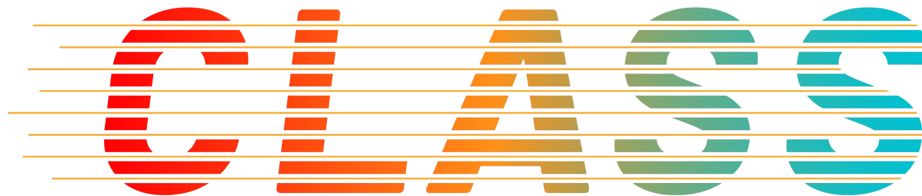
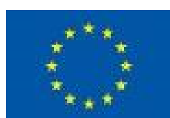


D3.5 Final release of the real-time analysis methods and tools on the edge

Version 1.0

Document Information

Contract Number	780622
Project Website	https://class-project.eu/
Contractual Deadline	M29, May 2020 (Due to COVID situation this deliverable has been submitted on M31, July 2020)
Dissemination Level	PU
Nature	DEC
Author(s)	Roberto Cavicchioli (UNIMORE)
Contributor(s)	
Reviewer(s)	Danilo Amendola (CRF)
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".*

©2018 CLASS Consortium Partners. All rights reserved.

Change Log

Version	Author	Description of Change
V0.1	Roberto Cavicchioli (UNIMORE)	First version
V0.2	Danilo Amendola (CRF)	Review
V0.3	Roberto Cavicchioli (UNIMORE)	Review comments addressed
V1.0	Maria A.Serrano (BSC)	Final version, ready to EC revision

Contents

1 Introduction **4**

2 System model **5**
2.1 Architecture model 5
2.2 The HPC-DAG task model 6
2.2.1 Specification tasks 6
2.2.2 Concrete tasks 7

3 Autoware: a complex real-time framework for autonomous driving **8**

4 Scheduling analysis **11**
4.1 Alternative patterns 11
4.2 Tagged Tasks 12
4.3 Deadlines and offsets assignment 13
4.4 Single engine analysis 15
4.5 Anticipating the activation of sub-tasks 16
4.6 Preemption-aware analysis 16

5 Allocation **18**
5.1 Allocation of task specifications 18
5.2 Sequential allocation 19
5.3 Parallel allocation 20

6 Related work **21**

7 Experimental results and discussions **23**
7.1 Task set generation 24
7.2 Simulation results and discussions 24
7.3 Preemption cost simulation 25
7.4 Real World application on the NVIDIA Jetson AGX Platform 26
7.4.1 Hardware and software platform 26
7.4.2 Experimental results 28

8 Conclusions and future work **30**

List of Figures

1 Task specification and concrete tasks 7
2 The NDT algorithm of the Autoware localization package modeled using HPC-DAG. 10
3 Tagged tasks for the concrete task of Figure 1b 12
4 Example of offset and local deadline 14
5 Maximal sequential subset example 17
6 Schedulability and utilization results 23
7 Preemption cost Lemma 3 against Theorem 2 26
8 Example of computer vision pipeline of NVIDIA VPI 28
9 Execution times of different tasks (for readability, the Y axis has a variable scale). 29
10 Deadline misses rate for real-task execution on Jetson AGX board . 30

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 MS3 is the set of static and dynamic tools implemented in the CLASS architecture ready to evaluate all the use cases. This milestone has been subject to constraints that have slowed progress behind expectations, have already lead to a 2-month delay, and have been reported separately to the Project Officer. To avoid further delay in delivery, we deliver all planned content under some mitigations that are expected to complete shortly after the milestone. In particular, the principal edge component, the smart cameras, have not yet been deployed in the MASA area, neither a working prototype has been under study. To comply with the request of scheduling and analyzing a sufficiently complex real world use case, we applied our technique to a subset of the well-known Autoware library for autonomous driving.

This Deliverable is an extension of D3.4 and treats more in depth the topic shown there.

The 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. In Section 3, we demonstrate the use of the HPC-DAG model by representing a module of the Autoware library.

Then, in Section 4, we present a schedulability analysis that will be used in Section 5 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 for schedulability analysis (Section 4.6).

After discussing related work in Section 6, our methodology is evaluated in Section 7 by comparing it with state-of-the-art models and algorithms.

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 (i) its execution capabilities, (e.g. its Instruction Set Architecture), specified by the engine's *tag*, and (ii) its scheduling policy. An engine's tag $\texttt{tag}(e_i)$ indicates the ability of a processor to execute 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 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. A deep learning task, such as the DNN for object detection discussed in deliverable D1.4, can be compiled to run on any engine, including CPUs and GPUs, however its worst-case execution time will probably be lower when running on accelerators such as a GPU and a DLA. Relevant benchmarks with respect to inference performance on GPU and DLA can be found in [26].

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 accelerators. We treat copy engines as processing units having *CP* tag, 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*. Our model allows different engines to support different scheduling policies. As shown in Section 4, our methodology may cope with different schedulability analyses for each independent engine. However, to simplify the presentation, in this deliverable we focus on *preemptive Earliest Deadline First (EDF)*. To help the reader to identify the tags easily, we use the pink color for CPU tag, the beige color for GPU tag, the blue color for DLA, the violet for CP tag and the green for the PVA tag.

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 $\texttt{tag}(v)$ which represents the engines where it is eligible to execute. A sub-task can only be allocated 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. Instead, we model a GPU sub-task as a single executing context able to potentially exploit all the parallel resources of the GPU's execution engine. This is compliant with the GPU abstraction that has been proposed in [10] 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). Non-determinism implies that 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. During the configuration phase (which is detailed in Section 5.1), 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. This can be useful when modeling sub-tasks than can be executed on different engines with different execution costs.

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 $\mathsf{pred}(n_j)$, 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 exists 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 define the set of *immediate successors* of node n_j, denoted by $\mathsf{succ}(n_j)$, 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

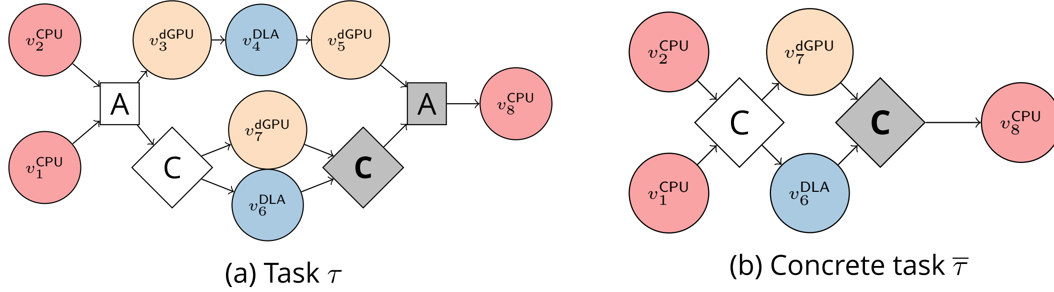
set of nodes for which there is a path from n_j. If a node has no successors, it is a *sink node* of the graph, and we allow a graph to have several sink nodes.

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. In the following, the *volume* vol$(\overline{\tau})$ denotes the total cumulative WCET of the concrete task. It is computed in linear time in the number of conditional vertices [5].

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.*



(a) Task τ

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

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 concurrently: during the off-line analysis they may be allocated to the same or to different engines. Sub-task v_4^{DLA} has an outgoing edge to v_5^{dGPU}, thus sub-task v_5^{dCPU} cannot start its execution before sub-task v_4^{DLA} has completed. Sub-tasks v_1^{CPU} and v_2^{CPU} have each one outgoing edge to the alternative node A. Thus, τ can continue the execution 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 subgraphs are alternative ways to execute the same functionalities at different costs. Figure 1b represents one of the concrete tasks of τ, where during the analysis, alternative execution $(v_3^{dGPU}, v_4^{DLA}, v_5^{dGPU})$ has been dropped.

Conditional nodes and alternative nodes always have at least two outgoing edges, so they cannot be sinks. For the sake of simplicity, we also assume that conditional nodes always have at least one predecessor node, so they cannot be sources.

In this study, we restrict ourselves to *well-nested graphs*: for every alternative (resp. conditional) node, there is always a corresponding *closing node* denoted by a black square (resp. black diamond) in Figure 1a, such that all paths starting from the alternative (resp. conditional) node contain the corresponding closing node. Please notice that relaxing this assumption would mean allowing edges from nodes that may potentially not be executed (e.g., because they belong to the false branch in a conditional statement) to nodes that are always executed, making it complex to track task dependencies at runtime.

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

We consider a *sporadic task* model, therefore parameter T represents the minimum inter-arrival time 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 $t +$ D. We assume *constrained deadline tasks*, that is D $\leq$ T.

We now present a procedure to generate a concrete task $\overline{\tau}$ from a specification task τ, by selecting one successor for all alternatives. 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, for each 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.

3 Autoware: a complex real-time framework for autonomous driving

In this section, we present how a complex real-time application can be modeled using the HPC-DAG task model.

Autoware[1] [20] is an open source autonomous driving framework that provides a set of software modules to perform sensing, computing and actuation for autonomous vehicles. Autoware can process different types of sensors data such as lidars, cameras, light detectors, etc., with the goal of creating a model of the surrounding environment that is then used to take actuation decisions. Autoware is composed of several packages ranging from *vehicle localization* to *path planning and following*. In this deliverable, we focus on the localization package.

The goal of this package is to compute a precise position of the autonomous vehicle in the environment. Therefore, it matches Lidar data with offline map data by means of the *Normal Distributions Transform (NDT) algorithm*. Autoware

[1] https://github.com/autowarefoundation/autoware

provides two implementations of NDT, a *pure* CPU implementation (all functions are implemented on the CPU) and a pure GPU-based implementation (all functions are implemented on the GPU).

We modeled the NDT algorithm with our HPC-DAG task model. Contextually, we have rewritten the NDT implementation to provide finer alternatives, by allowing the programmer to alternate different implementations at the function-call level.

Our task model allows the designer to express parallelization possibilities in two different ways: (i) using alternative nodes to express several off-line alternative implementations, each one having a different parallelism; or (ii) using conditional nodes for expressing alternative parallelization options to be chosen online based on runtime conditions. In the first case, the selection is done offline and does not change at run-time. In the second case, selection is performed online, e.g., depending on given data, sensors inputs, time to deadline, etc. When deriving a schedulability analysis, this latter case is harder, since the branch to execute is not known a priori, needing to consider the worst-case scenario among all the possible conditional branches. A similar approach can be found in [34–36].

To semplify the view of the complex graph, we report in Figure 2 only the first part of it, the remaining tasks from the NDT package will execute starting from node ndt_next. The part of the NDT presented here performs functions computePointGradients and computeHessian. Both can run on the CPU (with different variable configurations of threads) and on the GPU. The entry point is the node init, which reads data from different sources. Then, one of three alternative paths can be selected.If alternative A1 is selected, function computePointGradients is run in a fork/join fashion, using a scatter thread (CPGS1_C) and two parallel threads CPGP1_C and CPGP2_C before collecting the result on thread GPGG1_C.

The same processing can be achieved in another parallel configuration (the A2 alternative), by distributing the processing on 4 working threads. Finally, the compute point gradients can run on the GPU by following the alternative node A3. In this latter path, data must be copied from the CPU to the GPU by invoking CP1_M, then the actual processing can start by launching the GPU kernel CPG_G.

Just after the completion of the point gradient kernel, two alternative paths are possible, namely A4 and A5. If A4 is selected, condition HESO_2 is evaluated to check whether the function computeHessian must be performed. We highlight that such a condition is evaluated online, based on the result of the previous processing. Function computeHessian can be executed later on the GPU. Therefore, there is no need to copy buffers from host memory and CHG_1 GPU kernel can immediately start. If the hessian computation has been selected to run on any of the CPU fork join configurations delimited within the alt_3 block, and still assuming the precedent compute gradient run on the GPU, then the preliminary copy CP2_M is needed. Trivially, if the compute point gradients have been computed on any of the available CPU thread configurations and we elect to later compute the hessian on the GPU, then we will need the preliminary copy CP3_M, the GPU kernel itself CHG_2 and the copy back function CP5_M.

Typically there are more CPUs available than GPUs. Therefore, the GPUs are powerful but *scarce resources*, and the sub-task allocation must be care-

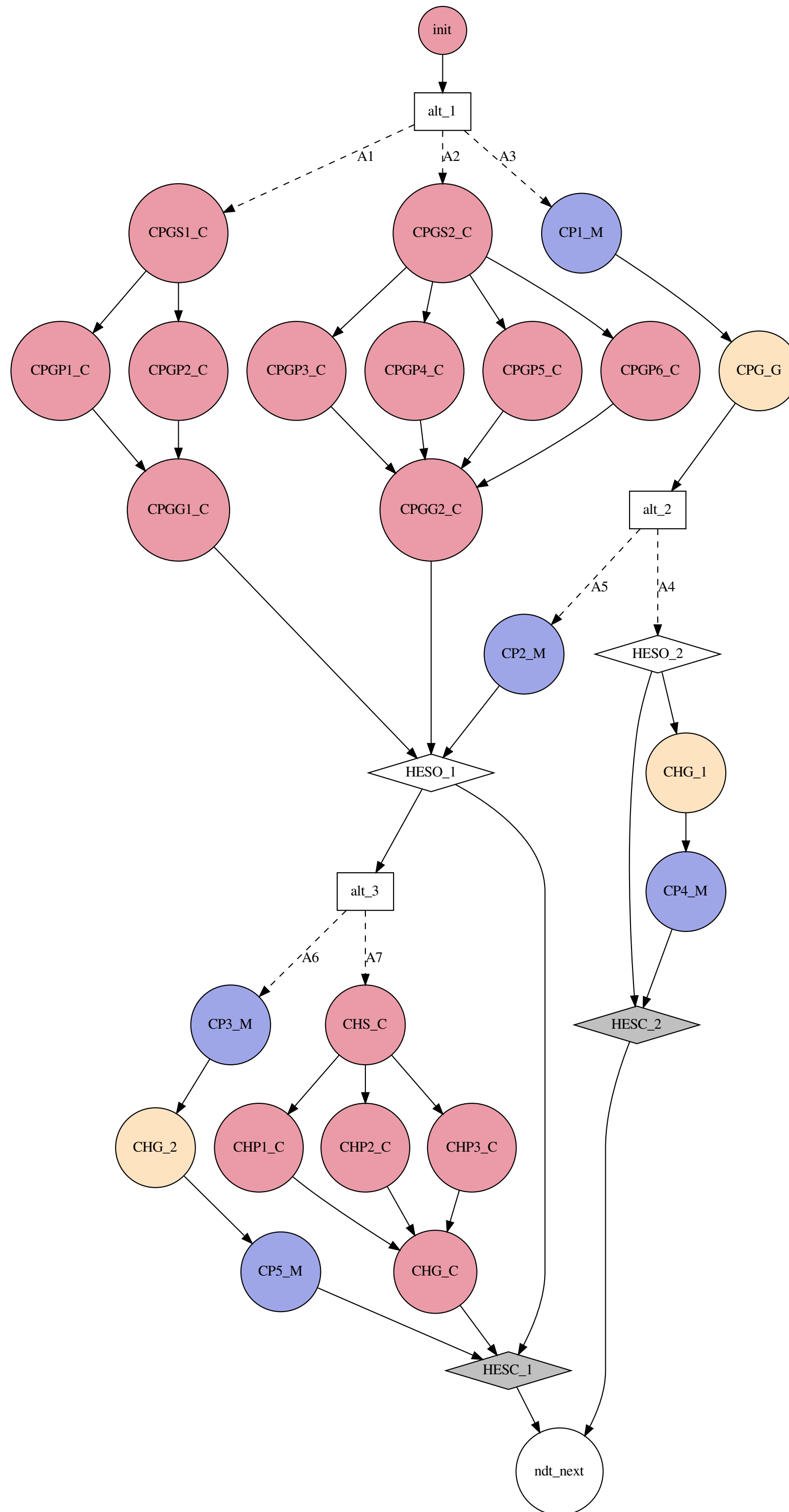


Figure 2: The NDT algorithm of the Autoware localization package modeled using HPC-DAG.

fully planned to avoid oversubscribing the GPUs and therefore decreasing the schedulability ratio.

With this example we showed that exploring the space of alternatives in such complex architectures and domain of applications is not a trivial aspect. The small example of Figure 2 contains 7 different concrete tasks. In the entire Autoware we counted more than 160 alternatives implementations of its functionalities, leading to an exponential number of combinations. Therefore, without a proper task model the system designer would simply be unable to rely on a sound schedulability analysis for deriving proper node-to-engine allocation policies. Our proposed HPC-DAG model is not only able to capture both the off-line complexity of defining multiple implementation of the same node, but it is also able to account for runtime task reconfigurations due to online conditionals. Such formal artefacts will be instrumental for the schedulability analysis we present in the following sections.

4 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 the well-known single processor schedulability tests, which make the analysis simpler and reduce the overhead due to thread migration compared to global scheduling.

The analysis presented in the Deliverable is modular, i.e., engines may have different scheduling policies, as described in Section 4.4. For the sake of simplicity, we will adopt preemptive-EDF (Earliest Deadline First) as a representative scheduler. EDF is known to be optimal for arbitrary collections of jobs on a single resource platform, and it has been recently implemented also for GPU platforms (see Section 6). The preemption cost estimation and analysis reported in Section 4.6 will consider both preemptive EDF and fixed priority scheduling (e.g. Deadline monotonic).

4.1 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 engines. Since the number of combinations can be very large, we propose a heuristic algorithm based on a *greedy* strategy (see Section 5). 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:*

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

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

4.2 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* $\overline{\tau}(\mathsf{tag}_i)$ *is a* tagged task *of* $\overline{\tau}$ *iff*

- $\overline{\tau}(\mathsf{tag}_i) = \{\mathsf{T}, \mathsf{D}, \mathcal{V}_i, \Gamma_i, \mathcal{E}_i\}$ *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* $\overline{\tau}(\mathsf{tag}_i)$ *in the isomorphism. If* $\mathsf{tag}(v) = \mathsf{tag}_i$*, then* $C(v') = C(v)$*, else* $C(v') = 0$*;*
- $\Gamma_i = \overline{\Gamma}$*.*

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

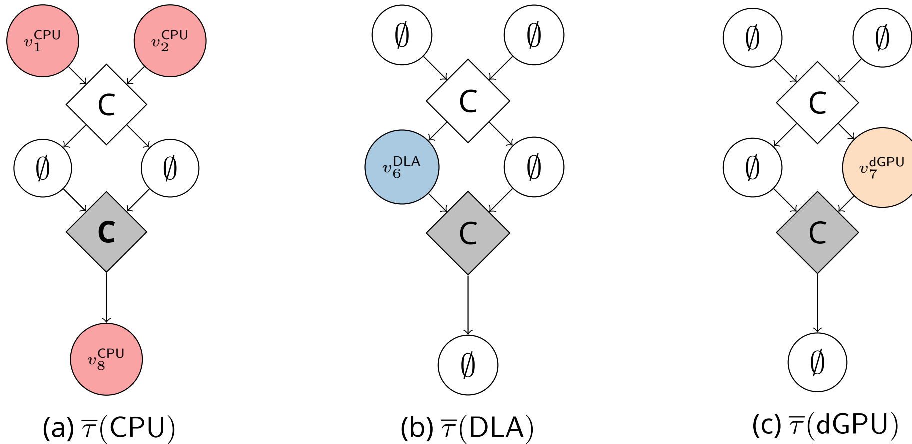


Figure 3: Tagged tasks for the concrete task of Figure 1b

Each concrete task generates as many *tagged tasks* as there are tags in the target architecture. Figure 3 shows the three tagged tasks for the concrete task in Figure 1b.

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

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

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

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

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 to that specific engine, the relation order prefers alternative patterns with lower workload for those engines.

4.3 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 automatically respected if every sub-task is activated after its offset and 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 [23] and [32].

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 of $\overline{\tau}$ 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 in the path:

$$\text{calculate_share}(v, P) = \frac{\text{Sl}(P, \text{D})}{|P|} \tag{2}$$

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

$$\text{calculate_share}(v, P) = \frac{\text{C}(v)}{\text{C}(P)} \cdot \text{Sl}(P, \text{D}) \tag{3}$$

Once the relative deadlines of the sub-tasks along the critical path have been assigned, we 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 [32].

Let $\text{O}(v)$ 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* $\text{O}(v) + \text{D}(v)$, 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 among the local deadlines of the predecessor sub-tasks.

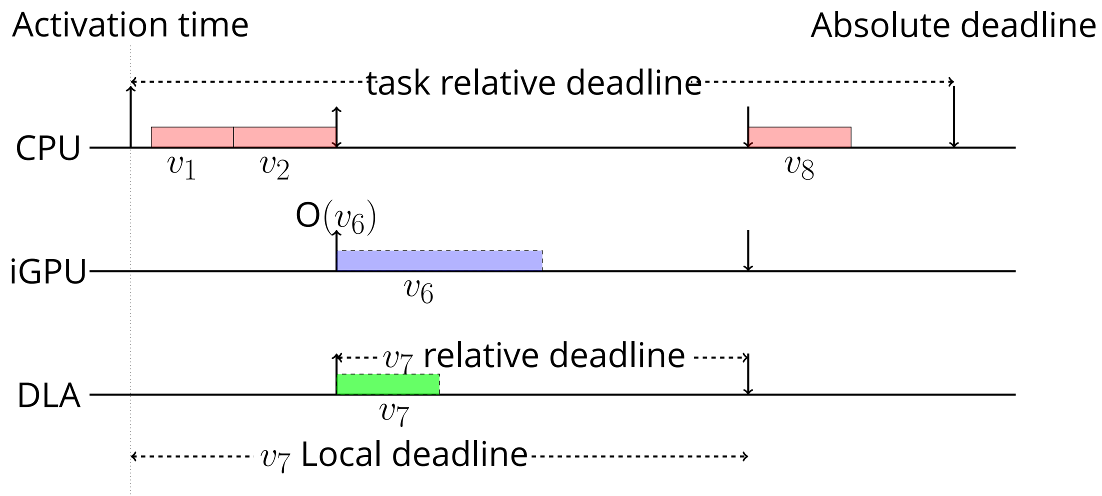


Figure 4: Example of offset and local deadline

Figure 4 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, 2) if sink sub-tasks are feasible, then the concrete task (tagged task, respectively) is feasible. □

4.4 Single engine analysis

In this section, we assume that sub-tasks have 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 $\mathcal{T}$ be 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 [6] 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. Similar analysis techniques can be found in [3].

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

4.5 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 [7]. 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.

4.6 Preemption-aware analysis

The cost of preemption may significantly vary depending on the preempted task and on the engine upon which the task is running. In recent GPUs for instance, preempting an executing task can be a costly operation (see Section 7.3). Due to the differences in preemption granularity in GPUs, preempting a graphics kernel induces a larger cost compared to preempting a compute-only GPU kernel (e.g., a CUDA kernel). Therefore, we need to account for the cost of preemption in the analysis. From now on, a sub-task will be also characterized by $\mathsf{pc}(v)$, the timing cost of preempting the sub-task v.

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. In the following, we consider the EDF and fixed priority (FP) scheduling policies. In the latter case, $P(v)$ denotes the fixed priority assigned to vertex v.

Lemma 3. *Let* $\mathcal{V} = \{v_1, v_2, \cdots, v_K\}$ *be a set of sub-tasks to be scheduled by EDF (resp. FP) 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* $\mathsf{C}(v'_i) = \mathsf{C}(v_i) + \mathsf{pc}^i$ *and* $\mathsf{pc}^i = \max\{\mathsf{pc}(v) | v \in \mathcal{V} \wedge D(v) > D(v_i)$*(resp.* $P(v) > P(v_i)$ *for FP*$)\}$*.*

If $\mathcal{V}^{\mathsf{pc}}$ *is schedulable by EDF (resp. FP) 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).

We will now further reduce the preemption cost estimation by considering only EDF and Fixed Priority with Deadline Monotonic priority assignment (DM). Let us start by observing that, in the case of EDF and DM 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, i.e., $D(v_i) < D(v_j)$.

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

We denote by $\text{cand}(\mathcal{V}^M)$ 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 $\text{cand}(\mathcal{V}^M)$.

Example 2. *Consider task τ in Figure 5. Assume all subtasks have been allocated on a given engine, except for subtask v_4.*

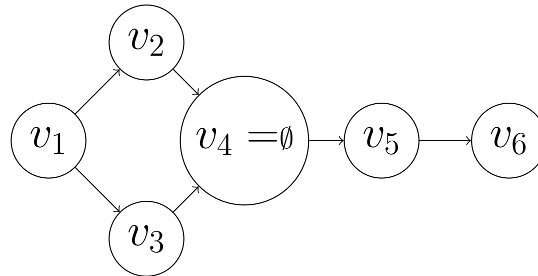


Figure 5: Maximal sequential subset example

According to this allocation, the task has two maximal sequential subsets: the first is composed of v_1, v_2 and v_3 whereas the second is composed of v_5, v_6

We observe that, since all the sub-tasks in $\mathcal{V}^M$ are allocated onto 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 EDF/DM 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 Equations* (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/DM when considering a null preemption cost, then $\mathcal{V}$ is schedulable when considering the cost of preemption.

Proof. For space constraints, 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 tasks 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-tasks 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 from v^M are not subject to preemptions. □

If EDF/DM are selected as engine scheduling policies, the preemption cost is computed using Theorem 2. For other scheduling policies the estimation of Lemma 3 can be used.

5 Allocation

5.1 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 are completed before their deadlines. From an operational point of view, this is equivalent to finding a feasible solution to a complex Integer Linear Programming (ILP) problem. However, 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 time. In [36], authors tried to formulate a similar relaxed allocation problem using ILPs. They proved that finding an optimal solution can take several hours for very small

size problems. Therefore, in this section we propose a set of greedy heuristics to quickly explore the space of solutions.

Algorithm 1 describes the basic approach. The algorithm can be customised with four parameters: order is the sorting order of the concrete task sets (see Sections 4.1 and 4.2); slack concerns the way the slack is distributed when assigning intermediate deadlines and offsets (see Section 4.3); alloc can be best-fit (BF) or worst-fit (WF); omit concerns the strategy to eliminate sub-tasks when possible (see Section 5.3).

At each step, the algorithm tries to allocate one single task specification (the for loop at line 5). For each task, it first generates all concrete tasks (line 6), and sorts them according to one relationship order ($\succ$ or $\gg$). For each concrete task, it first assigns the intermediate deadlines and offsets according to the methodology described in Section 4.3 (line 8), using one between the fair or the proportional slack distributions. It then separates the concrete tasks into tagged tasks according to the corresponding tags (line 9).

The algorithm tries to allocate every tagged task onto single engines having the corresponding tag (line 13) (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 (line 14). 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 4.6. In particular, by allocating an entire tagged task onto a single engine, we reduce the number of null sub-task to the minimum 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 done by procedure parallelize, which will be described in Section 5.3. 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 this process is also unable to find a feasible concrete task, the analysis fails (line 29).

5.2 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 parameter alloc (Best-Fit or Worst-Fit utilization order). 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.

Algorithm 1 Allocation algorithm

```
1: input : 𝒯: set of task specifications
2: parameters : order (≻ or ≫), slack (fair or proportional),
3:              alloc (BF or WF), omit (parallel or random)
4: output : SUCCESS or FAIL
5: for τ ∈ 𝒯 do
6:     Ω = generate_and_sort_concrete_task(τ, order)
7:     for (τ̄ ∈ Ω) do
8:         assign_deadlines_offsets(τ̄, slack)
9:         𝒮(τ̄) = filter_tagged_task(τ̄)
10:    end for
11:    allocated = false
12:    for (τ̄ ∈ Ω) do
13:        if (feasible_sequential(𝒮(τ̄), alloc)) then
14:            assign task to engines; allocated = true; break;
15:        end if
16:    end for
17:    if (not allocated) then
18:        for (τ̄ ∈ Ω) do
19:            (τ̄′, τ̄″) = parallelize(τ̄, alloc, omit)
20:            if (τ̄′ ≠ ∅) then
21:                allocate τ̄′ to selected engines
22:                add back τ̄″ to 𝒯
23:                allocated = true; break
24:            end if
25:        end for
26:        if (not allocated) then return FAIL
27:    end if
28: end for
29: return SUCCESS
```

5.3 Parallel allocation

When the sequential allocation fails for a given task specification, Algorithm 1 tries to allocate one or more of its tagged tasks onto multiple engines having the same tag by invoking *parallelize* (Algorithm 3). The latter 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 , *parallelize* algorithm (Line 5) 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 the list of non allocated sub-tasks $\bar{\tau}''$ (line 11). We propose two heuristics:

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

Algorithm 2 feasible_sequential

```
1:  input: S(τ̄): set of tagged tasks, alloc
2:  output: feasibility: SUCCESS or FAIL
3:  for (τ̄(tag) ∈ S(τ̄)) do
4:      engine_list=select_engine(tag)
5:      sort_engines(engine_list, alloc)
6:      f = false
7:      nfeas = 0
8:      for (e ∈ engine_list) do
9:          f = dbf_test(τ̄ ∪ T_e)                 ▷ T_e=subtasks of engine e
10:         if (f) then
11:             save_allocation(τ̄, e)
12:             nfeas ++
13:             break
14:         end if
15:     end for
16:     if (not f) then return FAIL;
17: end for
18: if (nfeas = |S(τ̄)|) then
19:     return SUCCESS, saved_allocations
20: end if
```

2. **Parallel** heuristic: to be feasible, the critical path of each tagged task must be feasible on an unlimited number of engines. Thus, we start by removing sub-tasks that do not belong to the critical path as they may be the ones causing the non-feasibility (here we assume that sequential allocation has failed). Once a vertex is omitted, the next omitted vertexes will be its predecessors or successors, so to maximize the size of maximal sequential subsets.

The feasibility test is repeated (Line 10, Algorithm 3) 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, Algorithm 3). 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.

6 Related work

Many tasking models have been proposed in the real-time literature to properly capture timing requirements and concurrent execution flows. The multiframe model has been proposed in [25] representing each task as a sequence of sub-tasks (called frames), each with its own deadline. In [4], such a model has been generalized using directed acyclic graphs to express conditional dependencies within a task. A further extension has been presented in [30] with the digraph model, using state machines to represent cycles within a task. While all above

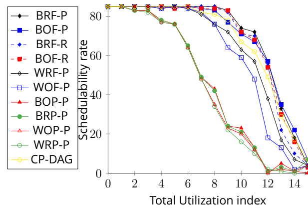
Algorithm 3 parallelize

```
1:  input: τ̄: concrete task, alloc (BF or WF),
2:         omit (parallel or random)
3:  output: concrete tasks (τ̄′, τ̄″)
4:  τ̄′ = ∅, τ̄″ = ∅
5:  for (τ̄(tag) ∈ S(τ̄)) do
6:      engine_list=select_engines(tag)
7:      sort(engine_list, alloc)
8:      for (e ∈ engine_list) do
9:          f=dbf_test(τ̄(tag) ∪ T_e)                ▷ T_e=subtasks of engine e
10:         while (not f) do
11:             τ̄″ = τ̄″ ∪ remove(τ̄(tag), omit)
12:             f=dbf_test(τ̄(tag) ∪ T_E)
13:         end while
14:         if (τ̄(tag) ≠ ∅ ) then
15:             τ̄′ = τ̄′ ∪ save_allocation(τ̄(tag), e)
16:             τ̄(tag) = τ̄″, τ̄″ = ∅
17:             allocated = true
18:             break
19:         end if
20:     end for
21:     if (not allocated) return ∅, τ̄
22: end for
23: return τ̄′, τ̄″
```

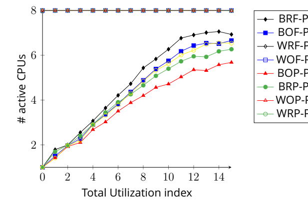
models are restricted to sequential tasks, DAG-based models have been proposed to allow expressing parallelism within a task. A response time analysis for partitioned fixed-priority DAGs is presented in [16]. A similar model is adopted in [22], proposing speedup and capacity augmentation bounds for global EDF, RM and a federated scheduling approach. In [24], conditional nodes have been introduced to parallel DAG tasks for modeling conditional branches, providing a response-time analysis under global scheduling.

Many other works proposed real-time task models based on DAGs [16, 22–24, 27–29, 35, 36]. However, to the best of our knowledge, none of them allows modeling alternative implementations of the same functionality on heterogeneous computing engines. The closest model to the one we propose in this work can be found in [34], where alternative implementations are modeled within parallel digraphs. We extend the model in [34] by allowing both alternative and conditional execution blocks. The Syndex project [15], [21] proposes a methodology, called AAA, to execute real-time tasks onto heterogeneous architectures. However, it does not address alternative implementations as those captured by our novel HPC-DAG model.

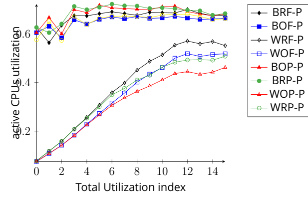
In [23], the deadline assignment problem is addressed for distributed real-time systems. Two algorithms are proposed: Fair Laxity Distribution (FLD) and Unfair Laxity Distribution (ULD). In [33], a general framework is presented for partitioning real-time tasks onto multiple cores using resource reservation. A



(a) Schedulability rate VS util.



(b) #Active CPUS vs util.



(c) Active CPU util. VS tot. util.

Figure 6: Schedulability and utilization results

technique is proposed to set task activation times and deadlines, using an ILP formulation to solve the allocation and assignment problems. However, these approaches may be very time consuming when applied to applications consisting of tens or hundreds of sub-tasks.

Regarding scheduling algorithms to be adopted within the heterogeneous engines of modern embedded platforms, a key factor to consider is the cost of preemptions, which may be non negligible, especially for GPU engines. Initial work on preemptive GPU scheduling assumed that preemption was viable at the kernel granularity [38]. Preemption for GPU CUDA Kernels in NVIDIA architectures are nowadays natively supported, both via hardware and software mechanisms [8–10, 12, 17, 31]. More recently, other device vendors and programming models started to natively support preemption at software (e.g. Khronos' OpenCL [13]) and architectural level, e.g., AMD devices [1] and Intel iGPUs [19]. Depending on the computing architecture and on the nature of the workload, GPU tasks present different degrees of preemption granularity and related preemption costs.

7 Experimental results and discussions

In this section, we evaluate the performance of our scheduling analysis and allocation strategies.

In the first set of experiments (Section 7.2), we evaluate the algorithms on a large number of synthetically generated tasksets, using well-known tools in the real-time systems literature, as detailed in the following section. We compare our novel approach against the CP-DAG model proposed by Melani et al. in [24], in which a global scheduling approach is adopted; in contrast, our analysis is based on partitioned scheduling. For the sake of fairness related to this comparison, we extend the CP-DAG model to support multiple engines. We applied the same allocation heuristics of Section 5 and the same scheduling analysis of Section 4 to both our proposed HPC-DAGs and the CP-DAG. For this first set of experiments,

we consider an hardware platform consisting of 1 dGPU, 1 iGPU + 8 CPUs + 1 PVA + 1 DLA. This corresponds to using half the resources of a NVIDIA Pegasus board. In fact, in the engineering practice it is common to reserve half of the engines for real-time applications and the rest for best-effort applications.

To further demonstrate the effectiveness of our approach, in Section 7.4 we implemented, executed and analysed realistic computer vision applications using our novel scheduling and allocation mechanisms. For this second set of experiments, we consider a NVIDIA Jetson AGX platform consisting of 8 CPU + 1 iGPU + 1 DLA + 1 PVA.

7.1 Task set generation

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* [14] 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 either an alternative node or a conditional node with a probability of 0.7.

To avoid untractable hyper-periods, the period of every task is generated randomly according to values taken from a range in $[120, 120000]$. For every sub-task, we randomly select a tag. Further, for each tag, we use the UUniFAST-Discard algorithm again to generate individual sub-task utilizations. Thus, the sub-tasks' utilization can never exceed 1. Further, we multiply the utilization of each sub-task by the task period to generate the vertex execution time. A CP-DAG is generated from a HPC-DAG by selecting one of the possible concrete tasks at random.

7.2 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 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 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 for the random heuristic which randomly selects the sub-task to remove. Average values for 85 simulations per utilization step are presented.

Figure 6a 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-based heuristics: this can be explained by observing that BF tries to pack the largest possible number of sub-tasks into the minimum number of engines, hence providing for more flexibility to schedule *heavy* tasks on other engines.

In the figures, the CP-DAG model proposed in [24] 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 6a 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 are able to achieve high schedulability rates. BRF outperforms the others because it aims at relaxing the utilization of scarce engines, thus avoiding the unfeasibility of certain task sets due to high load on the scarce engines (i.e. DLA, PVA and GPUs). However, when dealing with highly loaded task sets, BOF presents better schedulability rates, as it reduces the execution overheads on all engines. Such overhead reduction occurs because BOF selects the first schedulable concrete task having the lowest volume.

Figure 6b reports the average number of active cores (CPUs) as a function of the total utilization. WF-based heuristics always use all the available cores, as we generate at least 15 CPU subtasks, i.e., the number of tasks is larger than the number of CPUs. BF heuristics aim at packing the maximum number of sub-tasks within the minimum number of engines, thus the utilization increases quasi-linearly. This occurs until the maximum schedulability limit is reached. BRF heuristic uses more CPU cores because it *preserves* the scarce resources, thus it uses more CPU engines. As BOF privileges the overall load reduction, it also manages to reduce the load on the CPUs compared to BRF.

Figure 6c shows the average active utilization for CPUs. As expected, average utilization of BF-based heuristics is higher compared to WF. 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.

7.3 Preemption cost simulation

In all previous experiments, we applied the analysis described in Section 4.6 to account for preemption costs. In particular, we applied the technique of

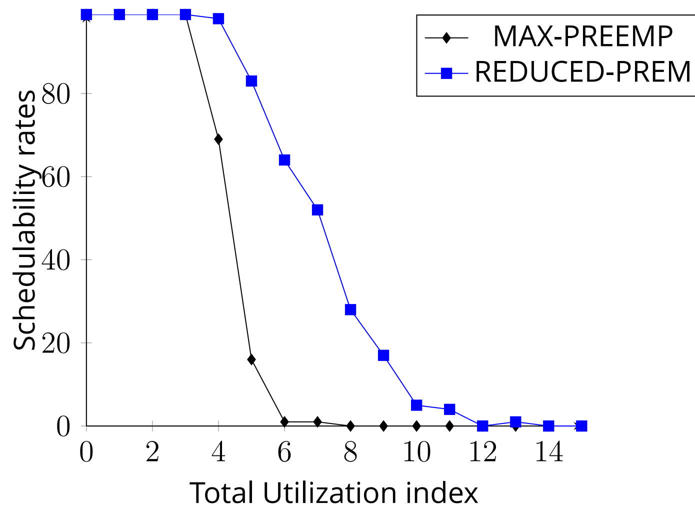


Figure 7: Preemption cost Lemma 3 against Theorem 2

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. These are representative numbers experimented in real settings. Even if DLA and PVA are non-preemptable engines, longer jobs might be split into smaller chunks. More specifically, on a DLA, a neural network inference kernel can be split on a per-layer basis. Similarly, on the PVA, a vision processing algorithms might be applied to smaller images obtained as portions of the original full-size input images. In such cases, we factor in the preemption overhead the splitting cost due to the submission of multiple kernel calls instead of a single batch of commands. To highlight the importance of a proper analysis for preemption costs, we report in Figure 7 the schedulability rates obtained by BRF-P in two 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. As the utilization increases, the schedulability drastically falls for the first method, while the analysis of Theorem 2 provides for tighter schedulability rates.

7.4 Real World application on the NVIDIA Jetson AGX Platform

In this section, we present results of the actual execution of a set of computer vision tasks on a NVIDIA Jetson AGX board.

7.4.1 Hardware and software platform

The first set of experiments considers the NVIDIA Pegasus, an automotive-grade platform featuring an 8-core CPU and four different kinds of accelerators: integrated GPUs, discrete GPUs, a cluster of DLA accelerators and a cluster of PVA accelerators. This second set of experiments considers the less expensive

Jetson AGX Xavier platform[3], featuring a similar set of engines, excluding the discrete GPU. Each processing engine is treated as a single computing resource, i.e., allowing the execution of only one sub-task at a time.

We have profiled different implementations of different processing algorithms from the NVIDIA Vision Programming Interface (VPI)[4]. The NVIDIA VPI is a software library that provides Computer Vision/Image Processing algorithms implemented on the different engines available on the Jetson AGX board: 8-Core ARM-v8.2 compliant CPU, the Programmable Vision Accelerator (PVA) and the CUDA enabled integrated GPU. Memory allocation operations are managed before executing the real-time processing. We provide a brief description of the different tasks available on the NVIDIA VPI that we use in our tests, denoting as X the input buffers (i.e. L or R, as in left and right image) and as Y the engine tag where a sub-task is meant to execute (i.e. C for CPU, G for GPU and P for PVA in Figure 8).

- Disparity Estimator (DISX) uses a pair of images from a stereo camera to infer the depth of a scene. The NVIDIA VPI library provides implementation for the CPU, the GPU and the PVA.
- Box Image Filter (BFXY) is a low-pass filter that smoothens the input image by averaging surrounding pixels, removing details, noise and edges. The NVIDIA VPI library provides implementation for the CPU, the GPU and the PVA.
- Bilateral Image Filter (BLY) is a non-linear, edge-preserving smoothing filter that is commonly used in Computer Vision as a simple noise-reduction stage in a pipeline. It can run on CPU or in the GPU.
- Image Resampler (DSXY) is used to re-scale the input image, resampling its content to make it compliant to a given output size. It can run on CPU or in the GPU.
- Harris Keypoint Detector (HKY) implements a detection operator that is commonly used to detect keypoints and infer features of an image. This detector can run on CPU or in the GPU.

Figure 8 describes an example of application that uses the above tasks. The example is provided by NVIDIA VPI documentation as a reference to a typical vision processing pipeline applied for scene analysis obtained through cameras. The task in Figure 8 has two outputs, the Harris keypoints and the disparity estimation: it processes two images in parallel (left and right) to compute disparity, and only the right image to compute the Harris keypoints. The task starts by reading input images via sub-task INIT. To compute disparity, images are pre-processed using Box Filters applied to both the right and the left images (BFXY). We highlight that this pre-processing can rely on three different implementations within the NVIDIA VPI library (CPU, GPU and PVA). In parallel, the Bilateral filter is

[3] https://elinux.org/Jetson_AGX_Xavier
[4] https://docs.nvidia.com/vpi/index.html

applied before starting the Harris Key Point detection. As the stereo disparity requires the two input images to be of the same size, input images are then rescaled to the desired resolution by invoking the DSXY processing node before invoking the DISX filter. In a similar way, HKY is released once the BLY terminates.

The task graph is very complex, as it generates 432 alternative implementations (concrete tasks), making it impossible to adopt brute-force analysis of all alternatives.

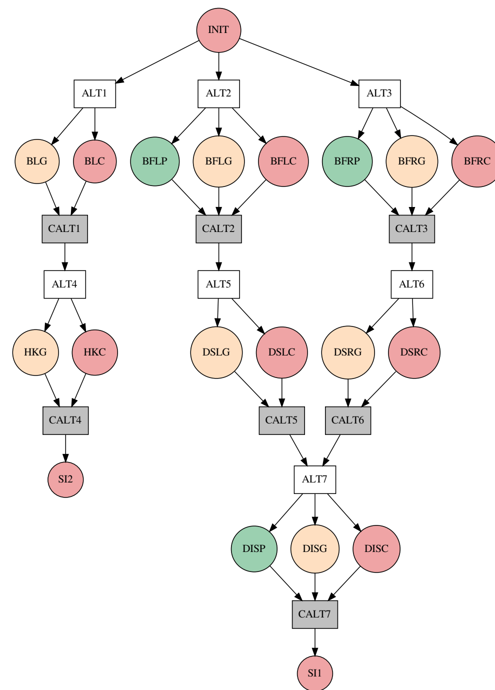


Figure 8: Example of computer vision pipeline of NVIDIA VPI

In Figure 9, we report average execution times as well as confidence intervals as measured from 200 executions of every sub-tasks in the pipeline of Figure 8 on the engines available on the selected platform. We observe that GPU and PVA implementations are usually much faster than CPU counterparts. For example, image disparity runs 50 times faster on the PVA than on the CPU. However, only few functions, such as box filters and disparity estimators, are available on the PVA, leaving the GPU the only alternative to the CPU for more general parallelizable tasks. We would like to point out that the NVIDIA VPI library handles memory copies and translations transparently to the programmer. Thus, memory transfers and address translation costs are included in the execution times reported in Figure 9, instead of considering them as separate nodes within the task-graph.

7.4.2 Experimental results

We used the functions described in the previous section to generate a set of randomised experiments. For each experiment, we consider 5 task graphs. Each

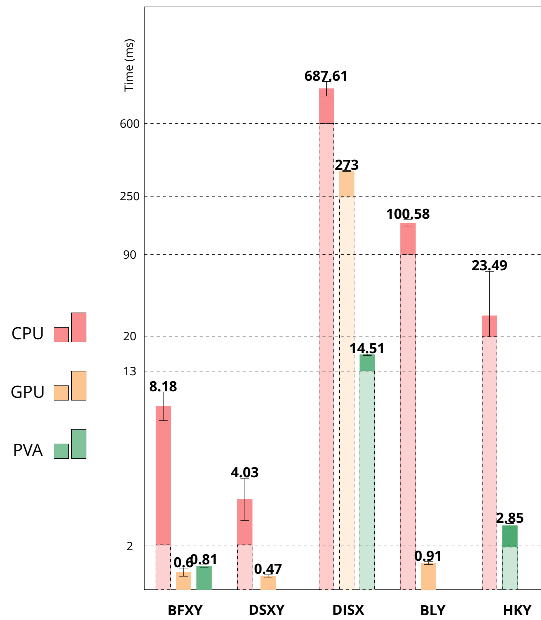


Figure 9: Execution times of different tasks (for readability, the Y axis has a variable scale).

task graph is obtained from the task graph depicted in Figure 8 by adding/removing a random vertex (and the corresponding edges). For each task graph, we consider analysis subtasks WCET from the measures of Figure 9, by adding average execution time to the confidence interval values. Then we generate a random utilisation, and a period using the same techniques described in Section 7.1. We assume that the task deadlines are equal to their assigned periods. Task graphs are allocated on the considered platform (8 CPUs + 1 iGPU + 1 PVA + 1 DLA) using the algorithms of Section 5, however we allow all heuristics to continue allocation even when schedulability fails (i.e. line 14 of Algorithm 1 is executed unconditionally).

In order to compare against a baseline approach, we have additionally implemented a naive heuristic, which always selects the concrete task allocations that feature the shortest execution times, therefore privileging the PVA and GPU implementations when available (i.e. line 6 of Algorithm 1 only generates one concrete task). We adopt non-preemptive FP scheduling for both the GPU and the PVA, using the PRUDA library [37], and partitioned preemptive FP scheduling for the CPUs.

Finally, the code for the task graphs is automatically generated, compiled and executed on the target platform. Each sub-task is implemented by a separate thread, and thread synchronization is implemented using semaphores. Deadline misses are detected and counted during execution. The results are reported in Figure 10.

Assigning long periods and deadlines to the nodes causes the system as a whole to easily sustain the required computational load for all the relevant engines. In these cases (e.g. utilization below 2 in Figure 10), even the naive scheduling and allocation policy has no deadline miss. When the load increases, i.e. deadlines and period are assigned smaller values, the accelerators (PVA and

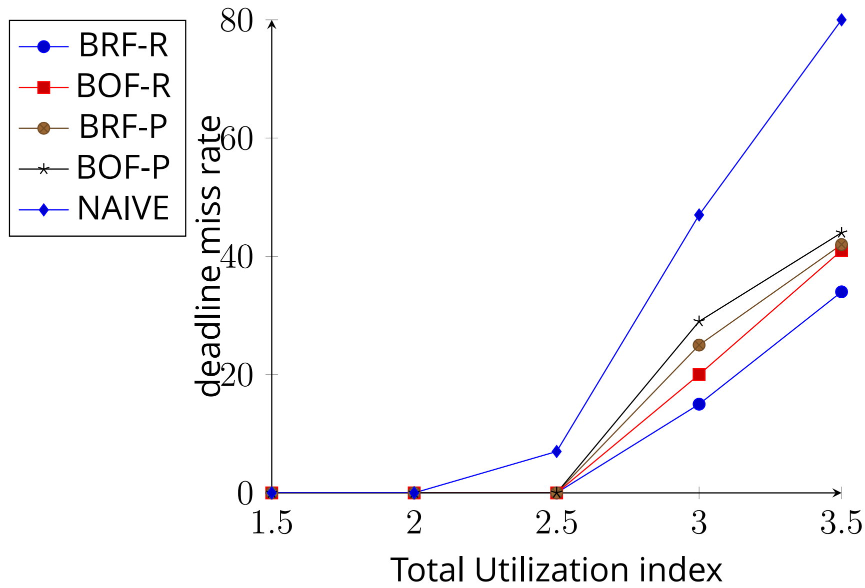


Figure 10: Deadline misses rate for real-task execution on Jetson AGX board

GPU) cannot sustain the assigned load, leading to deadline misses when the naive approach is adopted. Instead, our proposed heuristics are able to alleviate the load on the accelerators by assigning tasks to the CPU core as well, hence reducing the resulting deadline misses.

It is not trivial to select which task to run on the CPUs, as the difference between the accelerators execution time and the CPU execution time might be dramatic. Our heuristics are able to detect when such differences would still enable a feasible taskset. More specifically, during these experiments we observed that the CPU is selected for the box filter and the downscaling, which have a limited performance deterioration when executing on the CPUs, compared to the Harris and disparity task.

From the discussion in Section 7.2, the fair deadline assignment heuristics result in better schedulability rates than the proportional deadline assignment heuristics, and this is confirmed in this set of experiments. When the sequential execution fails, the parallelize heuristics select the subtasks to parallelize according to (i) random and (ii) parallel approaches. The parallel approach has less deadline misses because it enforces subtasks in the same path to be allocated together, thus reducing the need for expensive copy operations.

8 Conclusions and future work

In this Deliverable, we presented the HPC-DAG real-time task model, which allows specifying both off-line and on-line alternatives of a task instance, to fully exploit the heterogeneity of complex embedded platforms. We also presented a schedulability analysis and a set of heuristics to allocate HPC-DAGs on heterogeneous computing platforms.

Our proposed model and related analysis are able to capture the complex-

ity of the underlying hardware architecture and can be exploited to provide a formal and tightly-bound schedulability assessment even when the available accelerators have scheduling limitations, e.g., when they differ in the provided preemption support. In case of preemptive scheduling, our analysis takes into account the often non-negligible cost of preemption, by providing a significantly less pessimistic analysis. We provided an exhaustive evaluation of different heuristics to tackle the complex allocation problem on heterogenous embedded platforms, profiling the schedulability ratio for workloads of different sizes. The heuristics derived from our novel model are able to outperform naive approaches without being computationally intractable as brute-force approaches. While this work only considered NVIDIA-based platforms (due to our choice of platform in the project), our model and related analysis and allocation mechanisms can be easily ported to similar heterogenous platforms.
As future work, we are considering extending our framework to account for memory interference among different compute engines. This is a crucial aspect when measuring the worst-case execution time of latency-sensitive applications executing on heterogeneous engines sharing a common memory interface. Ignoring the effect of memory contention might significantly impact performance and time predictability [2, 11].

References

[1] Anirudh R Acharya, Swapnil Sakharshete, Michael Mantor, Mangesh P Nijasure, Todd Martin, and Vineet Goel. Primitive level preemption using discrete non-real-time and real time pipelines, February 19 2019. US Patent App. 15/828,055.

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

[3] S. Baruah. The federated scheduling of systems of conditional sporadic dag tasks. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2015.

[4] Sanjoy Baruah. The non-cyclic recurring real-time task model. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 173–182. IEEE, 2010.

[5] 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.

[6] 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.

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

[8] Jon Calhoun and Hai Jiang. Preemption of a cuda kernel function. In *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 247–252. IEEE, 2012.

[9] 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.

[10] 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.

[11] 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.

[12] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *ACM SIGPLAN Notices*, volume 52, pages 3–16. ACM, 2017.

[13] Ming-Tsung Chiu and Yi-Ping You. Enabling opencl preemptive multitasking using software checkpointing. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, page 15. ACM, 2018.

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

[15] O. Feki, T. Grandpierre, M. Akil, N. Masmoudi, and Y. Sorel. Syndex-mix: A hardware/software partitioning cad tool. In *2014 15th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, pages 247–252, 2014.

[16] 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.

[17] Christoph Hartmann and Ulrich Margull. Gpuart-an application-based limited preemptive gpu real-time scheduler for embedded systems. *Journal of Systems Architecture*, 97:304–319, 2019.

[18] Sihuizi Jin, Zhenning Wang, Quan Chen, and Minyi Guo. Preemption-aware kernel scheduling for gpus. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 525–532. IEEE, 2017.

[19] Stephen Junkins. The compute architecture of intel processor graphics gen9. *paper, Aug*, 14:22, 2015.

[20] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[21] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *Proceedings of European Control Conference, ECC'91*, Grenoble, France, July 1991.

[22] 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.

[23] 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.

[24] 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.

[25] Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. *IEEE transactions on Software Engineering*, 23(10):635–645, 1997.

[26] Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 23, 2019.

[27] 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.

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

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

[30] 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.

[31] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. *ACM SIGOPS Operating Systems Review*, 51(2):483–496, 2017.

[32] 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.

[33] 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.

[34] 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*, 68(10):1511–1524, Oct 2019.

[35] 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.

[36] 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.

[37] Houssam-Eddine Zahaf and Giuseppe Lipari. Design and analysis of programming platform for accelerated gpu-like architectures. In *Accepted in Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '20. ACM, 2020.

[38] 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, 2013.