Scheduling real-time template-tasks on manycores heterogeneous platforms

Houssam-Eddine Zahaf, Nicola Capodieci, Audrey Queudet

Scheduling real-time template-tasks on manycores heterogeneous platforms

Houssam-Eddine Zahaf
Nantes Université, Centrale Nantes,
INRIA (1), LS2N, UMR 6004
F-44000 Nantes, France
houssameddine.zahaf@univ-nantes.fr

Nicola Capodieci
University of Modena, Dept. of Physics,
Informatics and Mathematics
Modena, Italy
nicola.capodieci@unimore.it

Audrey Queudet
Nantes Université, Centrale Nantes,
INRIA (1), LS2N, UMR 6004
F-44000 Nantes, France
audrey.queudet@univ-nantes.fr

***Abstract*—Recent broadband cellular network technology requires to guarantee respect for timing constraints to ensure the required Quality of Service (QoS) and improve user experience. The Base Transceiver Station (BTS) must activate and process a large set of real-time tasks, representing user requests, on a heterogeneous many-core platform. Each request is an instance of a heterogeneous parallel and conditional Directed Acyclic Graph (HPC-DAG) task that must complete no later than a predefined deadline. Classical real-time scheduling and schedulability analysis approaches can not be directly applicable to such systems because task activation profiles can drastically change based on end-user requests. Additionally, classical schedulability analysis might exhibit scalability issues in supporting a large number of task instances on hundreds of cores.**

This work addresses the problem of allocating and scheduling a set of real-time tasks to a heterogeneous platform composed of hundreds of cores. We present efficient scheduling and allocation approaches to guarantee that all timing constraints are respected within a single scheduling frame. We propose novel schedulability tests, allowing us to explore numerous possible design choices for user requests. The performance of the proposed approaches is studied through a large set of synthetic experiments.

***Index Terms*—Real-time, Scheduling, HPC-DAG, Templates, Runtime Instantiation, Multi-objective Optimization, Scheduling Configuration**

I. Introduction

5G networks are cellular networks that divide the service area into small cells. In each cell, all 5G wireless devices communicate via radio waves with a base station using frequency channels assigned by the station. These base stations connect to switching centers and routers for phone services, internet access, and other services through high-speed connections to the core network.

5G users have diverse requests, ranging from phone calls and internet browsing to industrial low-latency communication. Each service triggers specific functionalities modeled as Directed Acyclic Graph (DAG) tasks processed by the base station. For example, users in the same cell with similar needs might trigger similar DAG tasks. These tasks consist of subtasks (nodes) representing signal processing operations and the flow of data between them (edges). They are scheduled to minimize latency and meet real-time constraints. DAGs may also involve conditional execution based on runtime evaluations.

Different 5G services have varying urgency levels. To guarantee a certain quality of service, each DAG task associated with a service is subject to timing constraints and must be completed within a predefined window. To ensure all user requests are fulfilled within these scheduling frames, all DAG instances must complete by their deadlines. Providers equip base stations with many-core platforms (around a hundred cores) featuring CPUs and accelerators like DSPs. A single station typically serves hundreds of users concurrently. User requests and their corresponding DAG tasks are processed in intervals. A base station handles all considered user requests within a single time window. In other words, once processing begins, any incoming requests are queued and processed after the current batch is complete. We call this time window a scheduling frame. User requests can change drastically between scheduling frames. Classical real-time scheduling techniques are not suitable for handling large-scale scheduling due to their complexity and scalability limitations. Additionally, they lack the flexibility to adapt to varying user requests across different scheduling frames. The problem is further more complex by the heterogeneous nature of the platform (different instruction sets, microarchitectures) and the dynamic behavior of tasks. Commonly, 5G operators address these issues by increasing computational power and cores on their platforms. They also employ on-the-fly list scheduling across all cores. However, this approach introduces significant overhead due to complex scheduler design, making it challenging to guarantee timing constraints.

Contributions: In this work, we propose a novel scheduling and schedulability analysis for a set of DAG task templates on heterogeneous many-core platforms. Templates allow for dynamic instantiation at runtime based on user requests, whose profiles are unknown beforehand and can vary significantly between scheduling frames. All instances must complete within a single scheduling frame. Unlike existing task schedulers in 5G BTS that rely only on runtime decisions, our approach leverages offline scheduling to reduce online scheduler complexity.

Our approach addresses several key challenges: (i) efficiently allocating and scheduling a large number of task instances across the platform's diverse cores, (ii) handling cores with different instruction set architectures (ISA), and (iii) adapting to variations in user requests between scheduling frames. By leveraging multiple scheduling scenarios, our approach is able to dynamically adapt allocation and scheduling based on real-time activation requests while having a simpler runtime-scheduler design.

II. Related work

In this work, we focus on real-time task scheduling on a many-core platform. We briefly review the current

trends in many-core scheduling within the context of 5G networks and real-time scheduling on heterogeneous multicore platforms. Platforms can be categorized into two types: homogeneous and heterogeneous. In homogeneous platforms, all processors are identical while in heterogeneous platforms, processors differ, leading to different execution times for the same task on different cores. Furthermore, a task may not be allowed to execute on all processors due to differences in the ISA.

Compute platforms in BTS are heterogeneous, incorporating multiple accelerators alongside general-purpose processors, utilizing multi-core platforms with digital signal processors (DSP) and other accelerators [1] [2]. Related work in 5G task scheduling frequently employs the list scheduling approach, where tasks are executed in isolation, utilizing their dedicated resources [3], [4]. Typically, execution decisions are made on-the-fly according to user requests, and real-time tasks are scheduled as best-effort tasks. However, this approach does not guarantee the respect of timing constraints. During periods of high load, user requests might be denied.

Designing complex real-time applications on multicore exposes the designer to a number of choices, like deciding task allocation, the degree of parallelism, the scheduling policy, etc. Many models of computation have been proposed to capture the specific properties of real-time systems. The Directed Acyclic Graphs (DAGs) are one of the most popular models that are able to capture the parallel execution in real-time tasks [5], [6], [7], [8]. In this model, each vertex represents a sub-task, while edges express control flow dependencies among them. The real-time community has proposed schedulability tests to check whether all tasks in the system will meet their deadlines [9]. The HPC-DAG task model [10] has been recently proposed, as an extention of conditional DAG model [6]. It introduces the alternative feature, allowing alternative implementations of parts of the task. Authors in [11] addressed the worst-case response time (WCRT) analysis for DAG tasks on heterogeneous platform. Similar work has been achieved by [12] but limited to simpler platforms, featuring only CPU-GPU. Even when having the same ISA, differences in microarchitecture can have an important impact on execution times, increasing schedulability analysis complexity [13]. Authors in [14] targeted global scheduling approaches for heterogeneous processors. However, all the works cited have a high complexity that is not scalable to the large number of cores that features BTS platforms. In addition, they are not flexible enough to handle the severe changes that task activations might experience, according to the end-user requests.

In this work, we adopt the approach of allocating resources exclusively to a task instance during its execution. This ensures that the requirements of a simple online scheduler are met. Additionally, we provide both spatial and temporal isolation to respect task deadlines and to support various end-user requests. Our approach is proposed for, but not limited to, list schedulers."

III. System model

A. Hardware platform

We consider a hardware platform compound of m^* cores. Cores are indexed incrementally, we denote by p_j the j^{th} core of the platform. Each core p_j is characterized by its $\mathsf{ISA}(p_j)$. A typical architecture that we consider in this work is described in Figure 1. We denote by m_{g} the count of cores having ISA equal g as, and by ISAs the list of all ISAs supported in the platform.

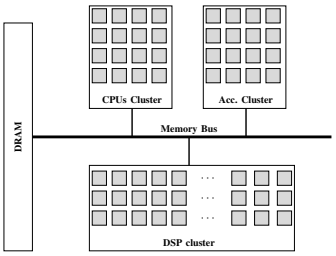


Fig. 1: A typical hardware architecture

B. HPC-DAG task model

In this work, we extend the HPC-DAG model [10], which stands for Heterogeneous Parallel Conditional DAG model, to support the differences that might exist at the instantiation stage of user requests. According to the moment when a task is considered, it can be in one of three stages: (i) specification task, (ii) concrete task, and (iii) runtime task. We consider a set of n tasks, that is, $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$.

1) Specification task: It is a DAG, that captures all the application developer specifications, denoted as τ_i. $\mathsf{D}(\tau_i)$ defines the task's relative deadline, $\mathcal{V}(\tau_i)$ is a set of graph nodes representing *subtasks*, $\mathcal{A}(\tau_i)$ is a set of *alternative nodes*, and $\Gamma(\tau_i)$ is a set of *conditional nodes*. The set of all the nodes is represented by $\mathcal{N} = \mathcal{V} \cup \mathcal{A} \cup \Gamma$. $\mathcal{E}(\tau_i)$ is the set of edges of the graph. An edge $e(n_i, n_j) \in \mathcal{E}(\tau_i)$ represents a precedence constraint and related communication between node n_i and node n_j, where n_i and n_j can be subtasks, alternative nodes or conditional nodes. The *immediate predecessors* of a node n_j is denoted by $\mathsf{pred}(n_j)$ and is the set of all nodes n_i for which there exists an edge (n_i, n_j). The set of *predecessors* of a node n_j is the set of all nodes for which there exists a path towards n_j. A node with no predecessor is a *source node* of the graph, and a graph can have several source nodes. The *immediate successors* of a node n_j is denoted by $\mathsf{succ}(n_j)$ and is the set of all nodes n_k for which there exists an edge (n_j, n_k). The set of *successors* of a node n_j is the set of all nodes for which there exists a path from n_j. A node with no successors is a *sink node* of the graph, and a graph can have several sink nodes.

A *subtask* $v \in \mathcal{V}(\tau_i)$ is the basic schedulable unit and represents a block of code to be executed sequentially. A subtask is characterized by:

- An instruction set architecture $\mathtt{ISA}(v)$ which defines the cores where the subtask is eligible to execute.
- A worst-case execution time $\mathcal{C}(v)$ when executing the subtask on a core having the same ISA.

A conditional node $\gamma \in \Gamma(\tau_i)$ represents alternative paths in the graph due to non-deterministic on-line conditions.

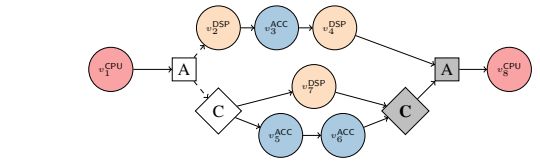
* m is equal to more than 100 compute elements

Non-determinism implies that at runtime, only one of the outgoing edges of γ is executed, but it is not possible to know in advance which one. An alternative node $a \in \mathcal{A}(\tau_i)$ represents alternative implementations of parts of the graph/task. During the configuration phase, our methodology selects one among many possible alternative implementations of the program by selecting only one of the outgoing edges of a and removing (part of) the paths starting from the other edges according to the *task activation profile*, timing constraints, and available resources. This can be useful when modeling subtasks that can be executed on different cores with different execution costs. Conditional nodes and alternative nodes always have at least two outgoing edges, so they cannot be sinks. For the sake of simplicity and without loss of generality, we also assume that conditional nodes always have at least one predecessor node, so they cannot be sources.

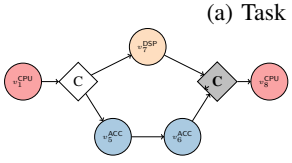
2) Concrete tasks: A concrete task is an instance of a specification task where all alternatives have been removed by making implementation choices. It is represented as $\overline{\tau}$. $\overline{\mathcal{V}}, \overline{\Gamma}, \overline{\mathcal{E}}$ are respectively the subset of subtasks, conditional nodes and their precedence constraints that have been selected to generate the concrete task $\overline{\tau}$. We denote by $\Omega(\tau)$ the set of all concrete tasks of specification task τ. The procedure to generate all concrete tasks is defined in [10].

3) Runtime tasks: A runtime task is an instantiation of a concrete task according to a end-user request. This instantiation is achieved at runtime and is known at the start of a scheduling frame. At this stage, conditional choices are known. We denote by $\overline{\overline{\tau}}$ a runtime task of concrete task $\overline{\tau}$. We denote $\Delta(\overline{\tau})$ the set of all possible runtime tasks of concrete task $\overline{\tau}$.

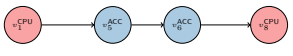
Example 1. *Consider the specification task shown in Figure 2a. Each subtask node is labeled by its index and* ISA. *Alternative nodes are represented by square boxes, while conditional nodes are represented by diamond boxes. The gray boxes indicate the corresponding closing nodes for alternatives and conditionals.*



(a) Task specification



(b) Concrete task $\overline{\tau}$



(c) Runtime task $\overline{\overline{\tau}}$

Fig. 2: Specification, concrete and runtime tasks

Subtask v_1^{CPU} is the source node of the DAG and is marked with the CPU ISA, indicating that it will be allocated exclusively to a core having ISA of type CPU. *Subtask v_3^{ACC} has an outgoing edge to v_4^{DSP}, so subtask v_4^{DSP} cannot start its execution before subtask v_3^{ACC} has completed. Subtask v_1^{CPU} has an outgoing edge to alternative node A. Thus, τ can continue the execution either: (i) by following v_2^{DSP} and then v_3^{ACC}, v_4^{DSP}, and finishing its instance on v_8^{CPU}; or (ii) by following the conditional node C and selecting, according to an undetermined condition evaluated online, either to execute (i) v_5^{ACC} followed by subtask v_6^{ACC} or (ii) v_7^{DSP}. The two sub-graphs represent alternative ways to execute the same functionalities at different costs.*

Figure 2b represents one of the concrete tasks of τ. Figure 2c represents a possible instantiation of the concrete task $\overline{\tau}$.

In this work, we restrict to *well-nested concrete graphs*: for every conditional (respectively alternative) node, there is always a corresponding *closing node* denoted with gray color in Figure 2a, such that all paths starting from the conditional (respectively alternative) node includes the corresponding closing node.

We consider scheduling within a single time-frame. Without loss of generality, we set the scheduling frame length T to the largest task deadline, i.e., $\mathsf{T} = \max\{\mathsf{D}(\tau), \forall\tau\}$.

We present now a set of definitions and symbols that might be applyable to specification, concrete or runtime tasks. Therefore, when the task stage is not required to be defined, the task is represented by "$\cdot$".

We denote by $\pi(\cdot)$ an arbitrary path of a given task. We denote by $\pi^h(\cdot)$ the h^{th} path of the task. $\Pi(\cdot)$ denotes the set of all paths of the task in the parameter.

We denote by $len(\pi(\cdot))$ the length of path $\pi(\cdot)$ and it is computed as follows:

$$len(\pi(\cdot)) = \sum_{v \in \pi(\cdot)} \mathsf{C}(v)$$

We denote by $\mathsf{crit}(\cdot)$ the critical path of a given task, such that :

$$len(\mathsf{crit}(\cdot)) = \max\{len(\pi(\cdot)), \forall \pi(\cdot) \in \Pi(\cdot)\}$$

We denote by $vol(\overline{\tau})$ the total cumulative WCET of concrete task $\overline{\tau}$. It is computed as follows :

$$vol(\overline{\tau}) = \max_{\forall \overline{\overline{\tau}} \in \Delta(\overline{\tau})} \{ \sum_{v \in \mathcal{V}(\overline{\overline{\tau}})} \mathsf{C}(v) \}\} \quad (1)$$

We denote by $vol_g(\overline{\tau})$ the total cumulative WCET of the concrete task for subtasks having ISA g. It is computed as follows :

$$vol_g(\overline{\tau}) = \max_{\forall \overline{\overline{\tau}} \in \Delta(\overline{\tau})} \{ \sum_{v \in \mathcal{V}(\overline{\overline{\tau}})} \{\mathsf{C}(v) | \mathsf{ISA}(v) = g\}\}$$

We define $U_g(\overline{\tau})$ as the utilization of the concrete task $\overline{\tau}$ on compute elements having ISA g, it is computed as follows:

$$U_g(\overline{\tau}) = \frac{vol_g(\overline{\tau})}{\mathsf{T}}$$

Finally, $|\overline{\tau}, g|$ denotes the number of subtasks of concrete task $\overline{\tau}$ having ISA g.

IV. Schedulability as a Multiobjective Optimization Problem

The main challenge of this work is to allocate a set of specification tasks to a platform composed of hundreds

of heterogeneous cores. In addition, it must take into account the severe variability that might be exhibited in end-user requests between two scheduling frames; i.e., a given number of activations of the same task in one frame might be very different from the number of activations of the same task in another frame. One approach to solving this problem is to use a pure on-the-fly scheduling approach that takes into account handling user requests at runtime, similarly to the current trend in industry. However, this approach does not provide guarantees on the timeliness of the end-user requests, and some of the user requests are dropped and only "high" level urgency tasks are considered.

Our approach builds offline a set of different schedulable scenarios that exhibit different activation profiles. At runtime, the most suitable scheduling scenario is selected. The main challenge of our approach is to generate accurate and representative scheduling scenarios that can handle a wide range of activation profiles.

Definition 1. *(Task Activation Profile) Activation profile* $\mathsf{act}(\overline{\tau})$ *of the concrete task* $\overline{\tau}$ *is a number of instances of* $\overline{\tau}$ *that can be active within a single scheduling frame, ensuring that all the* $\mathsf{act}(\overline{\tau})$ *instances of* $\overline{\tau}$ *complete no later than* $\mathcal{D}(\tau)$.

A task activation profile is not unique; a task might have multiple activation profiles during its lifetime.

Definition 2. *(Scheduling configuration)*

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ *be a set of* n *specification tasks, scheduled onto* m *cores.*

We denote by $\mathcal{S}^a(\mathcal{T})$ *the* a^{th} *scheduling configuration of task set* $\mathcal{T}$*, if and only if:*

- *For every task* τ*, a concrete task* $\overline{\tau}$ *is selected*
- *For concrete task* $\overline{\tau}$*, an activation profile is associated*

Our objective is to maximize the activation profiles for the different tasks. Since multiple specification tasks need to be optimized simultaneously, maximizing the number of instances for one task specification is contradictory to the objective of maximizing the number of instances for other specification tasks. Constructing a single activation profile that can accurately represent user requests across different scheduling frames is impractical. Instead of relying on a single scheduling configuration, as in classical real-time systems, we generate multiple scheduling configurations. At runtime, the system can dynamically select the most suitable scheduling configuration to be adapted according to the actual end-user requests.

Definition 3. *(Dominance) Let* $\mathcal{S}^1$ *and* $\mathcal{S}^2$ *be two distinct scheduling configurations.*

Configuration $\mathcal{S}^1$ *dominates* $\mathcal{S}^2$ *if and only if:*

$$\forall \tau, \mathsf{act}(\overline{\tau}) \in S^1 > \mathsf{act}(\overline{\tau}) \in S^2$$

Definition 4. *(Incomparability) Let* $\mathcal{S}^1$ *and* $\mathcal{S}^2$ *be two scheduling configurations.*

$\mathcal{S}^1$ *and* $\mathcal{S}^2$ *are* incomparable *if* $\mathcal{S}^1$ *does not dominate* $\mathcal{S}^2$ *and* $\mathcal{S}^2$ *does not dominate* $\mathcal{S}^1$.

The objective of this work is to generate a set of incomparable solutions that better represent a wide range of end-user requests, forming the Pareto front of scheduling configurations (as depicted in Figure 3).

Example 2. *Let us consider the following example. Our taskset is composed of two tasks* τ_1 *and* τ_2*. The results of Pareto front are represented in Figure 3.*

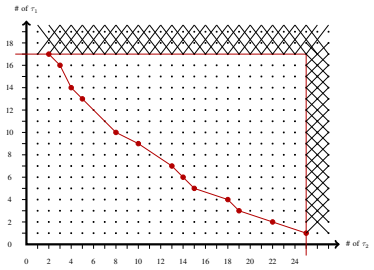


Fig. 3: An example of the pareto front of scheduling configurations for 2 tasks

On the first scheduling configuration on the left, the activation profile of $\mathsf{act}(\tau_1)$ *is equal to* 17*, and the one of* τ_2 *it is equal to* 2*. In the next configuration, we have* 16 *of task* τ_1 *and* 3 *of task* τ_2*. These two solutions as well as the other scheduling configurations presented in this figure, are incomparable.*

The optimization problem addressed in this paper consists of two distinct sub-problems: (i) building the different scheduling configurations (the Pareto front), and (ii) assessing the schedulability of every scheduling configuration. To avoid combinatorial explosion, it is necessary to find a compromise in complexity between the technique used to build the Pareto front and the techniques used to assess schedulability. In our approach, every task instance executes exclusively on a subset of cores in a federated scheduling fashion, in line with industrial approach. This allows a simple yet efficient runtime schedulers design. During the schedulability analysis phase, the maximal number of task instances that can be activated while still meeting the deadline is computed. We build an optimal pareto front.

V. HPC-DAG TEMPLATES SCHEDULABILITY ANALYSIS

In our approach, a task runs exclusively on a selected subset of cores. Two primary are considered: (i) determining the schedulability of a task, and (ii) defining the size of the core subset on which a given task executes. We will begin by presenting the schedulability analysis for a single specification task executing on a predefined subset of cores. Further, we will explain how to extend this analysis to multiple tasks and the entire platform.

A. Single task schedulability and execution subset of cores

We use list scheduling, similarly to existing solutions [2]. It is required to compute the WCRT, which depends on the concurrency within a task and the interference among its sub-tasks. This subsection presents two distinct schedulability tests, based on WCRT bounds derived from the literature. Our methodology is flexible and can accommodate other WCRT analysis for typed DAGs.

1) Single task worst-case response time: Different concrete tasks derived from the same specification task may require varying types and numbers of cores for execution, potentially leading to different WCRTs. At this stage, let's assume that concrete implementation decisions have been made. Given the large size of the platform,

each concrete task instance will only execute on a specific subset of cores. We will also assume that the number of cores on which a concrete task instance $\overline{\tau}$, will execute is known and represented as $\delta(\overline{\tau})$. The number of cores of ISA g allocated to task $\overline{\tau}$ is denoted by $m_g^{\delta(\overline{\tau})}$.

Let $\mathcal{R}(\overline{\tau}, \delta(\overline{\tau}))$ denote the WCRT for the concrete task $\overline{\tau}$ on the subset of cores $\delta(\overline{\tau})$. If $\mathcal{R}(\overline{\tau}, \delta(\overline{\tau})) \leq \mathsf{D}(\tau)$, then the task is schedulable; otherwise, it is not. In the literature, various methods have been proposed for computing the WCRT for typed DAG tasks (i.e., tasks without conditionals or alternatives) in [16] and [17].

Jeffrey et al. [16] introduced the first response time bound for typed DAG tasks on a heterogeneous platform.

Theorem 1. *(Jeffrey et al. [16]) Let τ be a* typed *DAG task. The response time of τ is upper-bounded by:*

$$\mathcal{R}(\tau) = len(\mathsf{crit}(\tau)) + \sum_{g \in \mathsf{ISAs}} \frac{vol_g(\tau)}{m_g^{\delta(\overline{\tau})}} - \frac{len(\mathsf{crit}(\tau))}{\max_g\{m_g^{\delta(\overline{\tau})}\}} \quad (2)$$

Han et al. [17] enhanced the Jeffrey-bound [16] by computing tighter bounds for intra-task interferences. This is achieved by enumerating parallel paths within a single task and identifying in part the interference that they might have on each other.

Theorem 2. *(Han et al. [17]) Let τ be a* typed *DAG task. The response time of τ is upper bounded by :*

$$\begin{cases} \mathcal{R}(\tau) = & \max_{\pi^h \in \Pi} \overline{R}(\pi^h) \\ \overline{R}(\pi^h) = & len(\pi^h) + \sum_{g \in \mathsf{ISAs}} \sum_{v \in \mathsf{ivs}(\pi^h, g)} \frac{\mathsf{C}(v)}{m_g^{\delta(\overline{\tau})}} \end{cases} \quad (3)$$

where $\mathsf{ivs}(\pi, g)$ is the set of subtasks having ISA g that are not in the path π, but can block sub-tasks of type g in π. It can be computed as the set minus of all sub-tasks and all ancestors and predecessors of all sub-tasks in path π.

The response time analysis detailed in Equations (2) and (3) is specifically designed for typed-DAG tasks without alternatives or conditionals. These analyses can not be directly applied to the HPC-DAG specification task. The Jeffrey and Han bounds can be easily adapted for the HPC-DAG task model by incorporating conditional nodes in the computation of the concrete task volume, as outlined in Equation (1). Conversely, alternative nodes provide the flexibility to apply these bounds to various concrete tasks, where the choice of a concrete task may depend on the task's timing constraints and the specific subset of cores allocated for the task instance's execution.

Selecting a concrete task with the smallest WCRT might lead to an increased demand on scarce resources, which are typically more efficient. In certain scenarios, it might be advantageous to choose a concrete task that reduces the demand on limited computing resources, even if this choice leads to a larger WCRT. In subsequent sections, we will explore how the selection of concrete tasks influences the effectiveness of our proposed approach.

2) Task minimal platform: As indicated by Equations (2) and (3), the response time of a concrete task is primarily determined by the task's volume for each ISA and the number of cores allocated per ISA. Consequently, it is crucial to define the number of cores per ISA necessary for a given task to meet its deadline. Allocating too many cores may lead to a shorter response time, but this could potentially decrease the number of task instances that can be executed in parallel. Conversely, allocating too few cores could increase the task's response time, possibly compromising the task's ability to meet its schedulability requirements. There are likely numerous core configurations that could satisfy the schedulability constraints. In this section, we propose a technique to focus only on the most advantageous configurations. Let's assume that the selection of concrete tasks has already been completed.

Example 3. *In this example, we consider concrete task $\overline{\tau}$ described in Figure 4. The task deadline is equal to 30.*

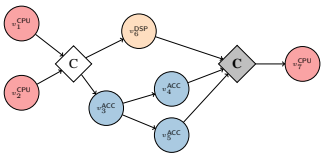


Fig. 4: Task example

Sub-task	ISA	$\mathcal{C}(v)$
v_1	CPU	5
v_2	CPU	5
v_3	ACC	4
v_4	ACC	7
v_5	ACC	7
v_6	DSP	3
v_7	CPU	6

Fig. 5: Task execution times

We will compute Jeffrey et al. response time bounds. Let consider the set of cores $\delta^1(\overline{\tau})$ compound of 4 CPUs, 5 DSPs and 3 ACCs. The response time for $\overline{\tau}$ on δ^1 is therefore equal to 28.2. When considering the set of cores $\delta^2(\overline{\tau})$ compound of 3 DSPs, 3 CPUs and 3 ACCs. The response time for $\overline{\tau}$ on δ^2 is equal to 27. For the the set of cores $\delta^3(\overline{\tau})$ compound of 2 DSPs, 2 CPUs and 2 ACCs, the response time is equal to 29.5, while for the set of cores δ^4, compound of 1 CPU, 1 DSP, and 1 ACC, the response time is equal to 37. The set of cores in δ^2 has fewer resources than δ^1 in every ISA and in addition results in a shorter response time. Thus, from a perspective of schedulability and resource utilization, it is more advantageous to use platform δ^2. It is important to note that the response time on platform δ^2 is less than that on δ^3. However, since δ^3 uses fewer resources, comparing the two response times directly is not straightforward; each platform presents its advantages and limitations. The last set of cores can not be considered, as it does not allow to respect the timing constraints, i.e. $\mathsf{D}(\tau) = 30$.

Similar observations apply to the computation of Han et al. bound.

Definition 5. *(Core configuration dominance)*

Let $\overline{\tau}$ be a concrete task and $\delta^1(\overline{\tau})$ and $\delta^2(\overline{\tau})$ be two core configurations to execute $\overline{\tau}$.

$\delta^1(\overline{\tau})$ dominates $\delta^2(\overline{\tau})$ if and only if

$$\exists g \in \mathsf{ISAs}, \quad m_g^{\delta^1(\overline{\tau})} < m_g^{\delta^2(\overline{\tau})} \quad (4)$$

$$\forall g' \neq g, \quad m_{g'}^{\delta^1(\overline{\tau})} \leq m_{g'}^{\delta^2(\overline{\tau})} \quad (5)$$

$$\mathcal{R}(\tau, \delta^1(\overline{\tau})) \leq \mathcal{R}(\tau, \delta^2(\overline{\tau})) \quad (6)$$

Definition 6. *(Core configuration incomparability)*

Let $\overline{\tau}$ be a concrete task and $\delta^1(\overline{\tau})$ and $\delta^2(\overline{\tau})$ be two core configurations.

$\delta^1(\overline{\tau})$) and $\delta^2(\overline{\tau})$ are incomparable if $\delta^1(\overline{\tau})$ is not dominated by $\delta^2(\overline{\tau})$ and $\delta^1(\overline{\tau})$ does not dominate $\delta^2(\overline{\tau})$.

The objective of our approach is to assess all incomparable core configurations that enable a given concrete task to meet its timing constraints. While one may examine every configuration from zero core up to the total number of cores available on the platform, the design space can become exceedingly vast, especially with platforms comprising hundreds of cores.

For a given task, if each subtask is allocated its own core, it will not experience interference and will start its execution immediately after its predecessor completes. In such scenarios, the WCRT is equal to the length of the concrete task's critical path. Consequently, we define the upper bound for the subset of cores as the number of subtasks per (ISA), that is:

$$\forall g \in \mathsf{ISAs}, m_g^{\delta^x} = |\overline{\tau}, g| \tag{7}$$

where $|\overline{\tau}, g|$ denotes the number of subtasks of task $\overline{\tau}$ that use ISA g.

We are aware that WCRT bounds provided in Equations (2) and (3) tend to overestimate the response time, implying that the actual response time could be smaller. Nevertheless, this bound is likely sufficient for respecting the task's timing constraints because it overestimates the task's requirements. Indeed, subtasks belonging to the same paths do not generate interference and can likely be executed on the same core without degrading the task's response time.

Our approach explores iteratively all incomparable core configurations, starting with the upper bound in Equation 7 and decreasing to an empty configuration without any cores. In each iteration, we examine a specific core configuration and calculate the WCRT for the considered concrete task. If the WCRT is shorter than or equal to the task's deadline and the new core configuration is not *dominated* (as per Definition 5), it is incorporated into the collection of non-dominated core configurations. If the new core configuration dominates other already core configurations in the non-dominated list computed core configurations, these are dropped. This process continues iteratively until all potential core configurations have been evaluated. The complete set of non-dominated core configurations is denoted as $\delta^{all}(\overline{\tau})$.

The technique outlined in this section for generating the list of core configurations considers exclusively concrete tasks. We consider the complete set of incomparable core configurations for a task specification. Therefore, we use an iterative process that generates the list of all core configurations for each concrete task. Subsequently, we merge these configurations, excluding dominated core configuration. The generation of core configurations is conducted only once, and the computed core configurations are not recomputed during the schedulability analysis.

In the following section, we will describe different techniques for selecting core configurations based on activation profiles, WCRT, required resources, and their scarcity, etc.

B. Task set schedulability

In the previous section, we focus on the execution of individual task instances in isolation. However, within a scheduling frame, multiple instances of various specification tasks are executed concurrently. To facilitate the execution of different task instances across the entire platform, resources must be allocated to concrete tasks, both spatially and temporally.

In this section, we will outline the techniques employed to allocate and schedule task instances onto different cores. Typically, a set of resources is allocated to a task instance, which remain dedicated to that instance during its execution. Upon completion, the task releases the resources, making them available for other task instances within the same scheduling frame.

Definition 7. *(Partial and Complete Configuration)*
A complete configuration is a scheduling configuration onto which every specification task is associated with a number of instances, scheduled without missing deadlines.
A partial configuration is a configuration that is not complete.

It is important to note that the objective of this work is to create a set of non-dominated scheduling configurations, each corresponding to different activation profiles. Algorithm 1 is invoked to perform schedulability analysis and to construct all possible scheduling configurations.

Algorithm 1 is iterative and recursive. At each invocation, it computes iteratively the different possible scheduling configurations for a single specification task in the input. Therefore, it produces a set of scheduling configurations. When all specification tasks are processed, all produced non-dominated scheduling configurations are returned, building therefore our pareto front used at runtime. For the initial launch, the partial scheduling configuration in the input is completely empty, i.e., no number of instances is associated with possible activation profiles for the specification task in the input. Tasks are submitted to Algorithm 1 in according to their relative deadline, (*i.e.* the smallest deadline first).

Algorithm 1 full_algorithm

```
1:  Input: Configuration config, sched, τ
2:  Output: ConfigList,
3:  if (isComplete(config)) then
4:      add_to_pareto(config, configList)
5:  end if
6:  (inst_numb, core_list) = max_instances(config, τ)
7:  while (inst_numb ≥ 0) do
8:      input_nb = min(inst_numb, input_nb);
9:      configT = clone(config)
10:     allocate_max_instances(configT, τ)
11:     τ = nextTask;
12:     full_algorithm(configT, sched, τ);
13:     inst_numb = inst_numb − 1
14: end while
15: return true;
```

The algorithm begins by checking if the current scheduling configuration is complete (Line 3), i.e., if all specification tasks have been processed. If it is complete, the current scheduling configuration is added to the configuration list. Dominance is checked, and all dominated scheduling configurations are discarded. Otherwise, there are still tasks that have not been processed yet. In such cases, the algorithm computes the maximum number of instances of the specification task in input, that can be feasibility allocated to different resources of

the scheduling configuration in input, in addition to those already allocated to it (Line 6). It's important to note that the invocation of the max_instances algorithm in Line 6 also returns the list of eligible cores on which instances of the current specification task might be allocated. The techniques for computing the number of instances and the list of candidate cores are detailed in the subsequent section.

Once the maximal number of instances is defined, the iterative process of our algorithm starts. It tries to allocate a given number of task instances ranging from the maximum number (defined in Line 6) to 0. Therefore, each iteration will create a new scheduling configuration, starting from the scheduling configuration in input (Line 9). Furthermore, the algorithm invokes itself recursively using the copied configuration and the next specification task (Line 12).

Each configuration keeps track of the tasks that have not yet been instantiated to compute the next task to instantiate (Line 11). The recursion stops when all specification task and possible number of instances have been explored.

C. Computing maximum number of task instances

This section aims to compute the maximum number of instances for a specification task in a scheduling configuration under two conditions: the schedulability of the already allocated task instances is not jeopardized, and all instances of the current task specification meet their deadlines.

Firstly, we need to define the list of cores where the current task specification can be allocated. It's important to note that we can only allocate concrete tasks to a sub-platform. Therefore, let's assume that the concrete task has been selected, denoted as $\overline{\tau}$. The latter requires the sub-platform $\delta(\overline{\tau_i})$ to be capable of completing the task such that $\mathcal{R}(\overline{\tau}, \delta(\overline{\tau_i})) \leq \mathsf{D}(\tau_i)$.

Lemma 1. *(Eligible cores)*

Let p be a core, and $\overline{\tau_i}$ be a concrete task, and $\delta(\overline{\tau_i})$ be a core configuration from the schedulable subplatforms list. Let $\mathcal{T}^p$ be the set of tasks that have already been allocated to to p, therefore $\forall \tau_{i'} \in \mathcal{T}^p, \mathsf{D}(\tau_{i'}) \leq D(\tau_i)$ as tasks are evaluated according to their relative deadline.

Core p can be used to schedule a part of $\overline{\tau}$, if and only if:

$$\mathsf{nb\_p}(\tau_i) \geq \left\lfloor \frac{\mathsf{D}(\tau_i) - \sum_{\tau_j \in \mathcal{T}^p} \mathcal{R}(\tau_j) \cdot \mathsf{act}(\tau_j)}{\mathcal{R}(\tau_i)} \right\rfloor > 0 \quad (8)$$

Proof. Tasks are explored in the order of their relative deadlines, therefore the task to allocate have always a deadline greater than the largest deadline already allocated to the current core. Therefore the processor is locked from the frame starting time, until the response time of all already allocated tasks, that is $\sum_{\tau_j \in \mathcal{T}^p} \mathcal{R}(\tau_j) \cdot \mathsf{act}(\tau_j)$. Therefore, the task can only execute into the slack time $\mathsf{D}(\tau_i) - \sum_{\tau_j \in \mathcal{T}^p} \mathcal{R}(\tau_j) \cdot \mathsf{act}(\tau_j)$. The ratio expresses the number of instances of task $\overline{\tau}$ that can be executed without missing deadlines. □

Lemma 1 enables us to verify whether a core can be eligible to execute a given concrete task. Furthermore, Lemma 1 is applied to every core to determine its eligibility, thus constructing the list of eligible cores, denoted as core_list. We then refine the list of eligible cores to select only the subset of cores that can be effectively locked to a concrete task instance.

Algorithm 2 computes the maximum number of instances that can feasibly be allocated to the platform under the current scheduling configuration. Given a list of eligible cores and a concrete task as input, the algorithm returns the maximum number of instances (inst_max). It's important to note that Algorithm 2 computes candidate cores for allocation and does not execute the actual allocation. The main algorithm (Algorithm 1) allocates the desired number of concrete tasks within the candidate cores defined by this algorithm.

Algorithm 2 max_instances

```
1:  Input: core_list, Task : τ̄  Output: candidates, inst_max
2:  quit = false; n_inst = 0
3:  while (not quit) do
4:    alloc_state = true
5:    for (g ∈ ISAs(τ)) do
6:      alloc_cores[g] = select_cores(core_list, g, act(τ̄))
7:      If (alloc_cores[g] = ∅) then alloc_state = false endif
8:    end for
9:    if (alloc_state) then
10:     candidates = add (alloc_cores)
11:     ∀p ∈ candidates, nb_p(τ̄, p) = nb_p(τ̄, p) − 1
12:     n_inst += 1
13:   else
14:     quit = true
15:   end if
16: end while
17: return candidate, n_inst
```

Algorithm 2 is iterative. In each iteration, it attempts to allocate a new task to the list of eligible cores, terminating when it can no longer allocate any more instances of task $\overline{\tau}$ to the platform.

During each iteration, the algorithm assesses if it can add a new task instance to the platform. For each ISA g, it tries to find the required number of cores having the same ISA g in $\delta(\overline{\tau})$, within the list of eligible cores. It is mandatory that at least $nb\_p(\overline{\tau}, p) > 1$, i.e., the selected core can support at least the execution of one instance. This is achieved by the select_core function. Two scenarios may arise. Firstly, if the returned list is empty (Line 7), it means that the required number of core to execute the selected concrete task has not been found, henceforth no task instance can be further allocated to the platform. Consequently, the algorithm breaks the iterative process and returns the allocation as computed in the previous iteration (Lines 7, 14). Secondly, if the invocation of the select_cores succeeds for all ISAs, then the cores in $alloc\_cores$ can support a new instance. Hence, $nb(\overline{\tau}, p)$ for every core in $alloc\_cores$ is decremented by 1 (Line 11), as the core will execute a new instance. The number of instances is then incremented by 1 (Line 12).

The algorithm for computing eligible cores and the algorithm for computing the maximum number of

instances depend on the response time of the given task. This response time, in turn, depends on both the selected concrete task and the subplatform on which the task is to be executed within the core configurations list $\delta^{all}(\tau)$. Thefore, the concrete task and core configuration selection is performed according to one of three different policies: **Breadth heuristic:** This approach selects the concrete task and the core configuration that minimizes the WCRT. This choice improves the temporal allocation of resources by allowing for the use of the same subset of cores by different task instances within a single scheduling frame; **Depth heuristic:** This approach selects those that reduce the number of required resources, maximizing spatial parallelization and enabling multiple tasks to be performed in parallel; **Scarce resources:** This approach selects the ones that minimizes the number of used scarce resources. The scarcity of a core depends on the number of cores with the same ISA within the platform. A higher core ISA count corresponds to lower scarcity. Throught the simulations, we will now demonstrate that these extreme choices offer a good performance compromise.

VI. Results and discussions

In this section, we assess the performance of our schedulability analysis and allocation strategies in two sets of expirements.

In the first, we evaluate the algorithms on a large number of synthetically generated task sets. We compare the performance of the sub-platform selection algorithm, comparing Han and Jeffay's WCRT analysis applied to our methodology. We assess the average WCRT, the number of cores. Furthermore, we compare the performance of the core configuration selection according to the heuristics presented in the previous section to build the scheduling configurations. For this second set of experiments, we consider a hardware platform consisting of 64 DSPs, 32 CPUs, and 32 other types of accelerators. Please note that as we maximize the platform utilization by generating the complete Pareto front, the total schedulable workload utilization is always maximal. Therefore, for the second set of experiments, we vary single task utilization, but the actual schedulable workload is always maximized. Consequently, it will not be used as a metric to evaluate our performance, as in the classical real-time system literature.

A. Task set generation

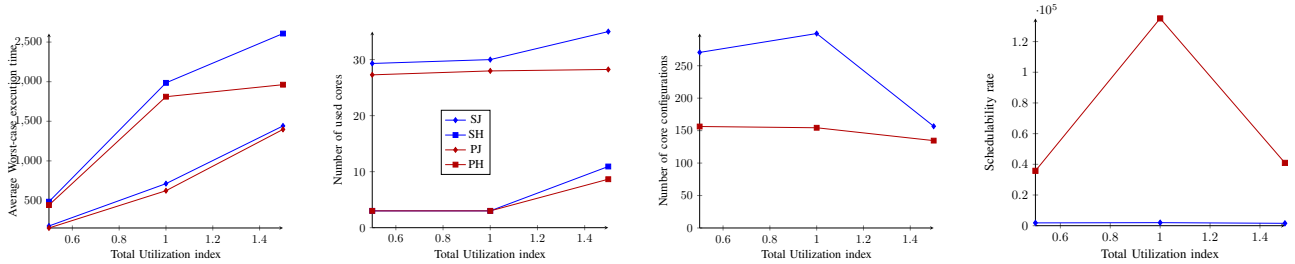
The task set generation process takes as input a tag utilization for each tag on the platform. Initially, we generate the utilization of the n tasks using the *UUniFast-Discard* algorithm for each input utilization (n is set to 4, which is typically the number of specification tasks in industry [2]). Graph sub-tasks can be executed in parallel, allowing task utilization to exceed 1. The sum of every per-tag utilization is constrained by a fixed number upper-bounded by the number of engines per tag. For the first set of experiments, we evaluate the performance of a single specification task, thus setting the input task specification utilization to 0.5, 1, and 1.5. Each task comprises a random number of sub-tasks ranging from 10 to 30. We define a probability p to denote the chance of having an edge between two nodes, and generate the edges accordingly. We ensure that the graph depth is bounded by an integer d proportional to the number of sub-tasks in the task. Additionally, we guarantee that the graph is *weakly connected* (i.e., the corresponding undirected graph is connected); if necessary, we introduce edges between non-connected portions of the graph. Given a sub-task node, one of its successors is either an alternative node or a conditional node with a probability of 0.7. The deadline of every task is randomly generated from values within the range $[1000, 2000]$. The length of each scheduling frame is defined as the largest deadline. For every sub-task, we randomly select a tag. Furthermore, for each tag, we utilize the UUniFAST-Discard algorithm again to generate individual sub-task utilizations. Hence, the sub-tasks' utilization can never exceed 1. We multiply the utilization of each sub-task by the task deadline to derive the vertex execution time.

B. Simulation results and discussions

The results of the first set of experiments are presented in Figure 6. Each point represents the average value of 100 executions. We explored various combinations of different approaches proposed in this paper, each represented by two letters. If the first letter of the combination is S, it signifies selecting the core configuration that enables the shortest WCRT. If the first letter is P, it indicates the core configuration with the smallest number of cores that allows a WCRT less than or equal to the task deadline. The second letter of each approach can be J, representing the Jeffay WCRT, or H, denoting the Han et al. WCRT.

In Figure 6a, we present the WCRT as a function of total utilization. The bound by Han et al. allows for shorter WCRTs than the Jeffay et al. bound across all combinations. This dominance is also evident when considering the number of used cores (refer to Figure 6b) for all combinations. The number of used cores is a crucial parameter when addressing the temporal and spatial allocation problem, which will be evaluated in the second set of experiments. The Han et al. bound enables the selection of fewer schedulable core configurations, as depicted in Figure 6c. By better identifying interference, it requires fewer cores for the same level of performance compared to the Jeffay approach. Consequently, it selects platforms that likely dominate others, resulting in a *smaller* Pareto front. This aspect is critical when exploring optimal spatial and temporal allocation. The optimality of selecting the core configuration is not explicitly addressed in this paper. Han et. al bound performance advantages has a large cost. The complexity of the Han et al. WCRT is significantly higher than that of the Jeffay et al. bound. Achieving the analysis using the Han approach can require up to ten times the magnitude (See Figure 6d) of time required for the Jeffay et al. analysis.

The second set of experiments, in which the Pareto front of scheduling configurations is constructed, is depicted in Figure 7. This figure illustrates the average number of tasks per Pareto front as a function of the utilization of a single task specification for the *breadth heuristic*, *depth heuristic*, and *scarce resources* approaches, using Han et al. bound, which dominate Jeffay et al. bound. Please note that our goal is to schedule the maximum workload on the platform. The effectively scheduled workload is bigger than the values indicated on the x-axis, which represent only



(a) Worst-case response time (b) number of cores. (c) Size of pareto. (d) Required anaylsis time.

Fig. 6: Results of the first set of expirements

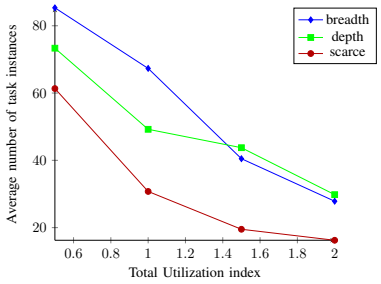


Fig. 7: Average number of tasks per pareto front

the upper bound of a single task specification's utilization. When the workload per task specification is low, the breadth heuristic outperforms all others. This is because it utilizes few resources due to the small workload, and it selects the shortest WCRT thus achieving efficient temporal and spatial isolation, enabling the execution of more tasks on average compared to the other approaches. However, as the workload increases, the depth heuristic slightly outperforms the breadth approach. The breadth heuristic typically selects a larger sub-platform to reduce worst-case response times, thereby significantly reducing spatial resource partitioning, while the depth approach increase spatial partitionning by selecting the smallest schedulable subplatform. In all scenarios, these approaches outperform the scarce resources approach, which is designed to perform well in the presence of scarce and efficient resources. However, since the platform under test does not have scarce resources, it exhibits poorer performance. It is worth noting that in scenarios with scarce resources, the scarce resources approach performs better than both others, although these results are not included here due to space constraints.

VII. Conclusions and Future Work

This paper introduces a novel scheduling approach tailored to the dynamic nature of BTS real-time software. It addresses real-time task allocation across heterogeneous many-core platforms. By leveraging HPC-DAG task templates and offline scheduling, it enables efficient processing of diverse user requests within a single scheduling frame. Our approach simplifies online scheduler design while ensuring the respect of the timing constraints of tasks. Through extensive experiments, we validates the effectiveness of the proposed approaches. As part of our future work, we aim to relax the constraint that all instances within the same frame must select the same alternative. Additionally, we plan to model this problem as an Integer Linear Program to further leverage an optimal solution, particularly when dealing with a small set of tasks. While the problem is inherently complex, the most time-consuming aspect lies in assessing the sub-platform size and selecting the best one while respecting schedulability constraints. Precomputing certain aspects of the ILP choices can potentially reduce the need for extensive design space exploration. *This work has been supported in Part by HeRITAGES ANR Project and by PHC-Tassili program.*

References

[1] L. Van der Perre, L. Liu, and E. G. Larsson, "Efficient dsp and circuit architectures for massive mimo: State of the art and future directions," *IEEE Transactions on Signal Processing*, vol. 66.

[2] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan, *Physical layer multi-core prototyping: A dataflow-based approach for LTE eNodeB*. Springer, 2013, vol. 171.

[3] O. Sinnen, *Task scheduling for parallel systems*. John Wiley & Sons, 2007, vol. 60.

[4] A. Rădulescu and A. J. Van Gemund, "On the complexity of list scheduling algorithms for distributed-memory systems," in *13th international conference on Supercomputing*, 1999.

[5] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," in *2013 25th Euromicro conference on real-time systems*.

[6] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 2015, pp. 211–221.

[7] H. Zahaf, G. Lipari, M. Bertogna, and P. Boulet, "The parallel multi-mode digraph task model for energy-aware real-time heterogeneous multi-core systems," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1511–1524, Oct 2019.

[8] H.-E. Zahaf, A.-E.-H. Benyamina, R. Olejnik, and G. Lipari, "Modeling parallel real-time tasks with di-graphs," in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, ser. RTNS '16. ACM, 2016, pp. 339–348.

[9] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys, volume=43, number=4, year=2011, publisher=ACM New York, NY, USA*.

[10] Z. Houssam-Eddine, N. Capodieci, R. Cavicchioli, G. Lipari, and M. Bertogna, "The hpc-dag task model for heterogeneous real-time systems," *IEEE Transactions on Computers*, vol. 70, no. 10, 2021.

[11] S. Chang, X. Zhao, Z. Liu, and Q. Deng, "Real-time scheduling and analysis of parallel tasks on heterogeneous multi-cores," *Journal of Systems Architecture*, vol. 105, p. 101704, 2020.

[12] N. Tsog, M. Becker, F. Bruhn, M. Behnam, and M. Sjödin, "Static allocation of parallel tasks to improve schedulability in cpu-gpu heterogeneous real-time systems," in *IECON 2019 - 45th Annual Conference of Industrial Electronics Society*, vol. 1, pp. 4516–4522.

[13] H.-E. Zahaf, A. E. H. Benyamina, R. Olejnik, and G. Lipari, "Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms," *Journal of Systems Architecture*, vol. 74, 2017.

[14] A. Bertout, J. Goossens, E. Grolleau, R. Jamil, and X. Poczekajlo, "Workload assignment for global real-time scheduling on unrelated clustered platforms," *Real-Time Systems*, vol. 58, no. 1, 2022.

[15] J. M. Jaffe, "Bounds on the scheduling of typed task systems," *SIAM Journal on Computing*, vol. 9, no. 3, pp. 541–551, 1980.

[16] M. Han, N. Guan, J. Sun, Q. He, Q. Deng, and W. Liu, "Response time bounds for typed dag parallel tasks on heterogeneous multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2567–2581, 2019.