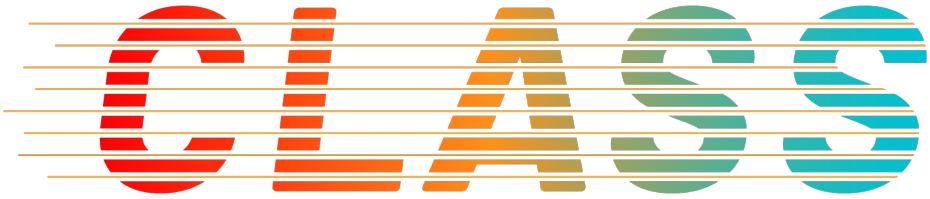

D3.4 First release of the real-time analysis methods and tools on the edge

Version 1.0

Document Information

Contract Number	780622
Project Website	https://class-project.eu/
Contractual Deadline	M15, March 2019
Dissemination Level	PU
Nature	DEC
Author(s)	Roberto Cavicchioli (UNIMORE)
Contributor(s)	
Reviewer(s)	Marta Corredoira (MAS)
Keywords	Tool - Real time - Schedulability - Heterogenous

***Notices**: The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No "780622".*

Change Log

Version	Author	Description of Change
V1.0	Roberto Cavicchioli	Complete deliverable

Contents

List of Figures

1 Introduction

Task 3.3. "Real-time analysis methods and tools on the edge". This task has developed, in collaboration with Task 3.2, the set of real-time analysis techniques deployed at both development (static) and execution (dynamic) time that will guarantee the responsiveness of big data analytics on the edge. At development time, these techniques will statically assign computing resources on the edge (e.g., CPU time, cores, accelerators time) to guarantee time predictability while ensuring the right level of performance. At execution time, this task will provide analysis tools (in the form of schedulability test) capable of dynamically adjusting the execution for improving performance while providing time predictable level. The target at MS2 is the static real-time analysis tools implemented at the edge.

To do that a novel model of real-time task called HPC-DAG (Heterogeneous Parallel Conditional Directed Acyclic Graph) is presented in Section 2. Thanks to the graph structure, the HPC-DAG model allows specifying the degree of parallelism of real-time sub-tasks. The designer can use special *alternative* nodes in the graph to model alternative implementations of the same functionality on different computing engines to be selected off-line, and *conditional* nodes in the graph to model if-then-else branches to be selected at run-time. Alternative nodes are used to leverage the diversity of computing accelerators within our target platform.

Then, in Section 3 we present a schedulability analysis that will be used in Section 4 by a set of allocation heuristics to map tasks on computing platforms and to assign scheduling parameters. In particular, we present a novel technique to reduce the pessimism due to high preemption costs in the analysis (Section 3.6).

After discussing related work in Section 5, our methodology is evaluated in Section 6 by comparing it with state-of-the-art algorithms through a set of synthetic experiments.

2 System model

2.1 Architecture model

A heterogeneous architecture is modeled as a set of *execution engines* $\texttt{Arch} = \{e_1, e_2, \ldots, e_m\}$. An execution engine is characterized by 1) its execution capabilities, (i.e. its Instruction Set Architecture), specified by the engine's *tag*, and 2) its scheduling policy. An engine's tag $\texttt{tag}(e_i)$ indicates the ability of a processor to execute a dedicated tasks.

As an example, a Xavier based platform such as the *NVIDIA pegasus* that will be installed in the CLASS cars, can be modeled using a total of 16 engines for a total of five different engine tags: 8 CPUs, 2 dGPUs, 2 iGPUs, 2 DLAs and 2 PVAs.

Tags express the heterogeneity of modern processor architecture: an engine tagged by dGPU (discrete GPU) or iGPU (integrated GPU) is designed to efficiently run generic GPU kernels, whereas engines with DLA tags are designed to run *deep learning inference* tasks.

Trivially, a deep learning task can be compiled to run on any engine, including CPUs and GPUs, however its worst-case execution time will be lower when running on DLAs. In this work, we allow the designer to compile the same task on different alternative engines with different tradeoffs in terms of performance and resource utilization, so to widen the space of possible solutions. As we will see in the next section, the HPC-DAG model supports *alternative* implementations of the same code. During the off-line analysis phase, only one of these alternative versions will be chosen depending on the overall schedulability of the system.

Communication is an important issue when considering the execution of real-time tasks on heterogeneous architectures. In modern GPUs, data transfers are performed by special engines called *copy engines*. A copy engine is a co-processor in charge of moving data between an address space visible to the CPU to an address space visible to the GPU. This translates in two physical separate memory devices in case of discrete GPUs, whereas for integrated GPUs system RAM is shared among both CPU cores and compute accellerators. We treat copy engines as processing units in which we *schedule* communication tasks.

Engines are further characterized by a scheduling policy (e.g. Fixed Priority or Earliest Deadline First), which can be *preemptive* or *non-preemptive*. In our model we allow different engines to support different scheduling policies: as we show in Section 3, in our methodology the schedulability analysis of each engine can be performed independently of the others. However, to simplify the presentation, in this deliverable we focus only on *preemptive EDF* for all the considered engines for now, but we plan to include other scheduling strategies for the final release.

2.2 The HPC-DAG task model

2.2.1 Specification tasks

A *specification task* is a Directed Acyclic Graph (DAG), characterized by a tuple $\tau = \{\mathsf{T}, \mathsf{D}, \mathcal{V}, \mathcal{A}, \Gamma, \mathcal{E}\}$, where: T is the period (minimum interarrival time); D is the relative deadline; $\mathcal{V}$ is a set of graph nodes that represent *sub-tasks*; $\mathcal{A}$ is a set of *alternative nodes*; and Γ is a set of *conditional nodes*. The set of all the nodes is denoted by $\mathcal{N} = \mathcal{V} \cup \mathcal{A} \cup \Gamma$. The set $\mathcal{E}$ is the set of edges of the graph $\mathcal{E} : \mathcal{N} \times \mathcal{N}$.

A sub-task $v \in \mathcal{V}$ is the basic computation unit. It represents a block of code to be executed by one of the engines of the architecture. A sub-task is characterized by:

- A tag $\mathtt{tag}(v)$ represent the ISA of the sub-task code. A sub-task can only be allocate onto an engine with the same tag;
- A worst-case execution time $C(v)$ when executing the sub-task on the corresponding engine processor.

In this work, we do not model the parallelization inside the GPU. We model a GPU node as a single sub-task able to potentially exploit all the computing

parallel resources of the GPU's execution engine. This is compliant with what have been disclosed with regards to the NVIDIA GPU application scheduler [6] [7] that assumes only one GPU context being resident within the GPU at a given time. As an example, if the GPU node is an image processing workload, parallelization is exploited at the level of pixels of the image and not by processing multiple images at the same time instant.

A conditional node $\gamma \in \Gamma$ represents alternative paths in the graph due to non-deterministic on-line conditions (e.g. if-then-else conditions). At run-time, only one of the outgoing edges of γ is executed, but it is not possible to know in advance which one.

An alternative node $a \in \mathcal{A}$ represents alternative implementations of parts of the graph/task, as introduced in the previous section. During the configuration phase (which is detailed in Section 4.1) our methodology selects one between 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. This can be useful when modeling sub-tasks than can be executed on different engines with different execution costs. In our model, the choice of where the sub-task should be executed is performed off-line by our proposed scheduling analysis and allocation strategy.

An edge $e(n_i, n_j) \in \mathcal{E}$ models a precedence constraint (and related communication) between node n_i and node n_j, where n_i and n_j can be sub-tasks, alternative nodes or conditional nodes.

The set of *immediate predecessors* of a node n_j, denoted by $\text{pred}(n_j)$, is the set of all nodes n_i such that 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 exist a path toward n_j. If a node has no predecessor, it is a *source node* of the graph. In our model we allow a graph to have several source nodes. In the same way we can define the set of *immediate successors* of node n_j, denoted by $\text{succ}(n_j)$, as the set of all nodes n_k such that there exists an edge (n_j, n_k), and the set of *successors* of n_j as the set of nodes for which there is a path from n_j. If a node has no successors, it is a *sink node* of the graph, and we allow a graph to have several sink nodes.

Conditional nodes and alternative nodes always have at least 2 outgoing edges, so they cannot be sinks. To simplify the reasoning, we also assume that they always have at least one predecessor node, so they cannot be sources.

2.2.2 Concrete tasks

A concrete task $\overline{\tau} = \{T, D, \overline{\mathcal{V}}, \overline{\Gamma}, \overline{\mathcal{E}}\}$ is an instance of a specification task where all alternatives have been removed by making implementation choices during the analysis. In the following, the *volume* $\text{vol}(\overline{\tau})$ denotes the total cumulative WCET of the concrete task. It is computed in linear time in the number of conditional vertices [3].

Before explaining how to obtain a concrete task from a specification task, we present an example.

Example 1. *Consider the task specification described in Figure 1a. Each sub-task node is labeled by the sub-task id and engine tag. Alternative nodes are denoted*

by square boxes and conditional nodes are denoted by diamond boxes. The black boxes denote corresponding junction nodes for alternatives and conditional, they are used to improve the readability of the figure but they are not part of the task specification[1].

Figure 1: Task specification and concrete tasks

Sub-tasks v_1^{CPU} and v_2^{CPU} are the sources (entry points) of the DAG. $v_1^{\text{CPU}}, v_2^{\text{CPU}}$ are marked by the CPU tag and can run cuncurrently: during the off-line analysis they may be allocated on the same or onto different engines. Sub-task v_4^{DLA} has an outgoing edge to v_5^{dGPU}, thus sub-task v_5^{dCPU} can not start its execution before sub-task v_4^{DLA} has finished its execution. Sub-tasks v_1^{CPU} and v_2^{CPU} have each one outgoing edge to the alternative node A. Thus, τ can execute either:

1. *By following v_3^{dGPU} and then v_4^{DLA},v_5^{dGPU} and finishing its instance on v_8^{CPU};*
2. *Or by following the conditional node F and select, according to an undetermined condition evaluated on-line, either to execute v_6^{DLA} or v_7^{dGPU}, and finishing its instance on v_8^{CPU}.*

The two patterns are alternative ways to execute the same functionalities at different costs.

Figure 1b represents one of the concrete tasks of τ_i. During the analysis, alternative execution $(v_3^{dGPU}, v_4^{DLA}, v_5^{dGPU})$ has been dropped.

In our model, data transfers between tasks can be modeled by special sub-tasks tagged with tag CP (*Copy-Engine*).

We consider a *sporadic task* model, therefore parameter T represents the minimum inter-arrival times between two instances of the same concrete task. When an instance of a task is activated at time t, all source sub-tasks are simultaneously activated. All subsequent sub-tasks are activated upon completion of their predecessors, and sink sub-tasks must all complete no later than time $t + \text{D}$. We assume *constrained deadline tasks*, that is $\text{D} \leq \text{T}$.

We now present a procedure to generate a concrete task $\overline{\tau}$ from a specification task τ, when all alternatives have been chosen. The procedure starts by initializing $\overline{\mathcal{V}} = \emptyset$, $\overline{\Gamma} = \emptyset$. First, all the source sub-tasks of τ are added to $\overline{\mathcal{V}}$. Then,

[1] In fact, it is not always possible to insert junction nodes for an arbitrary specification.

for every immediate successor node n_j of a node $n_i \in \{\overline{\mathcal{V}} \cup \overline{\Gamma}\}$: if n_j is a sub-task node (a conditional node, respectively), it is added to $\overline{\mathcal{V}}$ (to $\overline{\Gamma}$, respectively); if it is an alternative node, we consider the selected immediate successor n_k of n_j and we add it to $\overline{\mathcal{V}}$ or to $\overline{\Gamma}$, respectively. The procedure is iterated until all nodes of τ have been visited. The set of edges $\overline{\mathcal{E}} \subseteq \mathcal{E}$ is updated accordingly.

We denote by $\Omega(\tau)$ the set of all concrete tasks of a specification task τ. $\Omega(\tau)$ is generated by simply enumerating all possible alternatives.

3 Scheduling analysis

In this work, we consider partitioned scheduling. Each engine has its own scheduler and a separate ready-queue. Sub-tasks are allocated (partitioned) onto the available engines so that the system is schedulable. Partitioned scheduling allows to use well-known single processor schedulability tests which make the analysis simpler and allow us to reduce the overhead due to thread migration compared to global scheduling. The analysis presented here is modular, so engines may have different scheduling policies. As already stated begore, we restrict to preemptive-EDF for now but the analysis can be done with any other scheduling policy.

3.1 Alternative patterns

Given a specification task τ, we have to select one of the possible concrete tasks before proceeding to the allocation and scheduling of the sub-tasks on the computing engine. Since the number of combinations can be very large, in this work we propose an heuristic algorithm based on a *greedy* strategy (see Section 4). In particular, we explore the set of concrete tasks in a certain order. The order relation $\succ$ sorts concrete tasks according to their total execution time.

Definition 1. *Let $\overline{\tau}', \overline{\tau}''$ be two concrete tasks of specification task τ*
The partial order relation $\succ$ is defined as:

$$\overline{\tau}' \succ \overline{\tau}'' \implies \text{vol}(\overline{\tau}') \geq \text{vol}(\overline{\tau}'') \tag{1}$$

In the next section, we will define a second order relationship $\gg$ that sorts concrete tasks based on their engine tags.

3.2 Tagged Tasks

One concrete task may contain sub-tasks with different tags which will be allocated on different engines. Before proceeding to allocation, we need to select only sub-tasks pertaining to a given tag. We call this operation *task filtering*.

We start by defining an *empty sub-task* as a sub-task with null computation time.

Definition 2 (Tagged task)**.** *Let $\overline{\tau} = \{\mathsf{T}, \mathsf{D}, \overline{\mathcal{V}}, \overline{\Gamma}, \overline{\mathcal{E}}\}$ be a concrete task. Task $\overline{\tau}(\text{tag}_i)$ is a* tagged task *of $\overline{\tau}$ iff*

- $\overline{\tau}(\text{tag}_i) = \{\text{T}, \text{D}, \mathcal{V}_i, \Gamma_i, \mathcal{E}_i\}$ *is isomorphic to* $\overline{\tau}$*, that is the graph has the same structure, the same number of nodes of the same type, and the same edges between corresponding nodes;*
- *let* $v \in \overline{\mathcal{V}}$ *be a sub-task of* $\overline{\tau}$*, and let* $v' \in \mathcal{V}_i$ *be the corresponding sub-task of* $\overline{\tau}(\text{tag}_i)$ *in the isomorphism. If* $\text{tag}(v) = \text{tag}_i$*, then* $C(v') = C(v)$*, else* $C(v') = 0$*;*
- $\Gamma_i = \overline{\Gamma}$*.*

We denote with $\mathcal{S}(\overline{\tau}) = \{\overline{\tau}(\text{tag}_1), \ldots \overline{\tau}(\text{tag}_K)\}$ *the set of all possible tagged tasks of* $\overline{\tau}$*.*

Each concrete task generates as many *tagged tasks* as there are tags in the target architecture.

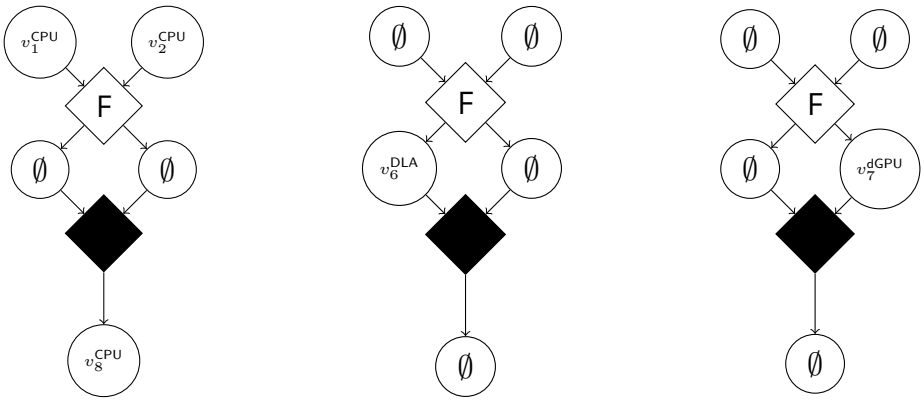

Figure 2: Tagged tasks for the concrete task of Figure

Figure 2 shows the three tagged tasks for the concrete task in Figure 1b. The first one contains only sub-tasks having CPU tag, the second contains only DLA sub-tasks, and the third one refers to GPU sub-tasks. Every tagged task will be allocated on one or more engines having the corresponding tag.

Definition 3 ($\gg$ order relationship)**.** *Assume the architecture supports* K *different tags. Let* $n(\text{tag})$ *denote the number of computing engines labeled with* tag*. Assume that tags are ordered by increasing* $n(\text{tag})$*, that is* $n(\text{tag}_i) < n(\text{tag}_j) \implies i < j$*.*

Let $\overline{\tau}', \overline{\tau}''$ *be two concrete tasks of specification task* τ*, and let* $\mathcal{S}(\overline{\tau}') = \{\overline{\tau}'(\text{tag}_1), \ldots, \overline{\tau}'(\text{tag}_K)\}$ *and* $\mathcal{S}(\overline{\tau}'') = \{\overline{\tau}''(\text{tag}_1), \ldots, \overline{\tau}''(\text{tag}_K)\}$ *be the respective tagged tasks.*

The order relation $\overline{\tau}' \gg \overline{\tau}''$ *is defined as follows:*

$$\overline{\tau}' \gg \overline{\tau}'' \implies$$
$$\exists\, 0 \leq i \leq K \begin{cases} \text{vol}(\overline{\tau}'(\text{tag}_j)) = \text{vol}(\overline{\tau}''(\text{tag}_j)) & \forall j < i \\ \text{vol}(\overline{\tau}'(\text{tag}_i)) < \text{vol}(\overline{\tau}''(\text{tag}_i)) \end{cases}$$

Relationship $\gg$ gives priority to concrete tasks that allocate less load on scarce resources: if there are few execution engines with a certain tag, and there is a large number of sub-tasks requiring allocation on that specific engine, the relation order prefers alternative patterns with lower workload for those engines.

3.3 Deadlines and offsets assignment

Meeting timing constraints of a concrete task depends on the allocation of the sub-tasks onto the different execution engines. As these sub-tasks communicate through shared buffers, they are forced to respect the execution order dictated by the precedence constraints imposed by the graph structure.

To reduce the complexity of dealing with precedence constraints directly, we impose intermediate offsets and deadlines on each sub-task. In this way, precedence constraints are respected automatically if every sub-task is activated after its offset and it completes no later than its deadline.

Many authors have proposed techniques to assign intermediate deadlines and offsets to task graphs. Here we use techniques similar to those proposed in [13] and [19].

Most of the deadline assignment techniques are based on the computation of the execution time of the critical path. A path $P_x = \{v_1, v_2, \cdots, v_l\}$ is a sequence of sub-tasks of task $\overline{\tau}$ such that:

$$\forall v_l, v_{l+1} \in P_x, \exists e(v_l, v_{l+1}) \in \mathcal{E}.$$

Let $\mathcal{P}$ denote the set of all possible paths of task $\overline{\tau}$. The critical path $P_{crit}(\overline{\tau}) \in \mathcal{P}$ is defined as the path with the largest cumulative execution time of the sub-tasks.

We define the slack $\mathsf{Sl}(P, \mathsf{D})$ along path P as:

$$\mathsf{Sl}(P, \mathsf{D}) = \mathsf{D} - \sum_{v \in P} \mathsf{C}(v)$$

The assignment algorithm starts by assigning an intermediate relative deadline to every sub-task along a path by distributing the path's slack as follows:

$$\mathsf{D}(v) = \mathsf{C}(v) + \mathsf{calculate\_share}(v, P)$$

The calculate_share function computes the slack for sub-task v along the path. This slack can be shared according to two alternative heuristics:

- **Fair distribution:** assigns slack as the ratio of the original slack by the number of sub-tasks along the path:

$$\mathsf{calculate\_share}(v, P) = \frac{\mathsf{Sl}(P, \mathsf{D})}{|P|} \tag{2}$$

- **Proportional distribution:** assigns slack according to the contribution of the sub-task execution time in the path:

$$\mathsf{calculate\_share}(v, P) = \frac{\mathsf{C}(v)}{\mathsf{C}(P)} \cdot \mathsf{Sl}(P, \mathsf{D}) \tag{3}$$

Once the relative deadlines of the sub-tasks along the critical path have been assigned, we can select the next path in order of decreasing cumulative execution time, and assign the deadlines to the remaining sub-task by appropriately

subtracting the already assigned deadlines. The complete procedure has been described in [19], and due to space constraints we do not report it here.

Let $\mathsf{O}(v)$ be the offset of a subtask with respect of the arrival time of the task's instance. The sum of the offset and of the intermediate relative deadline of a subtask is called *local deadline* $\mathsf{O}(v) + \mathsf{D}(v)$, and it is the deadline relative to the arrival of the task's instance.

The offset of a subtask is set equal to 0 if the subtask has no predecessors; otherwise, it can be computed recursively as the maximum between the local deadlines of the predecessor sub-tasks.

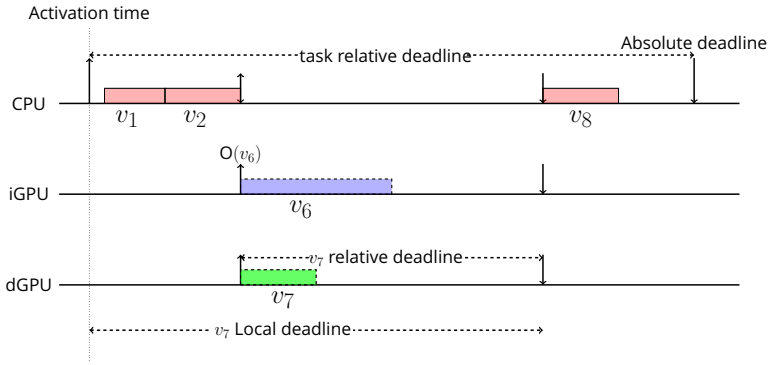

Figure 3: Example of offset and local deadline

Figure 3 illustrates the relationship between the activation times, the intermediate offsets, relative deadlines and local deadlines of the sub-tasks of the concrete task of Figure 1b. We assume that v_1, v_2, v_8 have been allocated on the same CPU whereas v_6 and v_7 each on a different engine. The activation time is the absolute time of the arrival of the sub-task instance. The activation time of a source sub-task corresponds to the activation time of the task graph. The offset is the interval between the activation of the task graph and the activation of the sub-task. The local deadline is the interval between the task graph activation and the sub-task absolute deadline.

Definition 4. *Sub-task $v \in \overline{\mathcal{V}_\tau}$ is feasible if for each task instance arrived at a_j, sub-task v executes within the interval bounded by its arrival time $a(v) = a_j + \mathsf{O}(v)$ and its* absolute *deadline $a(v) + \mathsf{D}(v)$.*

Lemma 1. *A concrete task (resp. tagged task) is feasible if all its sub-tasks are feasible.*

Proof. By the definition, the local deadline of the sink sub-tasks is equal to the deadline of the task D. Moreover, the offset of a sub-task is never before the local deadline of a preceding sub-task. Therefore 1) the precedence constraints are respected and 2) if sink sub-tasks are feasible then the concrete task (tagged task, respectively) is feasible. □

3.4 Single engine analysis

In this section, we assume that sub-tasks have been already been assigned offsets and deadlines, and they have been allocated on the platform's engines, and

we present the schedulability analysis to test if all tasks respect their deadlines when scheduled by the Earliest Deadline First (EDF) algorithm.

Theorem 1. *Let $\mathcal{T}$ a set of task graphs allocated onto a single-core engine. Task set $\mathcal{T}$ is schedulable by EDF if and only if:*

$$\sum_{\overline{\tau} \in \mathcal{T}} \mathsf{dbf}(\overline{\tau}, t) \leq t, \forall t \leq t^* \tag{4}$$

The dbf is the demand bound function [4] for a task graph $\overline{\tau}$ in interval t. The demand bound function is computed as the worst-case cumulative execution time of all jobs (instances of sub-tasks) having their arrival time and deadline within any interval of time of length t. For a task graph, the dbf can be computed as follows:

$$\mathsf{dbf}(\tau, t) = \max_{v \in \tau} \sum_{v' \in \tau} \left\lfloor \frac{t - \tilde{O}(v') - \mathsf{D}(v') + \mathsf{T}(\tau)}{\mathsf{T}(\tau)} \right\rfloor \mathsf{C}(v') \tag{5}$$

where[2]:

$$\tilde{O}(v') = (\mathsf{O}(v') - \mathsf{O}(v)) \bmod \mathsf{T}(\tau)$$

In our model, a task graph may contain *conditional nodes*, which model alternative paths that are selected non-deterministically at run-time. To compute the dbf for a tagged task that contains conditional nodes, we must first enumerate all possible conditional graphs by using the same procedure as the one used for generating concrete tasks from specification tasks. Hence, the dbf of a tagged task in interval t can be computed as the largest dbf among all the possible conditional graphs.

3.5 Anticipating the activation of sub-tasks

Given an instance of sub-task v with arrival at $a(v)$ and local deadline at $\mathsf{D}(v)$, at run-time it may happen that all instances of the preceding sub-tasks have already completed their execution before $a(v)$. In this case, we activate the sub-task as soon as the preceding sub-tasks have finished *with the same local deadline* $\mathsf{D}(v)$.

Lemma 2. *Consider a feasible set of sub-tasks allocated on a set of engines and scheduled by EDF. If a sub-task is activated as soon as all predecessor sub-tasks have finished, with the same local deadline, the set remains schedulable.*

Proof. Descends directly from the sustainability property of EDF [5]. In fact, by anticipating the activation of the sub-task without modifying its local deadline, the sub-task will be scheduled with a longer relative deadline, and the demand bound function will not increase. □

From an implementation point of view, this technique avoids the need to set-up activation timers for intermediate tasks; moreover, it allows us to reduce the pessimism of the analysis in the presence of high preemption costs, as we will see in the next section.

[2]We remind that the remainder of a/b is by definition a positive number r such that $a = kb + r$.

3.6 Preemption-aware analysis

In recent GPUs, preempting an executing task can be a costly operation (see Section 6.3). In particular, the cost of preemption may significantly vary depending on the preempted task and the engine. For example, preempting a graphical kernel induces a larger cost compared to preempting a computing CUDA kernel. Therefore, we need to account for the cost of preemption in the analysis.

We start by observing that, in the case of EDF scheduling, a job of a sub-task v_i can preempt a job of sub-task v_j at most once, and only if its relative deadline is shorter: $D(v_i) < D(v_j)$.

A simple (although pessimistic) approach is to always consider the worst-case preemption cost as part of the worst-case computation time of the preempting task. Let $\mathsf{pc}(v_j)$ denote the cost of preempting sub-task v_j.

Lemma 3. *Let $\mathcal{V} = \{v_1, v_2, \cdots, v_K\}$ be a set of sub-tasks to be scheduled by EDF on a single engine.*

Consider $\mathcal{V}^{\mathsf{pc}} = \{v'_1, v'_2, \cdots, v'_K\}$, where v'_i has the same parameters as v_i, except for the wcet that is computed as $\mathsf{C}(v'_i) = \mathsf{C}(v_i) + \mathsf{pc}^i$ and $\mathsf{pc}^i = \max\{\mathsf{pc}(v) | v \in \mathcal{V} \wedge D(v) > D(v_i)\}$.

If $\mathcal{V}^{\mathsf{pc}}$ is schedulable by EDF when considering a null preemption cost, then $\mathcal{V}$ is schedulable when considering the cost of preemption.

Proof. The Lemma directly follows from the simple observation that the cost of preemption can never exceed pc^i for sub-task v_i. □

Lemma 3 is safe but pessimistic. We can further improve the analysis by observing that a sub-task cannot preempt another sub-task belonging to the same task graph (we remind the reader that we assume constrained deadline tasks). Furthermore, it may be impossible for two consecutive sub-task of a task graph to both preempt the same sub-task as demonstrated by Theorem 2.

Definition 5 (Maximal sequential subset)**.** *A* maximal sequential subset $\mathcal{V}^M$ *of task τ is a maximal subset of $\mathcal{V}_\tau$ such that:*

1. *it is weakly-connected;*
2. *$\forall v \in \mathcal{V}^M$, $v' \in \mathsf{pred}(v)$ is either null and does not belong to $\mathcal{V}^M$, or non null and belongs to $\mathcal{V}^M$.*

We denote by $\mathsf{cand}(\mathcal{V}^M)$ the set of all sub-tasks in $\mathcal{V}^M$ that are either sources, or have a null predecessor. Further, we denote by v^M the sub-task with the shortest local deadline in $\mathsf{cand}(\mathcal{V}^M)$

We observe that, since all the sub-tasks in $\mathcal{V}^M$ are allocated on the same engine and since they do not have any predecessor sub-task allocated on a different engine (no empty predecessor), they can be activated as soon as the predecessor sub-tasks have finished.

Now, suppose $v_1, v_2 \in \mathcal{V}^M$ and that v_1 is an immediate predecessor of v_2. If v_1 preempts a sub-task v_j, and $D(v_2) \leq D(v_j)$, then v_j can be executed only after v_2 has finished. This means that the cost of preempting v_j can be accounted

to only v_1. We assign this preemption cost to the sub-task v^M with the shorter local deadline among all sub-tasks not having a predecessor in $\mathcal{V}^M$, whereas the others do not pay any preemption cost. The preemption cost of any other sub-task in $\mathcal{V}'$ is set equal to 0. For all sub-tasks that have a null predecessor, we compute a preemption cost as in Lemma 3.

Theorem 2 (Limited preemption cost)**.** *Let $\mathcal{V} = \{v_1, v_2, \cdots, v_K\}$ be a set of sub-tasks scheduled by to EDF on a single processor. Consider $\mathcal{V}^{\mathsf{pc}} = \{v'_1, v'_2, \cdots, v'_K\}$ where v'_i has the same parameters as v_i, except for the wcet that is computed as $\mathsf{C}(v'_i) = \mathsf{C}(v_i) + \mathsf{pc}^i$, and pc^i is computed as in Equation* (6) *or* (7)*.*

- *If $v_i = v^M$, then*

$$\mathsf{pc}^i = \max\{\mathsf{pc}(v) | v \in \mathcal{V} \setminus \mathcal{V}_\tau \wedge D(v) > D(v_i)\}; \tag{6}$$

  where $\mathcal{V}_\tau$ is the set of sub-tasks of task τ where v_i belongs.

- *otherwise,*

$$\mathsf{pc}^i = 0 \tag{7}$$

If $\mathcal{V}^{\mathsf{pc}}$ is schedulable by EDF when considering a null preemption cost, then $\mathcal{V}$ is schedulable when considering the cost of preemption.

Proof. We report here a proof sketch.

Consider any sub-task $v_i \in \mathcal{V}^M$ not belonging to $\mathsf{cand}(\mathcal{V}^M)$: it is activated as soon as the preceding sub-tasks have finished executing their corresponding instances. Then, if one of the preceding task of v_i preempted a sub-task v_j, the preemption cost has already been accounted in the worst-case execution time of the preceding task; as discussed above v_j can only resume execution after v_i has completed. Thus, no further preemption cost need to be accounted.

If instead none of the preceding sub-task of v_i has preempted v_j, then v_j cannot start executing before v_i completes because its deadline is not smaller than $D(v_i)$, hence no preemption will occur.

In any case, no cost of preemption needs to be accounted for to v_i.

Similarly, sub-tasks belonging to $\mathsf{cand}(\mathcal{V}^M)$ and different than v^M are not subject to preemptions. □

4 Allocation

4.1 Allocation of task specifications

The goal of our methodology is to allocate a set of task specifications into a set of engines, by selecting alternative implementations, so that all tasks complete before their deadlines. From a operational point of view, is is equivalent to finding a feasible solution to a complex Integer Linear Programming problem. In facts, given the large number of combinations (due to alternative nodes, condition-control nodes, and allocation decisions), an ILP formulation of this

problem fails to produce feasible solutions in an acceptable short time. Therefore, in this section we propose a set of greedy heuristics to quickly explore the space of solutions.

Algorithm 1 describes the basic methodology of our approach. The algorithm can be customised with four parameters: oder is the sorting order of the concrete task sets (see Sections 3.1 and 3.2); parameter slack concerns the way the slack is distributed when assigning intermediate deadlines and offsets (see Section 3.3); parameter alloc can be best-fit (BF) or worst-fit (WF); parameter omit concerns the strategy to eliminate sub-tasks when possible (see Section 4.3).

At each step, the algorithm tries to allocate one single task specification (for loop at line 3). For each task, it first generates all concrete tasks (line 4), and sorts them according to one relationship order ($\succ$ or $\gg$). Then, for each concrete task, if first assigns the intermediate deadlines and offsets according to the methodology described in Section 3.3 (line 9), using one between the fair or the proportional slack distributions. Then, it separates the concrete tasks into tagged tasks according to the corresponding tags (line 10).

Then, the algorithm tries to allocate every tagged task onto single engines having the corresponding tag (line 14) (this procedure is described below in Algorithm 2). If a feasible allocation is found, the allocation is generated, and the algorithm goes to the next specification task (lines 15-16). If no feasible sequential allocation can be found, the next concrete task is tested.

The algorithm gives priority to single-engine allocations because they reduce preemption cost, as discussed in Section 3.6. In particular, by allocating an entire tagged task onto a single engine, we reduce the number of null sub-task to the minum necessary, and so we can assign the cost of preemption to fewer sub-tasks.

If none of the concrete tasks of a specification task can be allocated (line 17), this means that one of the tagged tasks could not be allocated on a single engine. Therefore, the algorithms tries to break some of the tagged tasks of a concrete task into parallel tasks to be executed on different engines of the same type. This is performed by procedure parallelize, which will be described in Section 4.3. In particular, one part of the concrete task will be allocated, while the second part will be put back in the list of not-yet-allocated task graphs (line 24).

If also this process is unable to find a feasible concrete task, the analysis fails (line 29).

4.2 Sequential allocation

Algorithm 2 tries to allocate a concrete task on a minimal number of engines. It takes as input a set of tagged tasks. For each tagged task, it selects the corresponding engines, and sorts them according to the alloc parameter, that is in decreasing order of utilization in the case of Best-Fit, or in increasing order of utilization in case of Worst-Fit. Then, it tests the feasibility of allocating the tagged task on each engine in turn. If the allocation is successful, the next tagged task is tested, otherwise the algorithm tries the next engine. If the tagged task

cannot be allocated on any engine, the algorithm fails. If all tagged tasks have been allocated, the corresponding allocation is returned.

4.3 Parallel allocation

When the sequential allocation fails for a given task specification, the algorithm tries to allocate one or more of its tagged tasks onto multiple engines having the same tag. Algorithm 3 takes as input a concrete task and two parameters, alloc for BF or WF heuristics, and omit to select which sub-task to remove first.

For each tagged task of the concrete task (line 5), the algorithm selects the list of engines corresponding to the selected tag, and sorts them according to BF or WF (line 7). Then, it tries to test the feasibility of the tagged task on each engine (line 9). If the test fails, it removes one sub-task from the tagged task and adds it to list of non allocated sub-tasks $\overline{\tau}''$ (line 11). We propose two heuristics:

1. **Random** heuristic: it selects a random sub-task and adds it to the omitted list.

2. **Parallel** heuristic: to be feasible, the critical path of each tagged task must be feasible even on a unlimited number of engines. Thus, we are interested in sub-tasks that do not belong to the critical path because they are the ones causing the non-feasibility. Thus, they are omitted one by one until finding a feasible schedule.

The feasibility test is repeated until a feasible subset of $\overline{\tau}(\text{tag})$ is found. The omitted tasks are tried on the next engine with the same tag (line 16). At the end of the procedure, two concrete tasks are produced, $\overline{\tau}'$ is the feasible part that will be allocated, while $\overline{\tau}''$ will be tried again in the following iteration of Algorithm 1.

5 Related work

Many authors [11–17, 21, 22] have proposed real-time task models based on DAGs. However, to the best of our knowledge, none of the existing models supports alternative implementations of the same functionality on different computing engines.

Authors of [13] studied the deadline assignment problem in distributed real-time systems. They formalize the problem and identify the cases where deadline assignment methods have a strong impact on system performances. They propose Fair Laxity Distribution (FLD) and Unfair Laxity Distribution (ULD) and study their impact on the schedulability. In [12], authors analyze the schedulability of a set of DAGs using global EDF, global rate-monotonic (RM), and federated scheduling. In [20], the authors present a general framework of partitioning real-time tasks onto multiple cores using resource reservations. They propose techniques to set activation time and deadlines of each task, and they an use ILP formulation to solve the allocation and assignment problems. However,

when applying such approaches on large applications consisting of hundred of sub-tasks, the analysis can be highly time consuming.

DAG fixed-priority partitioned scheduling has been presented in [11]. The authors propose methods to compute a response time with tight bounds. They present partitioned DAGs as a set of self-suspending tasks, and proposed an algorithm to traverse a DAG and characterize the worst-case scheduling scenario.

Unlike previous models, Melani et al [14] proposed to model conditional branches in the code in a way similar to our conditional nodes, however their model is not able to express off-line alternative patterns. They proposed different methods to compute an upper-bound on the response-time under global scheduling algorithms. In [18], alternative on-line execution patterns can be expressed using *digraph*s. However, the digraph model cannot express parallelism and only supports sequential tasks.

In this work we assume preemptive EDF scheduling. Typically, preemption on classical CPUs can be assumed to be a negligible percentage of the task execution. However, this is not always the case with GPUs processors. Depending on the computing architecture and on the nature of the workload, GPU tasks present different degrees of preemption granularity and related preemption costs. Initial work on preemptive scheduling on GPUs assumed preemption was viable at the *kernel* granularity [23]. A finer granularity for computing workloads is represented by CTA (Cooperative Thread Array) level preemption, hence, preemption occurs at the boundaries of group of parallel threads that execute within the same GPU computing cluster [2, 8]. In such a scenario, the cost of preempting an executing context on a GPU might present significant differences as it will involve saving and restoring contexts of variable size and/or reaching the next viable preemption point. Overhead measurements operated in the cited contributions calls for modeling each GPU sub-task with a specific non-negligible preemption cost that can be in the same order of magnitude of the execution time of the sub-task.

6 Results and discussions

In this section, we evaluate the performance of our scheduling analysis and allocation strategies. We compare against the model cp-DAG proposed by Melani et al. [14]. Please notice that in [14] the authors proposed an analysis for cp-DAGs in the context of global scheduling, whereas our analysis is based on partitioned scheduling. Therefore, we extended the cp-DAG model to support multiple engines by adding a randomly selected tag to each node of the graph. Moreover we applied the same allocation heuristics of Section 4 and the same scheduling analysis of Section 3 to HPC-DAGs and to cp-DAG.

In the following experiments, we considered the NVIDIA Jetson AGX Xavier[3]. It features 8 CPU cores, and four different kinds of accelerators: one discrete and one integrated GPU, one DLA and one PVA. Each accelerator is treated as a single computing resource. In this way, we can exploit task level parallelism

[3] `https://elinux.org/Jetson_AGX_Xavier`

as opposed to allowing the parallel execution of more than one sub-task to partitions of the accelerators (e.g: at a given time instant, only one sub-task is allowed to execute in all the computing clusters of a GPU).

6.1 Task Generation

We apply our heuristics on a large number of randomly generated synthetic task sets.

The task set generation process takes as input an engine/tag utilization for each tag on the platform. First, we start by generating the utilization of the n tasks by using the UUniFast-Discard [10] algorithm for each input utilization. Graph sub-tasks can be executed in parallel, thus task utilization can be greater than 1. The sum of every per-tag utilization is a fixed number upper bounded by the number of engines per tag.

The number of nodes of every task is chosen randomly between 10 and 30. We define a probability p that expresses the chance to have an edge between two nodes, and we generate the edges according to this probability. We ensure that the graph depth is bounded by an integer d proportional to the number of sub-tasks in the task. We also ensure that the graph is *weakly connected* (i.e. the corresponding undirected graph is connected); if necessary, we add edges between non-connected portions of the graph. Given a sub-task node, one of its successors is an alternative node or a conditional node with probability of 0.7.

To avoid untractable hyper-periods, the period of every task is generated randomly according from the list, where the minimum is 120 and the maximum is $120,000$. For every sub-task, we randomly select a tag. Further, for each tag, we use algorithm UUNIFAST discard again to generate single sub-task utilization. Thus, the sub-task utilization can never exceed 1. Further, we inflate the utilization of each sub-task by the task period to generate the worst case execution time of every vertex.

A cp-DAG is generated from a HPC-DAG by selecting one of the possible concrete tasks at random.

6.2 Simulation results and discussions

We varied the baseline utilization from 0 to the number of engines per engine tag in 16 steps. Therefore, the step size vary from one engine tag to the other: the step size is 0.5 for CPUs, and 0.0625 for the others. For each utilization, we generated a random number of tasks between 20 and 25.

The results are presented as follows. Each algorithm is described using 3 letters: (i) the first letter is either B for best fit or W for worst first allocation techniques; (ii) the second is either O for the $\succ$ order relation, or R for the $\gg$ order relation; (iii) the third character describes the deadline assignment heuristic, F for fair and P for proportional. The algorithm name may also contain either option P for the parallel allocation heuristic that eliminates parallel nodes first, or R the random heuristic which randomly selects the sub-task to remove. For Figures 4, 5, 6, 7, we run 85 simulations per utilization step.

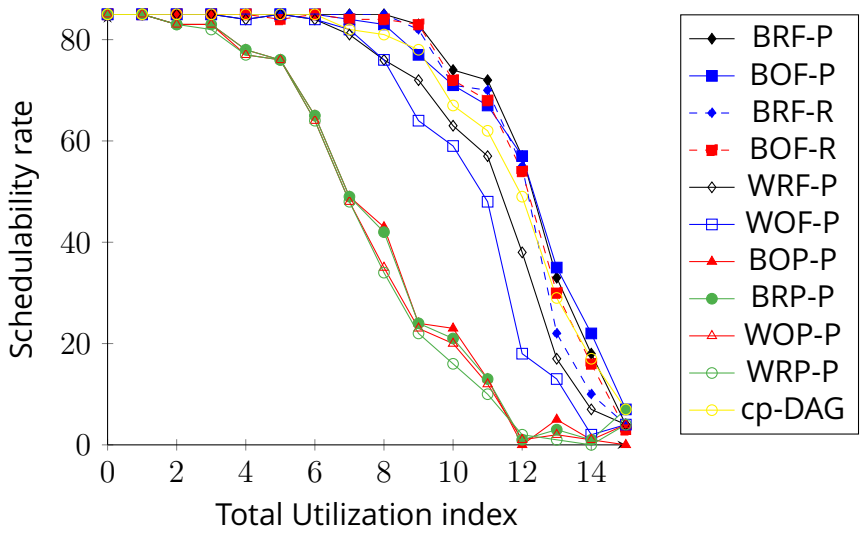

Figure 4: Schedulability rate VS total utilization.

Figure 4 represents the schedulability rate of each combination of heuristics cited above as a function of the total utilization. The fair deadline assignment technique presents better schedulability rates compared to proportional deadline assignment. In general, BF heuristic combinations outperform WF heuristic: this can be explained by observing that BF tries to pack the maximum number of sub-tasks into the minimum number of engines, and this allows for more flexibility to schedule *heavy* tasks on other engines.

In the figures, the cp-DAG model proposed in [14] is shown in yellow. Since the cp-DAG has no alternative implementations, the algorithm has less flexibility in allocating the sub-tasks, therefore *by construction* the results for HPC-DAG dominate the corresponding results for cp-DAG. However, it is interesting to measure the difference between the two models: for example in Figure 4 the difference in the schedulability rate between the two models is between 10% and 20% for utilization rates between 6 and 14.

When the system load is low, all combinations of heuristics allow having high schedulability rates. BRF shows better results because it is aimed at relaxing the utilization of scarce engines, thus avoiding the unfeasibility of certain task sets due to a high load on a scarce engines (DLA and PVA/ GPUs). However, when dealing with a highly loaded system, BOF presents better schedulability rates, as it reduces the execution overheads on all engines.

Figure 5 reports the average number of active cores (CPUs) as a function of the total utilization. WF-based heuristics always use the highest number of CPU cores because our task generator outputs at least 15 CPU subtasks. Hence, the number of tasks is larger than the available number of CPU cores (which is 8, in our test platform). BF heuristics allows to pack the maximum number of sub-tasks on the minimum number of engines, thus the utilization increases quasi-linearly. This occurs until the maximum schedulability limit is reached (i.e. number of cores). BRF heuristic uses more CPU cores because it *preserves* the scarce resources, thus it uses more CPU engines. As BOF privileges reducing the overall load, it reduces the load on the CPUs compared to BRF.

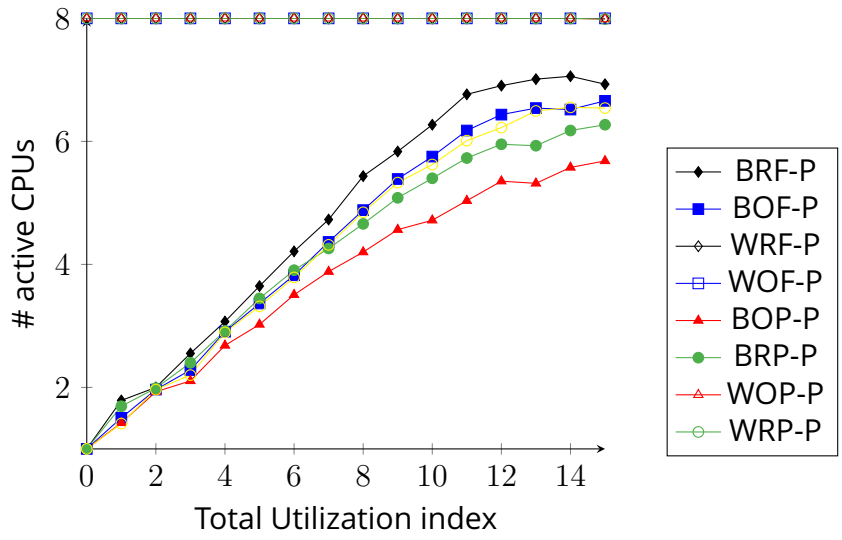

Figure 5: #Active CPUS vs total utilization.

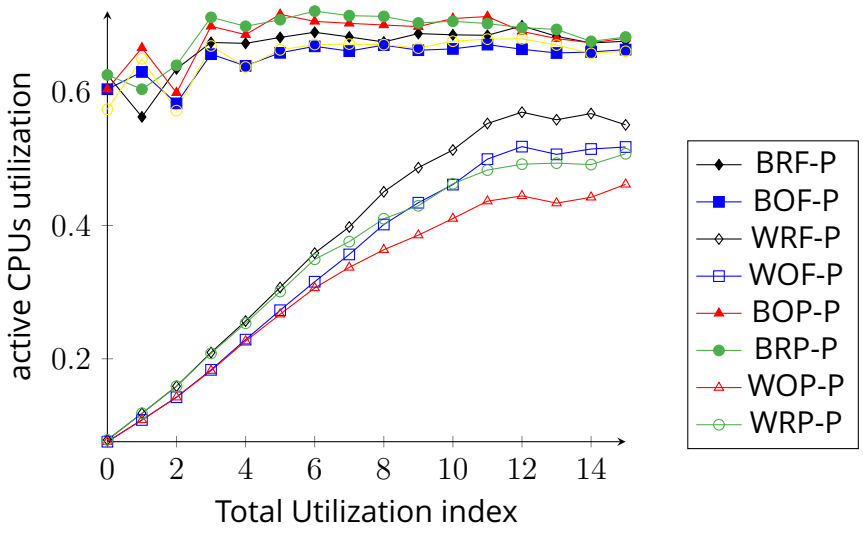

Figure 6: Active CPU utilization VS total utilization

Figure 6 shows the average active utilization for CPUs. Average utilization of BF-based heuristics is higher compared to WF. In fact, the latter distributes the work on different engines thus the per-core utilization is low in contrast to BF. Again, BRF has higher utilization than BOF because it schedules more workload on CPU cores than the other heuristics. As the workload is equally distributed on different CPUs, the WF heuristics may be used to reduce the CPUs operating frequency to save dynamic energy. Regarding BF heuristics, we see that BRF is not on the top of the average load because it uses more cores than the others.

Figure 7 shows the average utilization of the scarce resources. As you may notice, order relation $\gg$ based heuristics allows to reduce the load on the scarce resources compared to $\succ$. In fact, the higher is the load, the less loaded are the scarce resources.

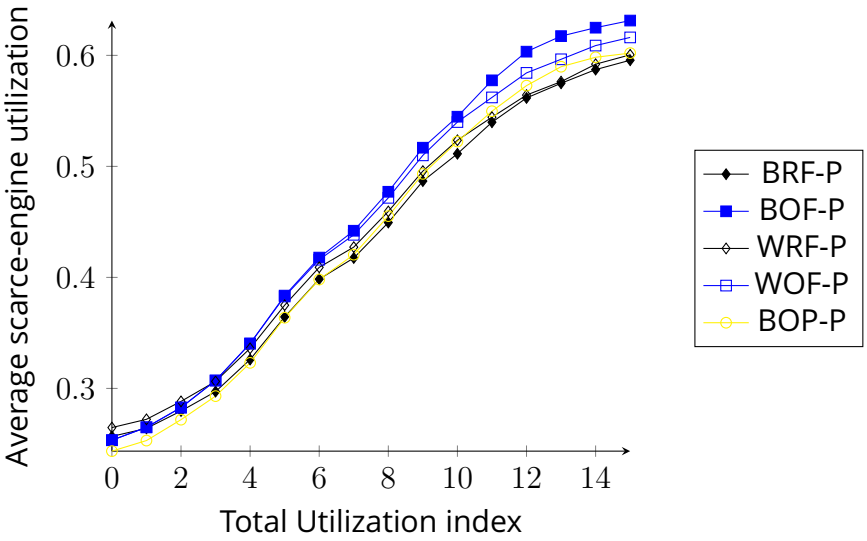

Figure 7: DLA, GPUs, PVA utilizations vs total utilization.

6.3 Preemption cost simulation

In all previous experiments, we applied the analysis described in Section 3.6 to account for preemption costs. In particular, we applied the technique of Theorem 2, by assuming that the cost of preempting a sub-task is 30% of the sub-task execution time on a GPU, 10% on DLA and PVA, and 0.02% on the CPUs. DLA and PVA are non-preemptable engines, however longer jobs might be split into smaller chunks and this translates in a splitting overhead as we submit many kernel calls as opposed of a single batch of commands.

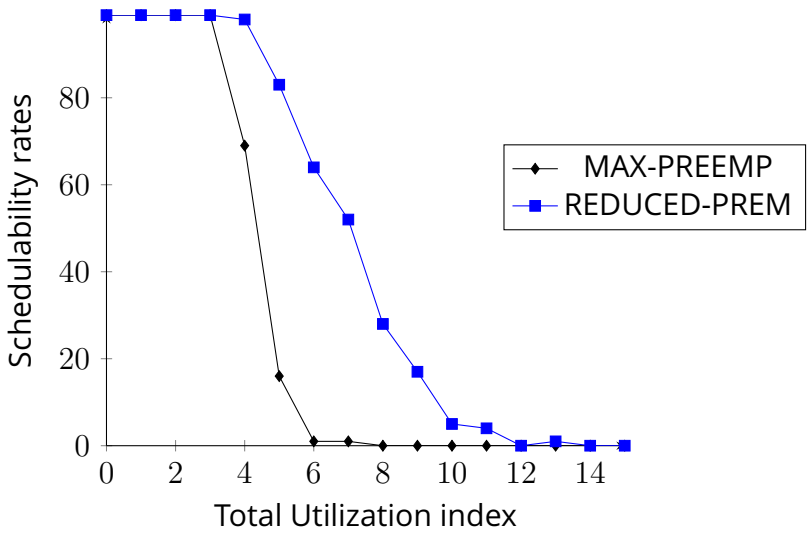

Figure 8: Preemption cost Theorem vs max

To highlight the importance of a proper analysis of the cost of preemption, in Figure 8 we report the schedulability rates obtained by BRF-P in two different cases: when considering the analysis of Lemma 3 (where the maximum preemption cost is charged to all preempting sub-tasks) and that of Theorem 2,

where the cost is only charged to one of the sub-tasks in the maximal sequential subset.

With the increase of of utilization, schedulability drastically falls for the first method, while the improved analysis of Theorem 2 keeps high schedulability rates.

7 Conclusions and future work

In this deliverable, we presented the HPC-DAG real-time task model, which allows to specify both off-line and on-line alternatives, to fully exploit the heterogeneity of complex embedded platforms. We also presented a scheduling analysis and a set of heuristics to allocate HPC-DAGs on heterogeneous computing platforms. The analysis takes into account the cost of preemption that may be non-negligible in certain specialized engines.

Results of our extensive synthetic simulations show that a significant reduction in pessimism occurs with our proposed approach. This lead to an increase in resource utilization compared to similar approaches in the literature. As for future work, we are considering extending our framework to account for memory interference between the different compute engines, as it is known to cause significant variations in execution times [1, 9] and also to include other scheduling policies.

References

[1] Waqar Ali and Heechul Yun. Protecting real-time gpu applications on integrated cpu-gpu soc platforms. *arXiv preprint arXiv:1712.08738*, 2017.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115. IEEE, 2017.

[3] Sanjoy Baruah. The federated scheduling of systems of conditional sporadic dag tasks. In *Proceedings of the 12th International Conference on Embedded Software*, pages 1–10. IEEE Press, 2015.

[4] Sanjoy K Baruah, Louis E Rosier, and Rodney R Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4), 1990.

[5] Alan Burns and Sanjoy Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97, 2008.

[6] Nicola Capodieci, Roberto Cavicchioli, and Marko Bertogna. Work-in-progress: Nvidia gpu scheduling details in virtualized environments. In *2018 International Conference on Embedded Software (EMSOFT)*, pages 1–3. IEEE, 2018.

[7] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018.

[8] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: Server based integrated gpu arbitration mechanism for memory accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 48–57. ACM, 2017.

[9] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms. In *Emerging Technologies and Factory Automation (ETFA), 2017 22nd IEEE International Conference on*, pages 1–10. IEEE, 2017.

[10] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*, 2010.

[11] José Fonseca, Geoffrey Nelissen, Vincent Nélis, and Luís Miguel Pinho. Response time analysis of sporadic dag tasks under partitioned scheduling. In *Industrial Embedded Systems (SIES), 2016 11th IEEE Symposium on*, pages 1–10. IEEE, 2016.

[12] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 85–96. IEEE, 2014.

[13] Dana Marinca, Pascale Minet, and Laurent George. Analysis of deadline assignment methods in distributed real-time systems. *Comput. Commun.*, 27(15):1412–1423, September 2004.

[14] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C. Buttazzo. Schedulability analysis of conditional parallel task graphs in multicore systems. *IEEE Trans. Computers*, 66(2):339–353, 2017.

[15] Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. Global edf scheduling of directed acyclic graphs on multiprocessor systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 287–296. ACM, 2013.

[16] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core Real-Time Scheduling for Generalized Parallel Task Models. pages 217–226, November 2011.

[17] Abusayeed Saifullah, David Ferry, Chenyang Lu, and Christopher Gill. Real-time scheduling of parallel tasks under a general dag model. 2012.

[18] M. Stigge, P. Ekberg, N. Guan, and W. Yi. The digraph real-time task model. In *Real-Time and Embedded Technology and Applications Symposium*, April 2011.

[19] Yifan Wu, Zhigang Gao, and Guojun Dai. Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations. *Journal of Systems Architecture*, 60(3):247 – 257, 2014. Real-Time Embedded Software for Multi-Core Platforms.

[20] Yifan Wu, Zhigang Gao, and Guojun Dai. Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations. *Journal of Systems Architecture*, 60(3):247–257, 2014.

[21] Houssam-Eddine Zahaf, Abou El Hassen Benyamina, Richard Olejnik, and Giuseppe Lipari. Modeling parallel real-time tasks with di-graphs. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 339–348. ACM, 2016.

[22] Houssam-Eddine Zahaf, Abou El Hassen Benyamina, Richard Olejnik, and Giuseppe Lipari. Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms. *Journal of Systems Architecture*, 74:46 – 60, 2017.

[23] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532, 2014.

Algorithm 1 Allocation algorithm

input : $\mathcal{T}$: set of task specifications
parameters : order ($\succ$ or $\gg$), slack (fair or proportional),
alloc (BF or WF), omit (parallel or random)
output : SUCCESS or FAIL
for $\tau \in \mathcal{T}$ **do**
  Ω = generate_concrete_task(τ)
  sort(Ω, order)
  for $(\overline{\tau} \in \Omega)$ **do**
    assign_deadlines_offsets($\overline{\tau}$, slack)
    $\mathcal{S}(\overline{\tau})$ = filter_tagged_task($\overline{\tau}$)
  end for
  allocated = **false**
  for $(\overline{\tau} \in \Omega)$ **do**
    if (feasible_sequential($\mathcal{S}(\overline{\tau})$, alloc)) **then**
      allocated = **true**; assign sub-tasks to engines
      break;
    end if
  end for
  if (**not** allocated) **then**
    for ($\overline{\tau} \in \Omega$) **do**
      $(\overline{\tau}', \overline{\tau}'')$ = parallelize($\overline{\tau}$, alloc, omit)
      if ($\overline{\tau}' \neq \emptyset$) **then**
        allocate $\overline{\tau}'$ to selected engines
        add back $\overline{\tau}''$ to $\mathcal{T}$
        allocated = **true**
        break
      end if
    end for
    if (**not** allocated) **then return** FAIL
  end if
end for
return SUCCESS

Algorithm 2 feasible_sequential

input: $\mathcal{S}(\overline{\tau})$: set of tagged tasks, alloc
output: feasibility: **SUCCESS** or **FAIL**
for $(\overline{\tau}(\text{tag}) \in \mathcal{S}(\overline{\tau}))$ **do**
    engine_list=select_engine(tag)
    sort_engines(engine_list, alloc)
    f = **false**
    nfeas = 0
    for $(e \in$ engine_list$)$ **do**
        f = dbf_test($\overline{\tau} \cup \mathcal{T}_e$)
        if (f) **then**
            save_allocation($\overline{\tau}$, e)
            nfeas ++
            break
        end if
    end for
    if (**not** f) **then return** FAIL;
end for
if (nfeas $= |\mathcal{S}(\overline{\tau})|$) **then**
    return SUCCESS, saved_allocations
end if

Algorithm 3 parallelize

input: $\overline{\tau}$: concrete task, alloc (BF or WF),
        omit (parallel or random)
output: concrete tasks $(\overline{\tau}', \overline{\tau}'')$
$\overline{\tau}' = \emptyset, \overline{\tau}'' = \emptyset$
for $(\overline{\tau}(\text{tag}) \in \mathcal{S}(\overline{\tau}))$ **do**
    engine_list=select_engines(tag)
    sort(engine_list, alloc)
    for $(e \in$ engine_list$)$ **do**
        f=dbf_test($\overline{\tau}(\text{tag}) \cup \mathcal{T}_e$)
        while (**not** f) **do**
            $\overline{\tau}''$ = $\overline{\tau}''\cup$ remove($\overline{\tau}(\text{tag})$, omit)
            f=dbf_test($\overline{\tau}(\text{tag}) \cup \mathcal{T}_E$)
        end while
        if ($\overline{\tau}(\text{tag}) \neq \emptyset$) **then**
            $\overline{\tau}' = \overline{\tau}'\cup$ save_allocation($\overline{\tau}(\text{tag}), e$)
            $\overline{\tau}(\text{tag}) = \overline{\tau}'', \overline{\tau}'' = \emptyset$
            allocated = **true**
            break
        end if
    end for
    if (**not** allocated) **return** $\emptyset, \overline{\tau}$
end for
return $\overline{\tau}', \overline{\tau}''$