflation of its execution time [25]. Following this model, the response time R_i of a self-suspending task τ_i^{ss} running on core p_k can be upper bounded by the least positive solution of the following recursive equation [25]:

$$R_i = C_i + S_i + \sum_{\tau_h^{ss} \in hp_{p_k}(\tau_i^{ss})} \left\lceil \frac{R_i + J_h}{T_h} \right\rceil C_h, \qquad (2)$$

where $C_i = \sum_{j=1}^{N_i^S} C_{i,j}$ and $S_i = \sum_{j=1}^{N_i^S - 1} S_{i,j}$, while $hp_{p_k}(\tau_i^{ss})$ is the set of tasks with priority higher than τ_i^{ss} allocated to p_k, and the value J_h is an upper bound of the jitter induced by the overall self-suspension regions (if any) of τ_h^{ss}. The value $J_h = R_h - C_h$ is a valid bound on the jitter [53]. If the task τ_h^{ss} has no suspension region, then $J_h = 0$.

As previously discussed, when adopting synchronous offloading, the execution region of each accelerated segment of a task τ_i can be mapped to a suspension region. Thus the task model presented in this paper can be mapped to a self-suspending task, and Equation (2) can be used to compute an upper bound of the worst-case response time.

4.2.2. Linearizing the Jitter-based Analysis

One of the advantages of the analysis technique presented above is its fitness in being linearized to be encoded in a mixed-integer linear programming (MILP) formulation.

To better understand the following steps, we first recall that a sufficient schedulability test for any task τ_i scheduled with a fixed-priority algorithm can be obtained by checking if there exists a value in $[0, D_i]$ that satisfies the following inequality [43]:

$$\exists t \in [0, D_i] : W_i(t) \leq t, \qquad (3)$$

where $W_i(t)$ is a function bounding the overall processing time required by τ_i and all other tasks running in the same core that can delay τ_i in any time window of length t. Intuitively, the result follows because, if Equation (3) holds, then the processing time provided by the core (i.e., t) is enough to satisfy the demand of $W_i(t)$ time units. For the case of self-suspending tasks, considering τ_i^{ss} as the task under analysis, the processor demand is expressed as (see Equation (2)):

$$W_i(t) = C_i + S_i + \sum_{\tau_h^{ss} \in hp_{p_k}(\tau_i^{ss})} \left\lceil \frac{t + J_h}{T_h} \right\rceil C_h, \qquad (4)$$

with $J_h = R_h - C_h$ if τ_h^{ss} has at least one suspension region, and $J_h = 0$ otherwise.

Pazzaglia et al. [59] showed that a very accurate (less than 2% pessimism), but sufficient schedulability test can be obtained for a wide range of task models by just checking the inequality of Equation (3) in a limited set of points $t \in [0, D_i]$. This result is particularly helpful when the test must be encoded in an MILP formulation, as it helps in drastically reducing the number of optimization variables and constraints and hence the time required to solve the optimization problem.

In the present work, the approach in [59] is leveraged to encode the optimization problem aimed at finding solutions for the task-to-core and priority assignment, which optimizes the end-to-end latency of the processing chains for applications running on a heterogeneous system. In particular, we propose an extension of the method used in [59] to handle the case of self-suspending tasks, on a per-core level.

The method in [59] builds upon an observation first made by Park and Park [57] according to which effective schedulability tests can be obtained by just checking the points in time at which the *last activations* of tasks occur in the worst-case scheduling pattern of the task under analysis. Under the jitter-based modeling of self-suspending tasks, the worst-case response time bound of Equation (2) is obtained under a release pattern defined as follows: **(i)** τ_i^{ss} is released at time $t = 0$ (without loss of generality) with no jitter and experiences a worst-case suspension of S_i time units, **(ii)** all high-priority are ready to execute at $t = 0$ after experiencing maximum jitter; and **(iii)** all successive instances are released with zero jitter [25]. We hereafter refer to this release pattern with the symbol Λ, which is illustrated in Figure 5.

The following theorem provides the schedulability points of interest (following the method of [59]) for a self-suspending task τ_i^{ss}, leveraging the release pattern Λ.

Theorem 1. *Consider a task $\tau_i^{ss} \in \Gamma_k^{ss}$ under analysis and assume that τ_i^{ss} and all tasks in $hp_{p_k}(\tau_i^{ss})$ are released according to Λ. A higher priority task $\tau_h^{ss} \in hp_{p_k}(\tau_i^{ss})$ has more than one activation in $[0, D_i]$ if $T_h - J_h < D_i$ and its last activation in the same interval occurs at time*

$$V_{i,h} = \left\lfloor \frac{D_i + J_h}{T_h} \right\rfloor \cdot T_h - J_h. \qquad (5)$$

Proof.

By definition of Λ, the first periodic instance of each task τ_h^{ss} starts at time $-J_h$, as it is subject to maximum release jitter J_h and it is ready to execute at time $t = 0$. Then, the second activation of each task τ_h^{ss} occurs at time $T_h - J_h$. Hence, τ_h^{ss} has more than one activation in $[0, D_i]$ if $T_h - J_h < D_i$. If this latter condition holds, the length of the interval in which the periodic instances of τ_h^{ss} overlap with the scheduling window under analysis $[0, D_i]$ is $D_i + J_h$. Since there are $\lfloor (D_i + J_h)/T_h \rfloor$ activations of τ_h that are fully-contained in this interval, the last one starts $\lfloor (D_i + J_h)/T_h \rfloor T_h$ time units after the first one, which occurs at time $-J_h$. Hence
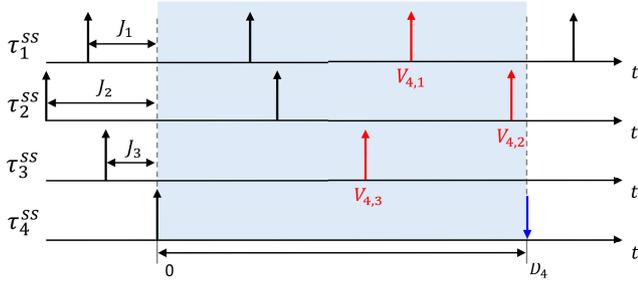
Figure 5: Example of release pattern Λ with 4 tasks. The red arrows represent the last activation of the corresponding tasks in the interval $[0, D_4]$ (highlighted in light blue).

Figure 6: "Mirrored" version of self-suspending task of Figure 4, as it is perceived from the perspective of the hardware accelerator.

Equation (5) and the theorem follows. $\square$

Figure 5 shows an example where the last activation instants computed as in Theorem 1 are highlighted in red.

Let now $\mathcal{J}_i$ be the set of initial release jitters J_h of each task $\tau_h^{ss} \in hp_k(\tau_i^{ss})$ in Theorem 1, i.e.,

$$\mathcal{J}_i = \bigcup_{\tau_h^{ss} \in hp_k(\tau_i^{ss})} J_h. \tag{6}$$

The set of checkpoints $\mathcal{V}_i(\mathcal{J}_i)$ for analyzing τ_i^{ss} is then obtained by the union of the points of Equation (5) computed for all the interfering tasks with their corresponding jitter in $\mathcal{J}_i$, plus the deadline of the task τ_i^{ss}, i.e.,

$$\mathcal{V}_i(\mathcal{J}_i) = \left\{ \bigcup_{\tau_h^{ss} \in hp_k^*(\tau_i^{ss})} V_{i,h} \right\} \cup \left\{ D_i \right\}, \tag{7}$$

where $hp_k^*(i) = \{\tau_h^{ss} \in hp_k(\tau_i^{ss}) : T_h - J_h < D_i\}$. By construction, the set $\mathcal{V}_i(\mathcal{J}_i)$ contains at least one point. Hereafter, we refer to an arbitrary check-point in $\mathcal{V}_i(\mathcal{J}_i)$ with the variable $v_{i,g} \in \mathcal{V}_i(\mathcal{J}_i)$.

By putting together Equation (2) and the set of checkpoints given by Equation (7), the resulting schedulability test consists in verifying the following condition for each task $\tau_i^{ss} \in \Gamma_k^{ss}$:

$$\exists v_{i,g} \in \mathcal{V}_i(\mathcal{J}_i) \mid C_i + S_i + \sum_{\tau_h^{ss} \in hp_k(\tau_i^{ss})} \left\lceil \frac{v_{i,g} + J_h}{T_h} \right\rceil C_h \leq v_{i,g}. \tag{8}$$

This formulation requires the knowledge of the set of jitter bounds $\mathcal{J}_i$ of each interfering task. In this work, we use $J_h = D_h - C_h$ as a safe bound, which can be easily encoded in an MILP formulation. This bound follows by noting that, since $J_h = R_h - C_h$ is a valid bound [53], then under the assumption that $D_h \geq R_h$, also $J_h = D_h - C_h$ holds. $J_h = R_h - C_h$ could provide, in principle, a less pessimistic solution. However, this choice adds a circular dependency in the response time of the target task τ_i^{ss} with the response time R_h of the higher priority tasks $\tau_h^{ss} \in hp_k(\tau_i^{ss})$. This dependency can be broken by introducing another set of variables (e.g., by bounding
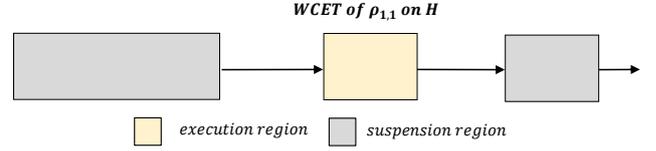
R_h with a suitable checkpoint of $\mathcal{V}_h$), which has factorial complexity with respect to the number of tasks, and quickly makes the optimization impractical. Again, note that the results presented here (which adopt the self-suspending task model) can be directly applied to the task model of this paper by leveraging the mapping of Section 4.1.1.

4.3. Response-Time Analysis for Accelerators

As previously discussed, the length of each suspension region of a task is bounded by the maximum response time of the corresponding activity executed by the accelerator. However, the scheduling policies used in hardware accelerators are often unknown: for example, internal details of the popular NVIDIA GPUs are not disclosed. Therefore, in this paper, we consider two predictable scheduling policies: *round-robin* (RR) and *non-preemptive fixed-priority* (NP-FP). Such policies can be either directly adopted by hardware accelerators, or externally enforced by application-level schedulers running on the processor cores [9, 31] The latter can be implemented by treating the accelerator as a shared resource, thus maintaining a queue of accelerated activities (on the processing cores) and providing one activity at a time to the accelerator.

Under RR scheduling, each task executes up to one accelerated execution phase in a cyclic fashion. Conversely, under NP-FP, accelerated execution phases are executed to completion and scheduled according to the priorities of the corresponding tasks. These two policies are deemed suitable for hardware accelerators, because they allow for predictable scheduling and execution to completion (i.e., without preemptions), thus favoring cache coherence, and do not require to save and restore the context to implement preemption.

Furthermore, round-robin and NP-FP also have different interesting features. The first one ensures a fair and starvation-free access to the accelerator. In contrast, the second one guarantees shorter delays to high-priority tasks, thus being more suitable for latency-sensitive tasks.

By supporting multiple scheduling policies for the hardware accelerator we highlight the generality of our approach, which can also be easily extended to other scheduling policies to serve specific purposes.

Deriving the WCRT bounds. To derive the WCRT bounds, the execution behavior of the segments running on the hardware accelerator may be modeled as an equivalent, but "mirrored", self-suspending task (shown in Figure 6), where execution regions runs on the accelerator and suspensions regions correspond to phases running on a processor core [30]. However, an analysis exploiting this modeling approach would

require knowing the suspension time (which in this case would occur when the task runs on the cores), as in Equation (2), thus creating a cyclic dependency.

To break the cycle, we compute individual upper-bounds on the suspension time of each execution phase running on the accelerator. In this way, the segment under consideration can be analyzed as a normal periodic task without self-suspension that is subject to interference due to self-suspending tasks, hence eliminating the dependency on the suspension time when deriving its WCRT bound.

4.3.1. Round-Robin Scheduling

First, we consider the case in which the hardware accelerator implements a round-robin scheduling policy.

Lemma 1 proposes a bound on the suspension time of a single accelerated segment under round-robin scheduling.

Lemma 1. *Consider a segment $\rho_{i,j} \in \rho_i^A$ of a task $\tau_i \in \Gamma$, and let $l_{i,x} = \mathcal{Q}(\rho_{i,j})$ be the corresponding suspension region of the matching task τ_i^{ss}. Then, under round-robin scheduling, the duration of $l_{i,x}$ is bounded by:*

$$S_{i,x} = E_{i,j}(\mathcal{H}) + \sum_{\rho_{h,u} \in \mathcal{I}(\tau_i)} E_{h,u}(\mathcal{H}), \qquad (9)$$

where $\mathcal{I}(\tau_i)$ is the set containing, for each task $\tau_h \in \Gamma \setminus \tau_i$, the accelerated segment $\rho_{h,u} \in \rho_h^A$ with longest execution phase among all segments in ρ_h^A (if ρ_h^A is not empty).

Proof.
Due to the round-robin scheduling policy, each accelerated segment can be delayed at most once for each other segment of each other task $\tau_h \in \Gamma \setminus \tau_i$. Due to the synchronous offloading mechanism, each task τ_h can have at most one pending acceleration at a time. Consequently, each other task $\tau_h \in \Gamma \setminus \tau_i$ can delay $\rho_{i,j}$ with at most one of its accelerated segment. The lemma follows by noting that, for each $\tau_h \in \Gamma \setminus \tau_i$, $\mathcal{I}(\tau_i)$ contains the interfering segment with the longest execution phase. $\square$

Given the bound on the suspension time for an individual accelerated region, an overall bound for the suspension time of self-suspending task τ_i^{ss} can be computed as $S_i = \sum_{\rho_{i,j} \in \rho_i^A} S_{i,x}$, where $l_{i,x} = \mathcal{Q}(\rho_{i,j})$.

This bound can be further improved by considering the ratio of periods among tasks. The improvement is not discussed here for the sake of simplicity.

4.3.2. Fixed-Priority Non-Preemptive Scheduling

When the hardware accelerator provides an NP-FP scheduler, the execution phase of each accelerated segment runs on the accelerator with the same priority of the corresponding task.

Under this setting, the suspension time due to an arbitrary accelerated segment $\rho_{i,j} \in \rho_h^A$ is bounded by Lemma 2.

Lemma 2. *Consider a segment $\rho_{i,j} \in \rho_i^A$ of a task $\tau_i \in \Gamma$, and let $l_{i,x} = \mathcal{Q}(\rho_{i,j})$ be the corresponding suspension region of the matching task τ_i^{ss}. Then, under non-preemptive fixed-priority scheduling, the duration of $l_{i,x}$ is bounded by $S_{i,x} = \Phi_{i,x} + E_{i,j}(\mathcal{H})$, where $\Phi_{i,x}$ is the last positive solution of the following equation:*

$$\Phi_{i,x} = B_i + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{\Phi_{i,x} + D_h - G_h}{T_h} \right\rceil G_h \qquad (10)$$

where $B_i = \max \left\{ E_{l,v}(\mathcal{H}) \mid \tau_l \in lp(\tau_i) \wedge \rho_{l,v} \in \rho_l^A \right\}$, $G_h = \sum_{\rho_{h,u} \in \rho_h^A} E_{h,u}(\mathcal{H})$, and $\forall \tau_h \in hp(\tau_i)$, $R_h \leq D_h$ holds.

Proof.
Since for each task τ_i at most one accelerated segment $\rho_{i,j}$ can be pending at a time on $\mathcal{H}$, as previously discussed, the duration of each suspension region $l_{i,x} = \mathcal{Q}(\rho_{i,j})$ can be analyzed by bounding the response-time of a normal periodic task without self-suspension, with a WCET equal to $E_{i,j}(\mathcal{H})$, which is subject to interference due to "mirrored" self-suspending tasks. Such mirrored self-suspending tasks have execution regions running in the accelerator, and suspension regions that corresponds to executions on the processor cores. Interference due to high-priority tasks can be then accounted for in the analysis as for periodic tasks subject to a release jitter [25]. For each high-priority task, Equation (10) considers the largest possible jitter (from the perspective of the execution region of $\rho_{i,j}$ running in the accelerator), and the overall interfering WCET G_h of each interfering task in the accelerator. The lemma follows as a simplified instance of the jitter-based analysis for self-suspending task in [18] where the task under analysis has a single segment (which corresponds to the accelerated execution phase of interest), and therefore does not self-suspend. $\square$

As for preemptive fixed-priority scheduling, the bound implied by Lemma 2 can be rewritten as follows. The duration of $l_{i,x}$ is bounded by any value $t + E_{i,j}(\mathcal{H})$, where $t \in [0, D_i - E_{i,j}(\mathcal{H})]$ (with $l_{i,x} = \mathcal{Q}(\rho_{i,j})$) satisfies the following inequality:

$$B_i + \sum_{\tau_h \in hp(\tau_i)} \left\lceil \frac{t + J_h^A}{T_h} \right\rceil G_h \leq t, \qquad (11)$$

where $J_h^A = D_h - G_h$, and G_h is defined as in Lemma 2.

Note that Equation (11) is similar to the ones for analyzing self-suspending provided in Section 4.2.2. The scheduling points derived in Theorem 1 and later used in Equation (8) can hence be extended to be applicable to Equation (11), provided that a suitable set $\mathcal{J}_{i,x}^A = \bigcup_{\tau_h^{ss} \in hp_k(\tau_i^{ss})} J_h^A$ of jitters for the acceleration request corresponding to each segment $l_{i,x}$ is computed. Then, the set of check-points is defined as $\mathcal{V}_i(\mathcal{J}_i^A) = \left\{ \bigcup_{\tau_h^{ss} \in hp_k^*(\tau_i^{ss})} V_{i,h}^A \right\} \cup \{D_i\}$, where $V_{i,h}^A$ is defined as in Theorem 1 by replacing J_h with J_h^A.

In this way, the duration of $l_{i,x}$ can be bounded by restricting the search of a value t that satisfies Equation (11) in the finite set $\mathcal{V}_i(\mathcal{J}_i^A)$.

5. Optimization Problem

This section presents a mixed-integer linear programming (MILP) formulation of the optimization problem. The main objectives of the proposed formulation are the following:

- minimize either the end-to-end latency of the processing chains or the task WCRT bounds, according to the proposed objective function(s);

- select the most convenient task-to-core placement, accounting for both the change in the WCETs depending on the core type and the interference due to other tasks assigned to the same core;

- determine whether to accelerate tasks to find the most convenient trade-off between shorter WCETs occurring when a task is accelerated and longer delays in the accelerator when many activities are offloaded;

- optimize the priority assignment; and

- ensure schedulability, i.e., guarantee that each task always completes within its deadline.

Section 5.1 introduces the main variables of the problem, which do not depend on the scheduling policy adopted on the accelerator. Section 5.2 presents the main constraints regarding task-to-core mapping, priority and accelerated segments. Next, Section 5.3 lists the constraints to bound the WCETs of tasks in the processors, while Section 5.4 presents the constraints to bound the WCRTs of the tasks. Sections 5.5 and 5.6 present the set of variables and the constraints at the accelerator level for RR and NP-FP scheduling policies, respectively. Only one of them needs to be used in an optimization problem instance, depending on the scheduling policy adopted for the accelerator. This gives rise to a modular approach that may be extended to other scheduling policies by just introducing a new set of variables and constraints, while leaving most of the optimization problem unaltered. Finally, Section 5.7 presents different objective functions that can be used in the optimization problem.

In the MILP constraints we often leverage the so-called *big-M* formulation: to this end, we define the symbol M, a very large positive constant (representing infinity).

5.1. Main MILP Variables

We start presenting the main variables needed to describe the problem. Other auxiliary and additional variables are introduced when required.

5.1.1. Boolean Variables

- *Task assignment in core:* For each task $\tau_i \in \Gamma$, and for each core $p_k \in \mathcal{P}$, $\mathrm{TC}_{i,k} \in \{0, 1\}$ is a binary variable set to 1 if τ_i is allocated to p_k; 0 otherwise.

- *Tasks in same core:* For each task pair $\tau_i, \tau_s \in \Gamma$, with $i \neq s$, $\mathrm{SC}_{i,s} \in \{0, 1\}$ is set to 1 if τ_i is allocated on the same core as τ_s; 0 otherwise.

- *Task priority assignment:* For each task $\tau_i \in \Gamma$, for each $q \in \mathbb{N}$, $1 \leq q \leq |\Gamma|$, $\mathrm{TP}_{i,q} \in \{0, 1\}$ is equal to 1 if τ_i is assigned priority q; 0 otherwise.

- *Priority relationship between tasks:* For each task pair $\tau_i, \tau_s \in \Gamma$, with $i \neq s$, $\mathrm{HP}_{i,s} \in \{0, 1\}$ is equal to 1 if task τ_i is assigned a higher priority than τ_s; 0 otherwise.

- *Accelerated segment:* For each task $\tau_i \in \Gamma$, and for each segment $\rho_{i,j} \in \rho_i$, $\mathrm{AS}_{i,j} \in \{0, 1\}$ is set to 1 if and only if $\rho_{i,j}$ is offloaded to the accelerator; 0 otherwise.

- *Selector variable for candidate WCRT of a task:* For each $\tau_i \in \Gamma$, for each $v_{i,g} \in \mathcal{V}_i$, $\mathrm{SV}_{i,g} \in \{0, 1\}$ is a binary variable set to 1 if $v_{i,g}$ is the candidate WCRT bound.

5.1.2. Real and Integer Variables

- *Priority index of a task:* For each task $\tau_i \in \Gamma$, $\mathrm{PR}_i \in \mathbb{N}^{>0}$ encodes the absolute priority of τ_i.

- *WCETs of tasks and segments:* For each task $\tau_i \in \Gamma$, and for each segment $\rho_{i,j} \in \rho_i$, $\mathrm{ET}_i \in \mathbb{R}^{\geq 0}$ and $\mathrm{ES}_{i,j} \in \mathbb{R}^{\geq 0}$ are the WCET of τ_i and $\rho_{i,j}$, respectively, depending on the core where it is allocated and on whether it is accelerated or not.

- *WCET of an interfering task:* For each task pair $\tau_i, \tau_s \in \Gamma$, with $i \neq s$, $\mathrm{EI}_{i,s} \in \mathbb{R}^{\geq 0}$ is equal to the WCET of τ_i (on the core where it is allocated) if it can interfere with τ_s; 0 otherwise.

- *Response time candidate of a task:* For each $\tau_i \in \Gamma$, for each $v_{i,g} \in \mathcal{V}_i$ (see Equation (7)), $\mathrm{RTC}_{i,g} \in \mathbb{R}^{\geq 0}$ is a candidate WCRT bound for τ_i.

- *Response time of a task:* For each task $\tau_i \in \Gamma$, $\mathrm{RT}_i \in \mathbb{R}^{\geq 0}$ is the WCRT bound of τ_i.

5.1.3. Common Variables for Hardware Acceleration

- *Suspension time of a segment:* For each task $\tau_i \in \Gamma$, for each segment $\rho_{i,j} \in \rho_i$, $\mathrm{STS}_{i,j} \in \mathbb{R}^{\geq 0}$ bounds the time spent by $\rho_{i,j}$ on the hardware accelerator.

- *Suspension time of a task:* For each task $\tau_i \in \Gamma$, $\mathrm{ST}_i \in \mathbb{R}^{\geq 0}$ bounds the overall time spent by τ_i on the hardware accelerator.

5.2. Basic Mapping Constraints

First, we enforce each task to be assigned to only one processor, through the variable $\mathrm{TC}_{i,k}$.

Constraint 1 (Task-to-core mapping). *For each $\tau_i \in \Gamma$,*

$$\sum_{p_k \in \mathcal{P}} TC_{i,k} = 1.$$

Proof.
By definition, $TC_{i,k}$ is set to 1 if and only if τ_i is assigned to core p_k. The constraint follows noting that τ_i is allocated to only one processor only if this constraint is imposed. □

It is convenient to introduce some auxiliary variables to cope with the task-to-processor assignment:

- *Tasks in same core p_k:* For each task pair $\tau_i, \tau_s \in \Gamma$, with $i \neq s$, for each core $p_k \in \mathcal{P}$, $SCC_{i,s,k} \in \{0,1\}$ is set to 1 if τ_i and τ_s are both allocated onto p_k; 0 otherwise.

Constraint 2 enforces the definition of variables $SCC_{i,s,k}$ and $SC_{i,s}$, which denote if two tasks are in the same processor.

Constraint 2 (Tasks in the same core). *For each task pair $\tau_i, \tau_s \in \Gamma$, with $i \neq s$, and for each core $p_k \in \mathcal{P}$,*

$$SCC_{i,s,k} \geq 1 - (2 - TC_{i,k} - TC_{s,k}),$$

$$SCC_{i,s,k} \leq TC_{i,k}, \quad SCC_{i,s,k} \leq TC_{s,k}.$$

Then, for each pair of tasks $\tau_i \in \Gamma$, $\tau_s \in \Gamma \setminus \tau_i$:

$$SC_{i,s} = \sum_{p_k \in \mathcal{P}} SCC_{i,s,k}.$$

Proof.
By definition, $SCC_{i,s,k} \in \{0,1\}$ is set to 1 if and only if task τ_i is allocated on the same core $p_k \in \mathcal{P}$ of task τ_s. If $TC_{i,k} = TC_{s,k} = 1$, by substituting in the constraint $SCC_{i,s,k} \geq 1 \wedge SCC_{i,s,k} \leq 1 \Rightarrow SCC_{i,s,k} = 1$ is enforced. If $TC_{i,k} = TC_{s,k} = 0$ or $TC_{i,k} \neq TC_{s,k}$, by substituting in the constraint we get $SCC_{i,s,k} \leq 0$ for at least one of the last two inequalities, while the first inequality enforces either $SCC_{i,s,k} \geq -1$ or $SCC_{i,s,k} \geq 0$. This implies $SCC_{i,s,k} = 0$, proving the first set of constraints. Finally, since due to Constraint 1 each task is assigned to a core, $\sum_{p_k \in \mathcal{P}} SCC_{i,s,k}$ can be either zero or one, and the last equality enforces the definition of variable $SC_{i,s}$. □

Constraint 3 enforces the uniqueness of the priority assignment through the boolean variable $TP_{i,p}$.

Constraint 3 (Uniqueness of the priority). *For each task $\tau_i \in \Gamma$,*

$$\sum_{p \in \{1,\ldots,|\Gamma|\}} TP_{i,p} = 1,$$

and for each priority $p \in \{1, \ldots, |\Gamma|\}$,

$$\sum_{\tau_i \in \Gamma} TP_{i,p} = 1$$

Proof.
By definition, $TP_{i,p}$ is set to 1 if and only if τ_i is assigned to priority p. The constraint follows noting that the uniqueness of the priority holds only if **(i)** each task $\tau_i \in \Gamma$ is assigned to

exactly one priority p ($p \in \{1, \ldots, |\Gamma|\}$), and **(ii)** each priority p ($p \in \{1, \ldots, |\Gamma|\}$) is assigned to exactly one task $\tau_i \in \Gamma$. □

Constraint 4 specifies the value of the integer variable PR_i, encoding the absolute priority of τ_i.

Constraint 4 (Priority index of a task). *For each task $\tau_i \in \Gamma$,*

$$PR_i = \sum_{1 \leq p \leq |\Gamma|} p \cdot TP_{i,p}.$$

Proof.
By definition, PR_i is an integer value encoding the absolute priority of a corresponding task $\tau_i \in \Gamma$. By Constraint 3, $TP_{i,p}$ is set to 1 if τ_i is assigned to priority p, and there is only one task with such a priority. The constraint follows by summing up the priority index multiplied by the boolean variable $TP_{i,p}$, with $p \in \{1, \ldots, |\Gamma|\}$. □

Next, we limit the possible values of $HP_{i,s}$. This is done with the combination of two constraints. With Constraint 5 we enforce that, for each pair of tasks $\tau_i, \tau_s \in \Gamma$, either τ_i has higher priority than τ_s or vice-versa.

Constraint 5 (Priority relationship between tasks). *For each pair of tasks $\tau_i \in \Gamma$, $\tau_s \in \Gamma \setminus \tau_i$,*

$$HP_{i,s} + HP_{s,i} = 1.$$

Proof.
By definition, $HP_{i,s} \in \{0,1\}$ is equal to 1 if task τ_i is assigned to a higher priority than $\tau_s \in \Gamma \setminus \tau_i$. The constraint enforces that either $\tau_i \in \Gamma$ has higher priority than τ_s, or vice versa, by imposing that only one between $HP_{i,s}$ and $HP_{s,i}$ can be set to one. □

Secondly, Constraint 6 enforces the relationship between absolute priorities, encoded by variables PR_i, and relative priorities, encoded by variables $HP_{i,s}$.

Constraint 6 (Relative and absolute priorities). *For each pair of tasks $\tau_i \in \Gamma$, $\tau_s \in \Gamma \setminus \tau_i$,*

$$-HP_{i,s} \cdot M \leq PR_s - PR_i \leq (1 - HP_{i,s}) \cdot M.$$

Proof.

If $HP_{i,s} = 0$, then we have $0 \leq PR_s - PR_i \leq M$. Hence $0 \leq PR_s - PR_i$, and $PR_i < PR_s$ because Constraint 3 enforces the uniqueness of the priority assignment.
If $HP_{i,s} = 1$, then we have $-M \leq PR_s - PR_i \leq 0$. Hence $PR_s - PR_i \leq 0$, and $PR_s < PR_i$. The constraint follows. □

Finally, Constraint 7 specifies that, for each task $\tau_i \in \Gamma$ and for each segment $\rho_{i,j}$ the processing phase can be executed on a CPU only if $\iota_{i,j} \neq \textsf{HWA}$, and a segment is executed

on the accelerator if $\iota_{i,j} = $ HWA, constraining the possible values of $AS_{i,j}$.

Constraint 7 (HW Acceleration). *For each task $\tau_i \in \Gamma$, for each segment $\rho_{i,j} \in \rho_i$, if $\iota_{i,j} = $ CPU, then*

$$AS_{i,j} = 0.$$

If $\iota_{i,j} = $ HWA then

$$AS_{i,j} = 1.$$

Proof.
By definition, $AS_{i,j} \in \{0, 1\}$ is set to 1 if and only if $\rho_{j,i}$ is offloaded to the accelerator, and it is set to 0 otherwise. If $\iota_{i,j} = $ CPU, then no HW-accelerated implementation of $\rho_{i,j}$ is available: hence, $AS_{i,j} = 0$ is enforced. Conversely, if $\iota_{i,j} = $ HWA, only an HW-accelerated implementation of $\rho_{j,i}$ is available, and hence $AS_{i,j} = 1$ is imposed. The constraint follows. $\square$

5.3. Bounding WCETs and CPU Interference

Next, we present a set of constraints to characterize the WCET of the segments and tasks. The following constraints will act only as safe lower bounds for the variables related to WCETs. This approach limits the complexity of the MILP formulation, and is justified by the fact that, whenever the objective function is a minimization involving the response time, the solver chooses the smallest possible value allowed by the constraints, for all the variables that contribute to the response time. Additionally, the WCET values chosen by the solver will also be limited by the necessity of guaranteeing the schedulability of the system (Constraint 12).

Constraint 8 imposes the value of $ES_{i,j}$ to be no smaller than the WCET of the corresponding segment $\rho_{i,j}$ on the core p_k where the MILP solver allocated τ_i, which depends on whether $\rho_{i,j}$ is accelerated or not.

Constraint 8 (WCET of a segment). *For each task $\tau_i \in \Gamma$, for each segment $\rho_{i,j} \in \rho_i$,*

$$ES_{i,j} \geq \sum_{p_k \in \mathcal{P}} E_{i,j}(p_k) \cdot TC_{i,k} - AS_{i,j} \cdot M$$

$$ES_{i,j} \geq \sum_{p_k \in \mathcal{P}} w_{i,j}(p_k) \cdot TC_{i,k} - (1 - AS_{i,j}) \cdot M,$$

where $w_{i,j}(p_k) = O_{i,j}(p_k) + F_{i,j}(p_k)$.

Proof.
Recall that the variable $ES_{i,j}$ denotes the WCET of $\rho_{i,j}$, depending on the core where it is allocated and on whether it is accelerated or not. If $\rho_{i,j}$ is not accelerated, then $AS_{i,j} = 0$, hence $ES_{i,j} \geq \sum_{p_k \in \mathcal{P}} E_{i,j}(p_k) \cdot TC_{i,k}$, while the second inequality does not have effect (i.e., since M is a large constant, it is reduced to $ES_{i,j} \geq -\infty$). By Constraint 1, $TC_{i,k}$ is equal to 1 only for the core p_k where τ_i is allocated, hence the sum $\sum_{p_k \in \mathcal{P}} E_{i,j}(p_k) \cdot TC_{i,k}$ actually consists in only one term,

while the others are set to 0. Such a term is the WCET of τ_i in the core where it is allocated, which consists of only the processing phase, with WCET $E_{i,j}(p_k)$. On the other hand, if $\rho_{i,j}$ is accelerated, $AS_{i,j} = 1$, hence the first inequality has no effect, while the second becomes $ES_{i,j} \geq \sum_{p_k \in \mathcal{P}} w_{i,j}(p_k) \cdot TC_{i,k}$. Since each term in the sum is multiplied by $TC_{i,k}$ as in the previous case, $ES_{i,j}$ is constrained to be greater than or equal to the WCET of the offloading and finalization phases of $\rho_{i,j}$ (i.e., $w_{i,j}(p_k)$), referred to the specific core p_k where the task τ_i is mapped. The constraint follows. $\square$

Constraint 9 enforces the definition of the variables ET_i.

Constraint 9 (WCET of a task). *For each task $\tau_i \in \Gamma$,*

$$ET_i \geq \sum_{\rho_{i,j} \in \rho_i} ES_{i,j}$$

Proof.
The constraint follows by noting that the overall WCET of $\tau_i \in \Gamma$ is the sum of the individual WCETs of each segment $\rho_{i,j} \in \rho_i$. $\square$

Constraint 10 copes with variable $EI_{i,s}$, which bounds the interference of one job of task τ_s on a job of τ_i. The value of $EI_{i,s}$ is constrained to be greater than or equal to ET_s, if $\tau_s \in \Gamma \setminus \tau_i$ can interfere with $\tau_i \in \Gamma$, zero otherwise.

Constraint 10 (CPU Interference). *For each pair of tasks $\tau_i \in \Gamma$, $\tau_s \in \Gamma \setminus \tau_i$,*

$$EI_{i,s} \geq ET_i - M \cdot (2 - HP_{i,s} - SC_{i,s})$$

Proof.
Under a partitioned fixed-priority scheme, a task τ_i may interfere with another task τ_s if and only if: **(i)** it is allocated in the same core, and **(ii)** it has higher priority than τ_s. Condition (i) is verified when $SC_{i,s} = 1$ (Constraint 2), whereas condition (ii) holds when $HP_{i,s} = 1$ (Constraint 6). The constraint follows by noting that $EI_{i,s} \geq ET_i$ is enforced if and only if $SC_{i,s} = 1$ and $HP_{i,s} = 1$. Otherwise, the constraint has no effect ($EI_{i,s} \geq -\infty$). $\square$

5.4. Bounding WCRTs

As discussed in Section 4.2.2, the processor demand constraint that uses the set of checkpoints in Theorem 1 is a suitable way to bound the WCRTs in a linear optimization problem. This however requires the knowledge of the release jitter for each interfering self-suspending task. The choice of a jitter $J_h = D_h - C_h$ for τ_h is a safe bound for a self-suspending task. However, such jitter still depends on the WCET of the task. In particular, due to the heterogeneity of the platform, the WCET depends on **(i)** the type of the processor where it is allocated, and **(ii)** whether it is accelerated. This way, it would be an explicit function of $TC_{i,k}$ and

$AS_{i,j}$. As a consequence, we would be required to introduce additional variables to compute the floor term of Equation (7) and the ceiling term of Equation (8). Additionally, the ceiling term of Equation (8) is multiplied by the WCET of each interfering task, which is itself a variable (i.e., $EI_{j,i}$), thus making the problem not linear.

To address this problem, we consider a more conservative (but linear) approach. Since the response-time bound is monotonic non-decreasing with respect to the jitter bound, which in turn is monotonic non-increasing with the WCET of the interfering task, by increasing the jitter of the interfering tasks we obtain a more conservative estimate of the WCRT of the task under analysis. Hence, taking the minimum WCET over all possible configurations of processor type and acceleration state of each segment yields a safe bound on the jitter. To this end, we introduce the *constant* term C_i^{MIN} for a task τ_i to denote the *minimum* WCET with which a task can be characterized, among all possible configurations of cores $p_k \in \mathcal{P}$ and all the possible combinations of accelerated (or not) segments $\rho_{i,j} \in \rho_i$. Additionally, for each task $\tau_i \in \Gamma$, since the priority level is a variable of the optimization problem, we do not know in advance which tasks will have higher priority than τ_i, thus we consider the (eventual) checkpoint associated to each task in the set. Note that extending the approach of Section 4.2.2 to all tasks in Γ (compared with only the ones that have higher priority than τ_i) has the only effect of possibly introducing additional checkpoints to the schedulability test of τ_i, which the solver is required to check.

By considering all tasks $\tau_s \in \Gamma \setminus \tau_i$ as potentially having higher priority than τ_i, the jitters used in Section 4.2.1 can be refined as follows:

$$\overline{\mathcal{J}}_i = \bigcup_{\tau_s \in \Gamma \setminus \tau_i} \{J_s\}, \tag{12}$$

with

$$J_s = \begin{cases} D_s - C_s^{\mathrm{MIN}} & \text{if } \rho_s^H \neq \emptyset \\ 0 & \text{otherwise,} \end{cases} \tag{13}$$

which is used to build the set of checkpoints $\mathcal{V}_i(\overline{\mathcal{J}}_i)$ as:

$$\mathcal{V}_i(\overline{\mathcal{J}}_i) = \left\{ \bigcup_{\tau_s \in \Gamma \setminus \tau_i} V_{i,s} \right\} \cup \{D_i\}, \tag{14}$$

where $V_{i,s}$ is defined as in Theorem 1.

Constraint 11 establishes a response-time bound for each task $\tau_i \in \Gamma$ with the method presented in Section 4.2.2, and considering the jitter set computed as in Equation (12).

Constraint 11 (WCRT bound candidate). *For each task $\tau_i \in \Gamma$, and for each $v_{i,g} \in \mathcal{V}_i(\overline{\mathcal{J}}_i)$ obtained with Equation (14) using the jitter set of Equation (12),*

$$RTC_{i,g} \geq ET_i + ST_i + \sum_{\tau_s \in \Gamma \setminus \tau_i} \left\lceil \frac{v_{i,g} + J_s}{T_s} \right\rceil EI_{s,i},$$

$$RTC_{i,g} \leq v_{i,g} + (1 - SV_{i,g}) \cdot M,$$

$$RT_i \geq RTC_{i,g} - (1 - SV_{i,g}) \cdot M.$$

Additionally, for each task $\tau_i \in \Gamma$,

$$\sum_{v_{i,g} \in \mathcal{V}_i(\overline{\mathcal{J}}_i)} SV_{i,g} = 1.$$

Proof.
For each $\tau_i \in \Gamma$, there exists up to $|\mathcal{V}_i|$ candidate response-time bounds, represented with variables $RTC_{i,g}$, where the index g corresponds to the index of $v_{i,g} \in \mathcal{V}_i(\overline{\mathcal{J}}_i)$.

For each $v_{i,g} \in \mathcal{V}_i(\overline{\mathcal{J}}_i)$, the first inequality implements the response-time bound of Equation (2) bounding the jitter as $J_s = D_s - C_s^{\mathrm{MIN}}$. The second inequality enforces that the selected response-time bound must be valid according to Equation (8). In all other cases, the inequality is disabled (i.e., $RTC_{i,g} \leq \infty$). The third inequality enforces that the response-time bound is greater than or equal to the selected response-time candidate: otherwise, it is disabled (i.e., $RTC_{i,g} \geq -\infty$). Finally, the fourth inequality enforces only one of the checkpoints to be selected as the actual response-time bound. □

Finally, Constraint 12 enforces RT_i to be a valid response-time bound.

Constraint 12 (Schedulability). *For each task $\tau_i \in \Gamma$,*

$$RT_i \leq D_i.$$

Next, we show how to bound the suspension time spent in the hardware accelerator. Clearly, this depends on the specific scheduling policy implemented by the accelerator. We present the cases in which such policies are round-robin and non-preemptive fixed priority, as a set of constraints that need to be added to the optimization problem *only when the scheduling policy is used*. In this way, we highlight the modular nature of our approach, which can easily be extended to other scheduling policies by just implementing some new constraints, while most of the optimization problem can be left unaltered.

5.5. Worst-Case Suspension Time with RR

This section presents the constraints to bound the duration of accelerated segment (and hence of the suspension time of the task running on the core) under round-robin scheduling.

Before proceeding, it is necessary to introduce additional variables to handle the RR scheduling.

5.5.1. Additional Variable for RR Scheduling
- *Longest accelerated segment:* For each task $\tau_i \in \Gamma$, $LA_i \in \mathbb{R}^{\geq 0}$ bounds the longest WCET of an accelerated segment $\rho_{i,j} \in \rho_i^H$ of task τ_i.

5.5.2. Constraints
First, Constraint 13 bounds the length of the longest accelerated segment of each task $\tau_i \in \Gamma$, enforcing the definition of LA_i.

Constraint 13 (Longest accelerated segment). *For each task* $\tau_i \in \Gamma$, *and for each segment* $\rho_{i,j} \in \rho_i^H$,

$$LA_i \geq E_{i,j}(\mathcal{H}) - (1 - AS_{i,j}) \cdot M$$

Proof.
When a segment $\rho_{i,j} \in \rho_i^H$ is accelerated, $E_{i,j}(\mathcal{H})$ bounds its WCET. The constraint follows by noting that if $\rho_{i,j}$ is accelerated, $AS_{i,j} = 1$ and $LA_i \geq E_{i,j}(\mathcal{H})$ is imposed; otherwise, $AS_{i,j} = 0$ and the constraint does not take effect (i.e. $LA_i \geq -\infty$). □

Constraint 14 bounds the time spent on the hardware accelerator by each segment $\rho_{i,j} \in \rho_i^H$.

Constraint 14 (Suspension time of a segment). *For each task* $\tau_i \in \Gamma$, *and for each segment* $\rho_{i,j} \in \rho_i^H$,

$$STS_{i,j} \geq E_{i,j}(\mathcal{H}) + \left(\sum_{\tau_j \in \Gamma \setminus \tau_i} LA_j \right) - (1 - AS_{i,j}) \cdot M$$

Proof.
The constraint follows from Lemma 1. □

Constraint 15 bounds the time spent on the hardware accelerator by each task $\tau_i \in \Gamma$.

Constraint 15 (Suspension time of a task). *For each task* $\tau_i \in \Gamma$ *such that* $\rho_i^H \neq \emptyset$,

$$ST_i \geq \sum_{\rho_{i,j} \in \rho_i} STS_{i,j}$$

Proof.
The constraint follows by noting that the overall time spent on the hardware accelerator by each task τ_i is bounded by the time spent on the hardware accelerator by each segment $\rho_{i,j} \in \rho_i$, encoded by variables $STS_{i,j}$. □

5.6. Worst-Case Suspension Time with NP-FP
This section presents the constraints to bound the time globally spent on the hardware accelerator, and the overall suspension-time experienced by the tasks running on the cores under non-preemptive fixed priority scheduling.

Before proceeding, it is necessary to introduce additional variables to handle the NP-FP scheduling.

5.6.1. Additional Variables for NP-FP Scheduling
- *WCET of an interfering segment on $\mathcal{H}$:* For each task $\tau_i \in \Gamma$, for each segment $\rho_{i,j} \in \rho_i^H$, for each task $\tau_s \in \Gamma \setminus \tau_i$, $EHR_{i,j,s} \in \mathbb{R}^{\geq 0}$ is equal to the WCET of $\rho_{i,j}$ (on $\mathcal{H}$) if τ_i can interfere with τ_s; it is 0 otherwise.

- *WCET of an interfering task on $\mathcal{H}$:* For each task pair $\tau_i, \tau_s \in \Gamma$, with $i \neq s$, $EH_{i,s} \in \mathbb{R}^{\geq 0}$ is equal to the WCET of τ_i (on $\mathcal{H}$) if it can interfere with τ_s; 0 otherwise.

- *NP Blocking time of a task on $\mathcal{H}$:* For each task $\tau_i \in \Gamma$, $BL_i \in \mathbb{R}^{\geq 0}$ bounds the blocking due to lower-priority tasks that any segment of τ_i can experience when accelerated on $\mathcal{H}$ due to non-preemptive scheduling.

- *Suspension time candidate of a segment:* For each $\tau_i \in \Gamma$, for each segment $\rho_{i,j} \in \rho_i^H$, for each $v_{i,g} \in \mathcal{V}_i$, $STC_{i,j,g} \in \mathbb{R}^{\geq 0}$ is a candidate suspension time bound for τ_i.

- *Selector variable for candidate suspension time:* For each $\tau_i \in \Gamma$, for each segment $\rho_{i,j} \in \rho_i^H$, for each $v_{i,g} \in \mathcal{V}_i$, $SS_{i,j,g} \in \{0, 1\}$ is binary variable set to 1 if $v_{i,g}$ is the candidate suspension time bound chosen by the solver.

5.6.2. Constraints
As discussed in Section 4.3.2, the suspension time can be bounded by leveraging Lemma 2 and Equation (11). Equation (11) requires bounding the blocking from lower-priority tasks and the interference from high-priority tasks. Constraint 16 bounds the former, enforcing the definition of variable BL_i.

Constraint 16 (NP blocking time). *For each task* $\tau_i \in \Gamma$ *such that* $\rho_i^H \neq \emptyset$, *for each task* $\tau_s \in \Gamma \setminus \tau_i$, *for each segment* $\rho_{s,f} \in \rho_s^H$,

$$BL_i \geq E_{s,f}(\mathcal{H}) - (2 - AS_{s,f} - HP_{i,s}) \cdot M.$$

Proof.
The variables BL_i bound the blocking time due to lower-priority tasks that any segment of τ_i can experience when accelerated on $\mathcal{H}$ due to non-preemptive scheduling. Following Lemma 2, such blocking is bounded by

$$B_i = \max \left\{ E_{l,v}(\mathcal{H}) \mid \tau_l \in lp(\tau_i) \wedge \rho_{l,v} \in \rho_l^A \right\}.$$

The constraint is enforced by requiring BL_i to be greater than or equal to the WCET (on the accelerator) of all the segments that **(i)** are accelerated (i.e., $AS_{s,f} = 1$), and **(ii)** are parts of a task τ_s with a lower priority than τ_i (i.e., $HP_{i,s} = 1$). In all the other cases, the constraint is disabled ($BL_i \geq -\infty$). □

Constraint 17 enforces the definition of the variables $EH_{i,s}$, which represent the WCET of τ_i on $\mathcal{H}$, if τ_i can interfere with τ_s.

Constraint 17 (WCET of an interfering task on $\mathcal{H}$). *For each task* $\tau_i \in \Gamma$, *for each segment* $\rho_{i,j} \in \rho_i^H$, *for each task* $\tau_s \in \Gamma \setminus \tau_i$ *such that* $\rho_s^H \neq \emptyset$,

$$EHR_{i,j,s} \geq E_{i,j}(\mathcal{H}) - (2 - AS_{i,j} - HP_{i,s}) \cdot M,$$

and for each $\tau_i \in \Gamma$, $\tau_s \in \Gamma \setminus \tau_i$,

$$EH_{i,s} \geq \sum_{\rho_{i,j} \in \rho_i^H} EHR_{i,j,s}$$

Proof.
The interfering WCET due to τ_i on a lower priority task τ_s is the sum of the individual contributions due to all segments $\rho_{i,j} \in \rho_i$. Each segment contributes to the interference if: **(i)** $\rho_{i,j}$ is accelerated ($AS_{i,j} = 1$), and **(ii)** τ_i has higher priority than τ_s ($HP_{i,s} = 1$). The constraint follows. □

Constraint 18 bounds the overall suspension time using Equation (11). This constraint follows alike to Constraint 11, but, in this case, we consider only those tasks that have at least one segment that can be accelerated as potential interfering tasks. The jitter component of Equation (11) depends on the WCET of the task on the accelerator where it runs. Since the response-time bound is monotonic non-decreasing with respect to the jitter bound, and it is monotonic non-increasing with the WCET of the interfering task, we consider the minimum WCET $C_i^{\text{MIN-HW}}$ with which a task τ_i can be characterized on the accelerator when $\rho_i^H \neq \emptyset$, when at least one of the segments in ρ_i^H is accelerated. It is defined as

$$C_i^{\text{MIN-HW}} = \begin{cases} \min_{\rho_{i,j} \in \rho_i^H} E_{i,j}(\mathcal{H}) & \text{if } \rho_i^W = \emptyset \\ \sum_{\rho_{i,j} \in \rho_i^W} E_{i,j}(\mathcal{H}) & \text{otherwise,} \end{cases}$$

where $\rho_i^W \subseteq \rho_i^H$ is the set of all segments of task τ_i that are *necessarily* accelerated because they have only an implementation on $\mathcal{H}$ (i.e., $\iota_{i,j} = \text{HWA}$). This minimum WCET is then used to build the initial release jitters for such tasks τ_s with $\tau_s \in \Gamma \setminus \tau_i$, such that $\rho_s^H \neq \emptyset$, as follows:

$$J_s = D_s - C_s^{\text{MIN-HW}}. \tag{15}$$

The corresponding checkpoint $V_{i,s}$ can then be computed again as in Theorem 1. For notation purposes we define the set of jitters at the hardware accelerator, for the tasks that can be accelerated, as

$$\overline{\mathcal{J}}_i^A = \bigcup_{\tau_s \in \Gamma \setminus \tau_i \mid \rho_s^H \neq \emptyset} \left\{ D_s - C_s^{\text{MIN-HW}} \right\}, \tag{16}$$

and the corresponding set of checkpoints as $\mathcal{V}_i(\overline{\mathcal{J}}_i^A)$.

Constraint 18 (Suspension bound candidate). *For each task $\tau_i \in \Gamma$, for each segment $\rho_{i,j} \in \rho_i^H$, for each $v_{i,g} \in \mathcal{V}_i(\overline{\mathcal{J}}_i^A)$, (see Eq. (11))*

$$STC_{i,j,g} \geq BL_i + \sum_{\substack{\tau_s \in \Gamma \setminus \tau_i \\ \rho_s^H \neq \emptyset}} \left\lceil \frac{v_{i,g} + J_s}{T_s} \right\rceil \cdot EH_{s,i},$$

$$STC_{i,j,g} \leq v_{i,g} + (1 - SS_{i,j,g}) \cdot M,$$

$$STS_{i,j} \geq E_{i,j}(\mathcal{H}) + STC_{i,j,g} - (1 - SS_{i,j,g}) \cdot M,$$

For each task $\tau_i \in \Gamma$, for each segment $\rho_{i,j} \in \rho_i^H$

$$\sum_{v_{i,g} \in \mathcal{V}_i(\overline{\mathcal{J}}_i^A)} SS_{i,j,g} = AS_{i,j}.$$

Proof.
This constraint follows similarly to Constraint 11. For each $\tau_i \in \Gamma$ and for each segment $\rho_{i,j} \in \rho_i^H$, there are up to $|\mathcal{V}_i|$ candidate suspension-time bounds, represented with variables $STC_{i,j,g}$ with $v_{i,g} \in \mathcal{V}_i(\overline{\mathcal{J}}_i^A)$. The fourth equality enforces only one of them to be selected as the actual suspension-time bound if the segment is accelerated. For each $v_{i,g} \in \mathcal{V}_i(\overline{\mathcal{J}}_i^A)$, the first inequality encodes Equation (11) bounding the jitter as $J_s = D_s - C_s^{\text{MIN-HW}}$. The second inequality enforces that the selected suspension-time bound must be valid according to Equation (11) (see Section 4.3.2). In all other cases, the inequality is disabled (i.e., $STC_{i,j,g} \leq \infty$). The third inequality enforces that the suspension-time bound is greater than or equal to the selected suspension-time candidate: otherwise, it is disabled (i.e., $STC_{i,j,g} \geq -\infty$). □

Finally, Constraint 19 must be enforced to bound the overall time spent on the hardware accelerator by each task $\tau_i \in \Gamma$ and therefore the overall suspension time.

Constraint 19 (Suspension time of a task). *For each task $\tau_i \in \Gamma$, such that $\rho_i^H \neq \emptyset$,*

$$ST_i \geq \sum_{\rho_{i,j} \in \rho_i^H} STS_{i,j}.$$

Conversely, for each task $\tau_i \in \Gamma$, such that $\rho_i^H = \emptyset$,

$$ST_i = 0$$

5.7. Objective Function
We design our optimization problem with four possible (alternative) objective functions. Two of them are devised to optimize the end-to-end latencies of individual chains, while the others consider the ratio between WCRT bounds and deadlines of individual tasks.

For those targeting end-to-end latencies, we introduce, for each $\omega_x \in \Omega$, the variables $L_x \in \mathbb{R}^{\geq 0}$ to encode the latency of chain ω_x, in accordance with Equation (1).[2]

Min-Max Latency The first objective function encodes the goal of minimizing the maximum latency of all chains $\omega_x \in \Omega$. It is defined as:

$$\textbf{minimize } L_{\text{MAX}}, \tag{17}$$

where $L_{\text{MAX}} \in \mathbb{R}^{\geq 0}$ is a real variable that encodes the maximum latency of all chains $\omega_x \in \Omega$ by enforcing the constraint $L_{\text{MAX}} \geq L_x, \forall \omega_x \in \Omega$.

Min-Sum Latency The second objective function encodes the goal of minimizing the sum of the latency due to all the chains $\omega_x \in \Omega$. It is defined as:

$$\textbf{minimize } \sum_{\omega_x \in \Omega} L_x \tag{18}$$

---

[2]Enforcing $L_x \geq \sum_{\tau_i \in \omega_x} (R_i + T_i) - T_{first}$ if sufficient because all objective functions are minimizations.

Table 2
Parameters and solutions of the optimization problem for the task set provided with the challenge model. All times are in milliseconds.

ID	Name	T_i (ms)	A57 C_i^{NA}	A57 C_i^A	Denver C_i^{NA}	Denver C_i^A	GPU C_i	Min-Max Lat. RR Sol. PRIO	CPU	ACC
1	Lidar Grabber	33	14,379	-	10,868	-	-	8	5	NO
2	DASM	5	1,958	-	1,3	-	-	5	1	NO
3	CAN Polling	10	0,632	-	0,6	-	-	1	1	NO
4	EKF	15	5,011	-	4,430	-	-	3	0	NO
5	Planner	15	13,939	-	12,437	-	-	2	2	NO
6	SFM	33	31,055	8,320	27,812	6,711	7,900	4	3	NO
7	Localization	400	407,811	18,568	294,808	14,516	124,000	7	4	NO
8	Lane Detection	66	53,732	8,667	42,238	7,626	27,333	6	5	NO
9	Detection	200	-	4,958	-	4,086	116,000	0	0	YES

Table 3
Running times.

	Min-Max Lat Time	Min-Max Lat Opt.	Min-Sum Lat Time	Min-Sum Lat Opt.	Min-Max RT Time	Min-Max RT Opt.	Min-Sum RT Time	Min-Sum RT Opt.
No Contention	4.13s	YES	1h	2.41%	0.94s	YES	1h	11.81%
Round-Robin	13.05s	YES	1h	0.18%	2.11s	YES	1h	1.68%
NP-FP	2.98s	YES	26.2m	YES	1.45s	YES	1h	0.02%

Min-Max WCRT-ratio The third objective function encodes the goal of minimizing the maximum ratio between the WCRT bound and the deadline of all tasks $\tau_i \in \Gamma$. It is defined as:

$$\text{minimize } \text{RT}_{\text{MAX}}, \quad (19)$$

where $\text{RT}_{\text{MAX}} \in \mathbb{R}^{\geq 0}$ is a real variable that encodes the maximum ratios between the WCRT bound and the deadline of all tasks $\tau_i \in \Gamma$ by enforcing the constraint $\text{RT}_{\text{MAX}} \geq \frac{\text{RT}_i}{D_i}$, $\forall \tau_i \in \Gamma$.

Min-Sum WCRT-ratio The last objective function encodes the goal of minimizing the sum of the ratios between the WCRT bound and the deadline of all tasks $\tau_i \in \Gamma$. It is defined as:

$$\text{minimize } \sum_{\tau_i \in \Gamma} \frac{\text{RT}_i}{D_i}. \quad (20)$$

6. Evaluation

We evaluate the proposed optimization method on the WATERS 2019 Challenge by Bosch [37]. As discussed in Section 2, the WATERS 2019 Challenge Platform Model is based on the NVIDIA Jetson TX2. This heterogeneous platform is composed of a quad-core 1.9GHz ARMv8 A57, a dual-core 2GHz ARMv8 Denver, and an integrated GPU.

Consequently, the proposed platform model comprises two types of processor cores: the first four cores $p_k \in \mathcal{P}$, with $k \in \{0, 1, 2, 3\}$ are the *A57* cores, while the last two, i.e., $p_k \in \mathcal{P}$, with $k \in \{4, 5\}$ are the *Denver* cores.

Table 2 reports the attributes of the WATERS 2019 task set. Each task includes a single segment. Tasks SFM, Localiza-

tion, and Lane Detection are provided with both a fully CPU-based implementation and a GPU-based implementation, while Detection is provided only with an accelerated implementation. All others tasks need to run on CPU cores. For each type of processor, Table 2 reports C_i^{NA} and C_i^A to denote the overall WCET of the task when it is running solely on a core or using the accelerator, respectively. Similarly, the column labeled with C_i under the GPU group reports the overall tasks' WCET when executing on the GPU, if the task is accelerated. The period (equal to the relative deadline) is also reported. All times are in milliseconds.

In such a scenario, our MILP formulation determines: **(i)** which tasks to accelerate on the GPU, **(ii)** which tasks to execute on the (faster) Denver cores, and which to execute on A57 cores, **(iii)** the task-to-core assignment, **(iv)** the priority assignment. The corresponding solution needs to guarantee the application timing constraints, i.e., each task needs to complete within its deadline.

The proposed MILP formulation has been coded in C++ and solved with IBM CPLEX on a machine equipped with an Intel Core i7-6700K @ 4.00GHz.

We compared the four different objective functions discussed in Section 5.7. For each of them, we considered three different policy for the hardware accelerator: **(i)** round-robin, **(ii)** non-preemptive fixed-priority and **(iii)** no contention, where each accelerated task runs in the accelerator without interference. The third case serves both for baseline comparison and it can give some intuitions on the results that can be obtained by the optimization problem for hardware accelerators that do not provide time interference (e.g., dedicated, statically-deployed FPGA-based accelerators where interference-related delays are only due to contention for

memories, interconnects, and devices[3]).

Figure 7 reports the ratio between: **(i)** the obtained WCRT and the deadline (RT/D) and, **(ii)** the suspension-time bound and the deadline (ST/D) for such configurations. For the configuration of Figure 7(a) the last three columns of Table 2 show the priority assigned by the solver, the task-to-core assignment, and whether the task has been selected for being accelerated. For all the objective functions, the solver accelerates only the Detection task both for round-robin and non-preemptive fixed priorities as it deems more convenient to run the other three accelerable tasks on a CPU core rather than congesting the GPU, while in the "no contention" case all tasks are accelerated due to the shorter WCETs they experience by running on the GPU. By comparing inset (a) with (b), and (c) with (d), we note that objective functions minimizing the maximum RT/D and chain latency tend to provide higher ratios, as in this case, the solver needs to optimize only the task leading to the maximum RT/D ratio, while objective functions minimizing the sum of RT/D and chain latencies generally provide lower values of the RT/D ratio. For example, this is the case of the CAN task, which achieves an RT/D of 1 in insets (a) and (c), and a very small RT/D in insets (b) and (d). On the other hand, the Min-Sum objectives are more difficult problems to solve: indeed, they manifested also higher running times. Table 3 shows the running times achieved by running CPLEX with a timeout of 1 hour. For the Min-Sum objectives, the solver found the optimal solution within the allowed time in only one case. However, in almost all cases, the solver provides a solution very close to optimality, with an optimality gap below 3%. Only for the Min-Sum RT objective in the "no contention" case, the optimality gap after 1-hour running is 11.81%: this is attributed to the fact that in the no contention case, there are fewer constraints to impose, and thus a larger search space.

Figure 8 reports another interesting configuration we found in our experimentation. In this case, the WATERS 2019 WCETs have been scaled to 80% of their values: this leads the optimizer to accelerate also the Localization task with the round-robin policy. Indeed, with smaller WCETs, the interference imposed by Localization on Detection becomes smaller, making it convenient to accelerate also Localization, which has a way smaller WCET when running on the accelerator (see Table 2). Therefore, the proposed optimization problem allows recognizing non-trivial trade-offs, allowing to obtain optimal solutions that would be very hard to grasp without relying on real-time analysis.

Finally, Table 4 reports the chain latencies obtained with the four different objective functions. The second column of the table reports an ordered list of task IDs (introduced in Table 2) representing the tasks in the chain. As expected, the longest chains are those involving the Localization task, which has a large period (see Equation (1)). From the results, we can observe that chain-specific objective functions may lead to greatly smaller latency values: for example, this is

the case of chain C4, which achieves a latency of about 760 ms with the Min-Max Lat objective and much higher latency of about 848 ms with the Min-Max RT.

7. Related Work

Heterogeneous platforms have received great attention in the last years, especially in the field of *high-performance parallel computing*. The primary focus of researchers in this field has been dedicated to improving performance (e.g., by pursuing faster computation) and reducing energy consumption, mainly regarding mainstream computing, but with little attention to real-time constraints or embedded systems. A wide survey on this topic of partitioning techniques and benchmarks regarding combined CPU/GPU architectures can be found in [52]. Among the many, it is worth mentioning the work of Li et al. [44], which presents a set of algorithms for task mapping in heterogeneous platforms with GPUs, with the goal of minimizing the makespan.

The introduction of heterogeneous platforms and hardware accelerators is more recent for what concerns embedded real-time applications. While most of the available works in real-time literature involving multicore platforms are often constrained to *homogeneous* processors [28], there is a significant portion of literature that started exploring the potential and challenges of heterogeneous platforms applied to the case of embedded real-time applications.

Concerning the study of specific accelerators in real-time systems, prior works mostly targeted GPU-based and FPGA-based acceleration. Several efforts have been spent on improving the predictability of workloads running on GPU accelerators. Since the scheduling policies implemented in such accelerators are typically not disclosed by hardware vendors, a branch of research investigates their internal behavior [2, 3, 54], e.g., through reverse engineering. Capodieci et al. [14], Ali and Yun [1], and Forsberg et al. [33] proposed mechanisms to achieve control on the memory traffic on platforms with GPUs. Cavicchioli et al. [21] studied novel methods to offload computations to a GPU. Other research provided an implementation of the constant bandwidth server on an NVIDIA GPU [13].

Concerning FPGA-based accelerators, Biondi et al. [9] introduced the FRED framework to support real-time applications on heterogeneous platforms. FRED provides a predictable infrastructure and a corresponding analysis to guarantee bounded delays when requesting a dynamically reconfigured hardware accelerator. Later, Pagani et al. [55] provided an implementation for FRED based on Linux. Other works aimed at improving the predictability of bus accesses of hardware-accelerated tasks on FPGA-based platforms and evaluating the profitability of performing acceleration requests [56, 63–65, 69, 70].

Considering one of the main problem addressed in this work, i.e., *task partitioning* onto heterogeneous platforms, in general, most of the available literature on the topic is limited to heterogeneous platforms composed of *CPUs only*. The problem of partitioning real-time applications onto platforms with heterogeneous processing cores is known to be NP-hard
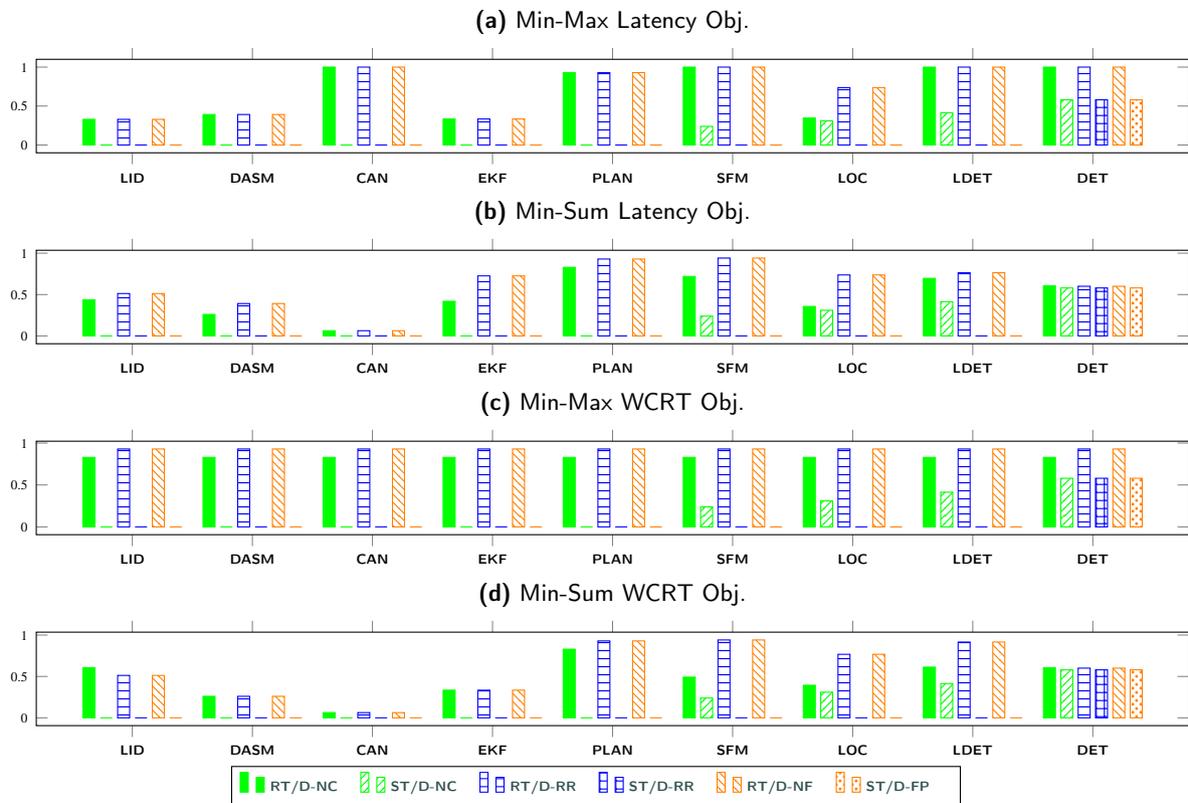
---

[3]Note that performing an optimization for FPGA-based accelerators requires additional considerations about the physical resource requirements on the programmable logic [67].

Figure 7: Ratio between the obtained WCRT bounds and the deadline, and the suspension time bounds and the deadline for each task of the WATERS 2019 Challenge obtained by running the MILP with different objective functions reported above each graph.
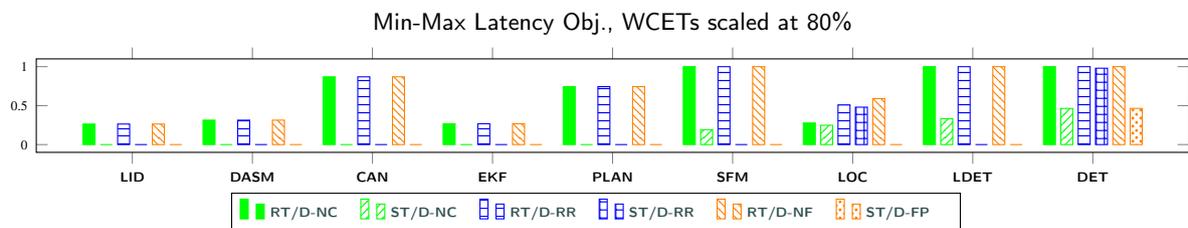

Figure 8: Ratio between the obtained WCRT bounds and the deadline, and the suspension time bounds and the deadline for each task of the WATERS 2019 Challenge obtained by running the MILP with the Min-Max Latency objective function and WCET scaled at the 80% of their values.

in the strong sense [5]. This problem has been mainly addressed with the usage of integer linear programming (ILP) techniques [5–7]. Additionally, heuristics methods have been proposed to solve this problem, e.g., those based on an *ant-colony optimization* approach [22] and an improved *particle swarm optimization* [62]. However, none of these methods considers the presence of hardware accelerators, which introduces considerable challenges, e.g., a suspension behavior on the processing cores and deciding whether it is convenient (for a WCRT perspective) to perform acceleration.

In the context of platforms with heterogeneous cores, a popular target platform is the big.LITTLE from ARM [42], which is composed of a "little" and power-efficient set of cores, together with a "big" set of cores for high-performance

computation. Partitioning heuristics exist for such a specific architecture, e.g., see [49–51, 73]. In the works mentioned above, schedulability is checked by means of utilization-based tests.

The problem of partitioning applications to multicore platforms also requires that the functional dependencies are preserved in the final deployment. Such dependencies are commonly coded in the form of precedence constraints between tasks, constituting a direct acyclic graph (DAG). The problem of partitioning DAGs on multicores while preserving their functional dependencies has been addressed, e.g., in [12], and it also drew much attention recently in the automotive field [39, 47]. Time determinism and causality in multicore platforms can be effectively addressed with the

Table 4
Latencies (in ms) of the processing chains with the hardware accelerator adopting the NP-FP policy.

ID	Tasks	Min-Max Lat	Min-Sum Lat	Min-Max RT	Min-Sum RT
C1	9 - 5 - 2	235,897	155,983	224,438	155,325
C2	6 - 5 - 2	68,897	66,952	69,251	66,294
C3	8 - 5 - 2	101,897	86,307	99,916	95,677
C4	3 - 7 - 4 - 5 - 2	760,716	757,222	848,523	762,948
C5	1 - 7 - 4 - 5 - 2	761,584	773,497	869,896	779,223
C6	1 - 5 - 2	46,76	52,804	69,251	52,146
C7	3 - 4 - 5 - 2	65,908	62,414	76,817	55,882
C8	3 - 5 - 2	45,897	36,529	47,878	35,871

usage of Logical Execution Time Paradigm [38]. Its practical application to multicore has been explored in [10, 11, 60], and specifically for addressing the problem of partitioning real-time applications on a (homogeneous) multicore in [58]. The work in [36] proposes a mapping of multiple DAG tasks that also minimizes power consumption, without considering hardware accelerators. A response time analysis for a DAG task on a heterogeneous platform with a single hardware accelerator is presented in [66]. Only applications performing a single acceleration request for the whole application (composed of a single DAG task) are supported. Zahaf et al. [40] presented the HPC-DAG model and corresponding schedulability analysis, specifically considering NVIDIA platforms.

The introduction of shared devices in the platform, such as GPUs and FPGAs, brings additional complexity in the analysis problem. Indeed, tasks accessing shared resources or performing operations on external devices are subject to *suspension delays*, which must be properly accounted for to ensure schedulability [25]. Many analyses have been then devised on the topic, with different goals, e.g., supporting global scheduling techniques [29, 46], analyzing EDF scheduling [35], providing a unifying analysis [24], analyzing parallel tasks [18, 32], or supporting soft real-time tasks [45]. A comprehensive review of works about self-suspensions can be found in the two excellent surveys by Chen et [23, 25]. A set of task partitioning algorithms in the presence of a shared resource, with the goal of guaranteeing schedulability while minimizing the required size of the shared resource, is presented in [30]. No priority assignment is provided, and only homogeneous cores are considered.

Introducing self-suspensions also brings additional complexity in the formulation of the task mapping problem. For instance, the response time formulation must be properly adapted in order to be used in a linear optimization environment. In this work, we make use of an approximated analysis based on checking only a smart subset of time instants, following the work in [59]. Other works addressed this problem with ad-hoc formulations for the response time. An example of the ILP method that addresses optimal partitioning in the case of shared resources, together with a response time analysis (but on a *homogeneous* multicore platform), is presented in [71]. A heuristic for resource-oriented task partitioning in multicores is presented in [41]. Other partitioning strategies

are proposed in [26]. All these works, however, target only homogeneous multicores.

To the best of our knowledge, there is no prior work targeting the problem of providing an optimal solution for the partitioning and priority assignment of real-time tasks on heterogeneous platforms that also include an hardware accelerator.

8. Conclusions

This paper provides solutions for modeling, analyzing, and partitioning real-time applications running onto heterogeneous platforms equipped with a hardware accelerator. We presented a task and platform model that is applicable to different use-cases, together with a response time analysis that supports different scheduling policies at the hardware-accelerator level, and which can be easily linearized to be used in an ad-hoc optimization problem. The proposed approach is a simple and flexible way for finding an optimal task-to-core mapping and priority assignment for real-time autonomous applications, and for deciding whether to accelerate tasks provided with both CPU-based and accelerated implementations under different scheduling policies for the accelerator. Nevertheless, there is plenty of space for future work in a flourishing research field as the one of heterogeneous real-time systems. For example, interesting future research directions include the consideration of more precise analyses of self-suspending tasks [53], the extension to asynchronous hardware acceleration [4], ROS-based systems [19] and frameworks for deep neural networks [15, 16, 61, 68], and the study of specific hardware accelerators [54] and acceleration methods [9] to provide fine-grained bounds on the suspension time.

References

[1] Ali, W., Yun, H., Jul 3 - 6, 2018. Protecting real-time GPU kernels on integrated CPU-GPU SoC platforms, in: 30th Euromicro Conference on Real-Time Systems (ECRTS 2018), Barcelona, Spain.

[2] Amert, T., Anderson, J., 2021. CUPiD RT: Detecting improper GPU usage in real-time applications, in: 2021 IEEE 24rd International Symposium on Real-Time Distributed Computing (ISORC).

[3] Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D., Dec 5-8, 2017. GPU scheduling on the NVIDIA TX2: Hidden details revealed, in: Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS 2017), Paris, France.

[4] Aromolo, F., Biondi, A., Nelissen, G., Buttazzo, G., 2021. Event-driven delay-induced tasks: Model, analysis, and applications, in: 2021 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).

[5] Baruah, S.K., 2004. Partitioning real-time tasks among heterogeneous multiprocessors, in: International Conference on Parallel Processing, 2004. ICPP 2004., IEEE. pp. 467–474.

[6] Baruah, S.K., Bonifaci, V., Bruni, R., Marchetti-Spaccamela, A., 2016. ILP-based approaches to partitioning recurrent workloads upon heterogeneous multiprocessors, in: 2016 28th Euromicro Conference on Real-Time Systems (ECRTS), IEEE. pp. 215–225.

[7] Baruah, S.K., Bonifaci, V., Bruni, R., Marchetti-Spaccamela, A., 2019. ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. Journal of Scheduling 22, 195–209.

[8] Bateni, S., Wang, Z., Zhu, Y., Hu, Y., Liu, C., 2020. Co-optimizing performance and memory footprint via integrated CPU/GPU memory management, an implementation on autonomous driving platform, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 310–323.

[9] Biondi, A., Balsini, A., Pagani, M., Rossi, E., Marinoni, M., Buttazzo, G., 2016. A framework for supporting real-time applications on dynamic reconfigurable FPGAs, in: Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016), pp. 1–12.

[10] Biondi, A., Di Natale, M., 2018. Achieving predictable multicore execution of automotive applications using the LET paradigm, in: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE. pp. 240–250.

[11] Biondi, A., Pazzaglia, P., Balsini, A., Di Natale, M., 2017. Logical execution time implementation and memory optimization issues in AUTOSAR applications for multicores, in: International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS).

[12] Buttazzo, G., Bini, E., Wu, Y., 2011. Partitioning real-time applications over multicore reservations. IEEE Transactions on Industrial Informatics 7, 302–315.

[13] Capodieci, N., Cavicchioli, R., Bertogna, M., Paramakuru, A., Dec 11 - 14, 2018. Deadline-based scheduling for GPU with preemption support, in: 2018 IEEE Real-Time Systems Symposium (RTSS), Nashville, TN, USA.

[14] Capodieci, N., Cavicchioli, R., Valente, P., Bertogna, M., Oct 4-6, 2017. SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses, in: Proceedings of the 25th International Conference on Real-Time Networks and Systems, Grenoble, France.

[15] Casini, D., Biondi, A., Buttazzo, G., 2019a. Analyzing parallel real-time tasks implemented with thread pools, in: Proceedings of the 56th Annual Design Automation Conference 2019.

[16] Casini, D., Biondi, A., Buttazzo, G., 2020. Timing isolation and improved scheduling of deep neural networks for real-time systems. Software: Practice and Experience 50, 1760–1777.

[17] Casini, D., Biondi, A., Cicero, G., Buttazzo, G., 2021. Latency analysis of I/O virtualization techniques in hypervisor-based real-time systems, in: 2021 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).

[18] Casini, D., Biondi, A., Nelissen, G., Buttazzo, G., 2018. Partitioned fixed-priority scheduling of parallel tasks without preemptions, in: 2018 IEEE Real-Time Systems Symposium (RTSS).

[19] Casini, D., Blaß, T., Lütkebohle, I., Brandenburg, B., 2019b. Response-time analysis of ROS 2 processing chains under reservation-based scheduling, in: Proceedings 31th Euromicro Conference on Real-Time Systems (ECRTS 2019).

[20] Casini, D., Pazzaglia, P., Biondi, A., Buttazzo, G., Di Natale, M., 2019c. Addressing analysis and partitioning issues for the waters 2019 challenge, in: 10th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2019).

[21] Cavicchioli, R., Capodieci, N., Solieri, M., Bertogna, M., Jul 9-12, 2019. Novel methodologies for predictable CPU-To-GPU command offloading, in: 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), Stuttgart, Germany.

[22] Chen, H., Cheng, A.M.K., Kuo, Y.W., 2011. Assigning real-time tasks to heterogeneous processors by applying ant colony optimization. Journal of Parallel and Distributed computing 71, 132–142.

[23] Chen, J.J., von der Brüggen, G., Huang, W.H., Liu, C., 2017. State of the art for scheduling and analyzing self-suspending sporadic real-time tasks, in: 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–10.

[24] Chen, J.J., Nelissen, G., Huang, W.H., 2016. A unifying response analysis framework for dynamic self-suspending tasks, in: 2016 28th Euromicro Conference on Real-Time Systems (ECRTS), pp. 61–71.

[25] Chen, J.J., Nelissen, G., Huang, W.H., Yang, M., Brandenburg, B., Bletsas, K., Liu, C., Richard, P., Ridouard, F., Audsley, N., et al., 2019a. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. Real-Time Systems 55, 144–207.

[26] Chen, Z.W., Lei, H., Yang, M.L., Liao, Y., Yu, J.L., 2019b. Improved task and resource partitioning under the resource-oriented partitioned scheduling. Journal of Computer Science and Technology 34, 839–853.

[27] Davare, A., Zhu, Q., Di Natale, M., Pinello, C., Kanajan, S., Sangiovanni-Vincentelli, A., 2007. Period optimization for hard real-time distributed automotive systems, in: 2007 44th ACM/IEEE Design Automation Conference.

[28] Davis, R., Burns, A., October 2011. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. ACM Computing Surveys 43, 35:1–35:44.

[29] Dong, Z., Liu, C., 2016. Closing the loop for the selective conversion approach: A utilization-based test for hard real-time suspending task systems, in: 2016 IEEE Real-Time Systems Symposium (RTSS), pp. 339–350.

[30] Dong, Z., Liu, C., Bateni, S., Chen, K.H., Chen, J.J., von der Brüggen, G., Shi, J., 2018. Shared-resource-centric limited preemptive scheduling: A comprehensive study of suspension-based partitioning approaches, in: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE. pp. 164–176.

[31] Elliott, G.A., Ward, B.C., Anderson, J.H., 2013. GPUSync: A framework for real-time GPU management, in: 2013 IEEE 34th Real-Time Systems Symposium, pp. 33–44.

[32] Fonseca, J., Nelissen, G., Nelis, V., Pinho, L.M., 2016. Response time analysis of sporadic DAG tasks under partitioned scheduling, in: 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES).

[33] Forsberg, B., Marongiu, A., Benini, L., 2017. GPUguard: Towards supporting a predictable execution model for heterogeneous SoC, in: Design, Automation Test in Europe Conference Exhibition (DATE), 2017, Lausanne, Switzerland.

[34] Gracioli, G., Alhammad, A., Mancuso, R., Fröhlich, A.A., Pellizzoni, R., 2015. A survey on cache management mechanisms for real-time embedded systems. ACM Computing Surveys 48.

[35] Günzel, M., von der Brüggen, G., Chen, J.J., 2020. Suspension-aware earliest-deadline-first scheduling analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39, 4205–4216.

[36] Guo, Z., Bhuiyan, A., Liu, D., Khan, A., Saifullah, A., Guan, N., 2019. Energy-efficient real-time scheduling of DAGs on clustered multi-core platforms, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE. pp. 156–168.

[37] Hamann, A., Dasari, D., Wurst, F., Sañudo, I., Capodieci, N., Burgio, P., Bertogna, M., . WATERS Industrial Challenge 2019.

[38] Henzinger, T.A., Kirsch, C.M., Sanvido, M.A., Pree, W., 2003. From control models to real-time code using giotto. IEEE Control Systems Magazine 23, 50–64.

[39] Höttger, R., Krawczyk, L., Igel, B., 2015. Model-based automotive partitioning and mapping for embedded multicore systems, in: International Conference on Parallel, Distributed Systems and Software Engineering, Citeseer. p. 888.

[40] Houssam-Eddine, Z., Capodieci, N., Cavicchioli, R., Lipari, G., Bertogna, M., 2020. The HPC-DAG task model for heterogeneous real-time systems. IEEE Transactions on Computers .

[41] Huang, W.H., Yang, M., Chen, J.J., 2016. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share?, in: 2016 IEEE Real-Time Systems Symposium (RTSS), IEEE. pp. 111–122.

[42] Jeff, B., 2012. Big. LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration., in: DAC, pp. 1143–1146.

[43] Lehoczky, J., Sha, L., Ding, Y., 1989. The rate monotonic scheduling algorithm: exact characterization and average case behavior, in: [1989] Proceedings. Real-Time Systems Symposium.

[44] Li, Z., Zhang, Y., Ding, A., Zhou, H., Liu, C., 2021. Efficient algorithms for task mapping on heterogeneous CPU/GPU platforms for fast completion time. Journal of Systems Architecture 114, 101936.

[45] Liu, C., Anderson, J.H., 2010. Improving the schedulability of sporadic self-suspending soft real-time multiprocessor task systems, in: 2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 13–22.

[46] Liu, C., Anderson, J.H., 2013. Suspension-aware analysis for hard real-time multiprocessor scheduling, in: 2013 25th Euromicro Conference on Real-Time Systems, IEEE. pp. 271–281.

[47] Lowinski, M., Ziegenbein, D., Glesner, S., 2016. Splitting tasks for migrating real-time automotive applications to multi-core ECUs, in: 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES), IEEE. pp. 1–8.

[48] Maiza, C., Rihani, H., Rivas, J.M., Goossens, J., Altmeyer, S., Davis, R.I., 2019. A survey of timing verification techniques for multi-core real-time systems. ACM Computing Surveys .

[49] Mascitti, A., Cucinotta, T., Abeni, L., 2020a. Heuristic partitioning of real-time tasks on multi-processors, in: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), IEEE. pp. 36–42.

[50] Mascitti, A., Cucinotta, T., Marinoni, M., 2020b. An adaptive, utilization-based approach to schedule real-time tasks for ARM big. LITTLE architectures. ACM SIGBED Review 17, 18–23.

[51] Mascitti, A., Cucinotta, T., Marinoni, M., Abeni, L., 2021. Dynamic partitioned scheduling of real-time tasks on ARM big. LITTLE architectures. Journal of Systems and Software 173, 110886.

[52] Mittal, S., Vetter, J.S., 2015. A survey of CPU-GPU heterogeneous computing techniques. ACM Computing Surveys 47, 1–35.

[53] Nelissen, G., Fonseca, J., Raravi, G., Nélis, V., 2015. Timing analysis of fixed priority self-suspending sporadic tasks, in: 2015 27th Euromicro Conference on Real-Time Systems.

[54] Olmedo, I.S., Capodieci, N., Martinez, J.L., Marongiu, A., Bertogna, M., 2020. Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 213–225.

[55] Pagani, M., Balsini, A., Biondi, A., Marinoni, M., Buttazzo, G., 2017. A Linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration, in: 2017 30th IEEE International System-on-Chip Conference (SOCC), pp. 96–101.

[56] Pagani, M., Rossi, E., Biondi, A., Marinoni, M., Lipari, G., Buttazzo, G., 2019. A bandwidth reservation mechanism for AXI-Based hardware accelerators on FPGAs, in: 31st Euromicro Conference on Real-Time Systems (ECRTS 2019).

[57] Park, M., Park, H., 2014. An efficient test method for rate monotonic schedulability. IEEE Transactions on Computers 63, 1309–1315.

[58] Pazzaglia, P., Biondi, A., Di Natale, M., 2019a. Optimizing the functional deployment on multicore platforms with logical execution time, in: 2019 IEEE Real-Time Systems Symposium (RTSS), IEEE. pp. 207–219.

[59] Pazzaglia, P., Biondi, A., Natale, M.D., 2019b. Simple and general methods for fixed-priority schedulability in optimization problems, in: Proceedings of the International Conference on Design, Automation and Test in Europe (DATE 2019).

[60] Pazzaglia, P., Casini, D., Biondi, A., Di Natale, M., 2021. Optimal memory allocation and scheduling for DMA data transfers under the LET paradigm, in: 58th Design Automation Conference (DAC).

[61] Pfenning, S., Holzinger, P., Reichenbach, M., 2021. Transparent FPGA acceleration with tensorflow. arXiv preprint arXiv:2102.06018 .

[62] Poongothai, M., Rajeswari, A., Kanishkan, V., 2014. A heuristic based real time task assignment algorithm for heterogeneous multiprocessors. IEICE Electronics Express 11, 20130975–20130975.

[63] Restuccia, F., Biondi, A., Marinoni, M., Buttazzo, G., 2020a. Safely preventing unbounded delays during bus transactions in FPGA-based SoC, in: 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).

[64] Restuccia, F., Biondi, A., Marinoni, M., Cicero, G., Buttazzo, G., 2020b. AXI HyperConnect: A predictable, hypervisor-level interconnect for hardware accelerators in FPGA SoC, in: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6.

[65] Restuccia, F., Pagani, M., Biondi, A., Marinoni, M., Buttazzo, G., 2020c. Modeling and analysis of bus contention for hardware accelerators in FPGA SoCs, in: 32st Euromicro Conference on Real-Time Systems (ECRTS 2020).

[66] Serrano, M.A., Quinones, E., 2018. Response-time analysis of DAG tasks supporting heterogeneous computing, in: Proceedings of the 55th Annual Design Automation Conference, pp. 1–6.

[67] Seyoum, B.B., Biondi, A., Buttazzo, G.C., 2019. FLORA: floorplan optimizer for reconfigurable areas in FPGAs. ACM Trans. Embed. Comput. Syst. 18.

[68] Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P., Jahre, M., Vissers, K., 2017. FINN: A framework for fast, scalable binarized neural network inference, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 65–74.

[69] Valente, G., Di Mascio, T., D'Andrea, G., Pomante, L., 2020. Dynamic partial reconfiguration profitability for real-time systems. IEEE Embedded Systems Letters , 1–1.

[70] Wang, H., Audsley, N.C., Chang, W., 2020. Addressing resource contention and timing predictability for multi-core architectures with shared memory interconnects, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 70–81.

[71] Wieder, A., Brandenburg, B.B., 2013. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks, in: 2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES), IEEE. pp. 49–58.

[72] Wurst, F., Dasari, D., Hamann, A., Ziegenbein, D., Sanudo, I., Capodieci, N., Bertogna, M., Burgio, P., 2019. System performance modelling of heterogeneous HW platforms: An automated driving case study, in: 2019 22nd Euromicro Conference on Digital System Design (DSD), pp. 365–372.

[73] Zahaf, H.E., Olejnik, R., Lipari, G., et al., 2017. Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms. Journal of Systems Architecture 74, 46–60.