Preemption-Aware Allocation and Deadline Assignment for Conditional DAGs on Partitioned EDF

Houssam Eddine Zahaf, Giuseppe Lipari, Smail Niar

► **To cite this version:**

Houssam Eddine Zahaf, Giuseppe Lipari, Smail Niar. Preemption-Aware Allocation and Deadline Assignment for Conditional DAGs on Partitioned EDF. The 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Aug 2020, Seoul, South Korea. 10.4230/LIPIcs.Pre . hal-02077110

HAL Id: hal-02077110
https://hal.science/hal-02077110

Submitted on 26 Nov 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Preemption-Aware Allocation, Deadline Assignment for Conditional DAGs on Partitioned EDF

Houssam-Eddine ZAHAF
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

Giuseppe Lipari
Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

Smail Niar
LAMIH, Université polytechnique Hauts de France, Valenciennes, France

Abstract

Complex heterogeneous hardware platform are increasingly used for implementing critical real-time application like ADAS and autonomous driving. To better support real-time workloads, GPUs have evolved to allow preemption for computationally intensive tasks and for graphical tasks. However for certain tasks the cost of preemption can be very high, and must be accounted in the design and in the scheduling analysis.

In this paper, we address the problem of allocating a set of real-time tasks, modeled by conditional directed acyclic graphs, onto multiprocessor platforms under partitioned preemptive Earliest Deadline First scheduling, assuming a non-negligible cost of preemption. We propose methods for assigning intermediate deadlines and offsets to real-time C-DAGs, so to remove unnecessary preemptions and reduce the total preemption overhead.

The effectiveness of the proposed technique is evaluated using a large set of synthetic tasks sets.

2012 ACM Subject Classification General and reference → General literature; General and reference

Keywords and phrases Real-time, partitioned, preemption, clustering

Digital Object Identifier 10.4230/LIPIcs.Pre Print.2019.X

Acknowledgements This work is supported in part by IRCICA, USR 3380, F-59650 Villeneuve d'Ascq, France, and by the ELSAT project.

1 Introduction

Recent embedded platforms combine several computing cores with different instruction-set architectures and computation capacities. An example of such architecture is the NVIDIA Xavier-based board, which comprises different accelerators, like GPUs, DLAs, etc., together with a set of classical ARM cores onto the same System On Chip (SoC). These platforms are the preferred choice for modern time-critical applications, like Advanced Driving Assistance Systems (ADAS), which need an ever-increasing amount of computational power for executing complex time-critical tasks.

These applications are typically structured as a set of concurrent tasks, each one modeled by a *Directed Acyclic Graph* (DAG) of sub-tasks. Moreover, they may exhibit dynamic behavior. For example, when an ADAS detects an obstacle, it may run effective algorithms to precisely detect and avoid the obstacle, otherwise it may continue running less-precise but less time-consuming sub-tasks. The Conditional DAG (C-DAG) [?] has been proposed to efficiently model and analyze such dynamic behavior.

© Author: Please provide a copyright holder;
licensed under Creative Commons License CC-BY
Pre print.
Editor: Sophie Quinton; Article No. X; pp. X:1–X:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Several difficult challenges are encountered when scheduling real-time applications modeled by C-DAGs on such architectures: the choice of the scheduling algorithm, and how to allocate (sub-)tasks to computational resources. Global scheduling may not always be available on heterogeneous architectures, and task migration may produce a high overhead. Therefore, in this paper we focus on *partitioned scheduling*. Preemptive EDF is known to be an optimal algorithm for single processors, therefore our strategy consists in allocating sub-tasks to computational resources, and then use preemptive EDF on each resource.

In most of the literature on real-time scheduling, the cost of preempting a task is considered to be negligible when doing the schedulability analysis. Indeed, this is the case in classical general purpose processors. However, preemption can be a very costly operation on some computing unit, e.g. GPUs. In some extreme cases, the cost of saving and restoring the context may exceed the worst-case execution time of the executing task (see Section 2.1 for a description of how preemption works on modern GPUs). Clearly, in such extreme cases, it is more effective to use a non-preemptive scheduling algorithm. However, in other cases, preemption is necessary to ensure schedulability; the ideal situation would be to perform a preemption only if strictly necessary.

Many techniques for limiting preemption have been proposed in the literature (see Section 7 for an overview of related work). In this paper we will use a novel technique that takes advantage of a property of the EDF scheduler. We will now briefly describe our main idea.

First of all, to correctly schedule the sub-tasks of a C-DAG, we need to assign them *artificial* scheduling deadlines and offsets, such that, if every instance of a sub-task executes within its *window* defined by its offset and its deadline, then all precedence constraints are respected and the task completes before its end-to-end deadline (see Example 2 in Section 2). The problem of optimally assigning offset and deadlines to sub-tasks is very difficult, and many heuristics have been proposed in the literature.

Second, we observe that in EDF a sub-task may preempt another sub-task allocated on the same processor if and only if the relative deadline of the former *is strictly less than* the relative deadline of latter. Therefore, to avoid preemption on a given sub-task, we can assign it a relative deadline shorter than the sub-tasks that can potentially preempt it. However, in doing so, we have to guarantee that other constraints are respected as well (precedence constraints and schedulability constraints).

Summarizing, in this paper we propose a novel methodology for 1) assigning artificial deadlines and offsets to the sub-tasks of a C-DAG, and 2) allocating sub-tasks to computational resources, so to guarantee schedulability and to reduce the overall preemption cost.

2 System model

2.1 Background on GPU preemption

To motivate our work, and to justify our model, in this section we describe the way preemption is done in recent GPU architectures.

Early GPUs were designed to maximize throughput, therefore preemption was limited to special cases. For example, when executing graphical tasks, preemption was allowed at the boundary of draw calls (i.e. triangles). With the evolution of GPU architecture, they have been used for a wide variety of different computational tasks (not only graphical), and they were increasingly used to speed up the performance of complex real-time critical tasks.

For this reason, recent NVIDIA Pascal architectures [?] include a hardware component called *preemption engine* to support preemption for different types of tasks. In particular, we

distinguish between *graphical tasks* and *computational tasks*. For graphical tasks, in addition to *Draw-Boundary Preemption*, Pascal also offers *Pixel-Level Preemption*: when a preemption request is received, Pascal stops rasterizing new pixels, completes all operations for the pixels currently in the pipeline, and initiates a context switch. For computational tasks, Pascal offers *Thread-Level* and *Instruction-Level* preemption: in the first case, preemption is initiated when all threads of a sub-task complete on CUDA cores (notice that a sub-task may consist of a sequence of groups of parallel threads). In the second case, preemption is achieved by saving the context at the currently executing instruction in a similar way to classical CPU preemptions.

The finer the preemption level, the more time consuming preemption is. Preempting at the draw boundaries involves very little state information to save. However, preempting at the thread level includes larger costs; and at instruction (resp. pixel) level involves a massive amount of information, including contents of caches and the register files of all CUDA cores.

Capodieci et al. [**?**] calculated the order of magnitude on the cost of preemption in modern GPUs: an upper bound to the timing cost can be estimated in 50μsec for a computational task, and to 750μsec for a graphical task. However, the exact preemption cost depends on many different factors, and different sub-tasks may exhibit different preemption costs even within the same task. For the scope of this paper, we model the worst-case cost of preemption as a parameter of each sub-task.

2.2 Task model

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ denote a set of n tasks. Each task $\tau_i \in \mathcal{T}$ is represented by a tuple: (i) a Directed Acyclic Graph (DAG) denoted by $\mathsf{G}(\tau_i)$, (ii) its period $\mathsf{T}(\tau_i)$ and (iii) its end-to-end deadline $\mathsf{D}(\tau_i)$. When it is not important, we drop task index for the sake of simplicity.

Each task graph $\mathsf{G} = \{\mathcal{N}, \mathcal{E}\}$ is compound of a finite set $\mathcal{N}$ of nodes, and a finite set $\mathcal{E}$ of directed edges representing precedence order between graph nodes. No cycles are allowed in the graph. A node can be either a sub-task $v \in \mathcal{V}$ or a condition-control node $\mathsf{c} \in \mathcal{C}$, ($\mathcal{N} = \mathcal{C} \cup \mathcal{V}$). A sub-task v is an elementary sequential execution block and can be implemented as a single thread. A condition-control node c is a non-deterministic condition evaluated on line. According to the value of the condition one of two successors of c is selected[1].

To simplify the model and the presentation of our work, in this paper we restrict to **identical multiprocessor platforms**. In fact, addressing heterogeneous code requires a more complex task model where sub-tasks are tagged with the core on which they are allowed to execute (a GPU sub-task can only execute on GPUs, etc.). A more complex model accounting for heterogeneous multicore has been presented in [**?**]. However, we remark that the work presented in this paper can be easily extended to heterogeneous multicores, and it will be the subject of a future work.

A sub-task is characterized by its execution time $\mathsf{C}(v)$ and by the overhead to account when preempting sub-task v, called *preemption cost* and denoted by $\mathsf{pc}(v)$. 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.

n_i is an *immediate predecessor* of n_j if $\exists e(n_i, n_j) \in \mathcal{E}$. $\mathsf{pred}(n_i)$ denotes the set of all immediate predecessors of node n_i. n_i is a *predecessor* of a n_j if there exist a path from n_i

[1] We can easily express the case of a condition with multiple successors by defining successively branches of two-successors condition-control nodes.

Pre print

to n_j. If a sub-task 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, n_i is an *immediate successor* of n_j if n_j is an immediate predecessor of n_i. n_i is a successor of n_j if there is a path from n_j to n_i. If a node has no successors, it is called *sink* node. $\mathsf{src}(\tau)$ denotes the set of all source nodes in task τ, and $\mathsf{src}(\mathcal{T}) = \bigcup_{\tau \in \mathcal{T}} \mathsf{src}(\tau)$ denotes all source sub-tasks in task set $\mathcal{T}$, respectively. $|\mathcal{S}|$ denotes the cardinality of the set $\mathcal{S}$.

We consider a *sporadic task* model, therefore parameter T represents the minimum inter-arrival time between two instances of the same 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 $t + \mathsf{D}$. We assume *constrained deadline tasks*, that is $\mathsf{D} \leq \mathsf{T}$.

We define an execution pattern $\mathsf{p}_j(\tau)$ of task τ as one possible combination of all condition-control nodes in G. $\mathcal{P}(\tau)$ denotes the set of all execution patterns. We define the pattern execution time as follows:

$$\mathsf{C}(\mathsf{p}_j(\tau)) = \sum_{v_k \in \mathsf{p}_j(\tau)} \mathsf{C}(v_k) \tag{1}$$

We denote by $\mathsf{vol}(\tau)$ the volume of task τ computed as the maximum execution time among all its execution patterns: $\mathsf{vol}(\tau) = \max\{\mathsf{C}(\mathsf{p}_j(\tau)), j \in \{0, \cdots, |\mathcal{C}| + 1\}\}$. and we define the utilization of task τ as: $\mathsf{u}(\tau) = \frac{\mathsf{vol}(\tau)}{\mathsf{T}(\tau)}$. We define also the task set utilization as the sum of utilizations of all its tasks: $\mathsf{U}(\mathcal{T}) = \sum_{\tau \in \mathcal{T}} \mathsf{u}(\tau)$.

In this paper, we allow sub-tasks of the same task to be allocated onto different cores. We define by $\mathsf{V}^k(\tau)$ the set of all sub-tasks of task τ that are allocated on core k. τ^k denotes an isomorphic graph of τ where sub-tasks not belonging to $\mathcal{V}^k$ have null execution time and null preemption cost, and the sub-tasks belonging to $\mathcal{V}^k$ have the same execution time and preemption cost.

▶ **Definition 1.** *Let v_i be is an immediate predecessor of v_j. If v_i and v_j are allocated onto different cores, then we say that v_i is a "null predecessor" of v_j regarding its allocation core.*

Let $\pi(\tau)$ denote a *complete path*[2] of task τ, $\Pi(\tau)$ denote the set of all paths of task τ. We define the slack $\mathsf{Sl}(\pi, \mathsf{D}(\tau))$ along path π as:

$$\mathsf{Sl}(\pi, \mathsf{D}(\tau)) = \mathsf{D}(\tau) - \sum_{v \in \pi} \mathsf{C}(v) \tag{2}$$

We denote by $\pi^*(\tau)$ the path having the maximum sum of execution time of its sub-tasks, called *critical path*.

▶ **Example 2.** Consider the task described in Figure 1. Task execution times and preemption costs are described in Table 1.

The task in Figure 1 starts its execution with a single source sub-task v_1. Further, according to condition C_2, the task may follow the execution pattern v_4, v_8 to join the sink node v_9, or follow the other pattern and execute v_3 and further fork a parallel execution v_5

[2] Complete path is a path from one source node to one sink node

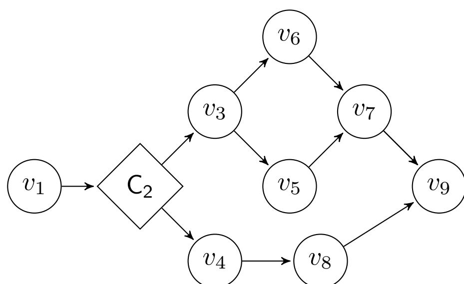


Figure 1 DAG Task example

v	$\mathsf{C}(v)$	$\mathsf{pc}(v)$	v	$\mathsf{C}(v)$	$\mathsf{pc}(v)$
v_1	4	4	v_3	9	3
v_4	3	2	v_5	5	4
v_6	10	0	v_7	4	1
v_8	5	1	v_9	2	1

Table 2 Parameters of the example.

and v_6, both tasks join back v_7 to finally execute the sink node v_9. Notice here that in this example source and sink nodes are unique, however this is not mandatory.

If sub-task v_3 gets preempted, the preempting sub-task has to account for 3 times units of preemption overheads. If the preempting task preempts sub-task v_5, it has to account 4 times units. It is important to account for the correct preemption cost to guarantee the respect of deadline constraints.

The volume $\mathsf{vol}(\tau)$ is equal to 34.

3 Deadlines and offsets assignment: Quick overview

In this paper we assume that sub-tasks are scheduled by EDF. Therefore, we need to assign *artificial* deadlines to sub-tasks. Additionally, several techniques have been proposed to enforce the respect of precedence constraints across different cores [?, ?, ?], among them also assigning artificial offsets to each sub-task. In fact, offsets and deadline are assigned so that, if each sub-task executes within its artificial offset and deadline, execution order is respected and the overall task respects its deadline.

We denote by $\mathsf{D}(v)$ the artificial deadline of v. The activation time of a task instance is the absolute time of the arrival of source sub-tasks instances. The artificial offset $\mathsf{O}(v)$ is the interval between the activation of the task graph and the activation of the sub-task. The absolute deadline of a sub-task instance is the activation time plus the artificial offset plus the artificial deadline $\mathsf{D}(v)$. We also define the *local deadline* as the interval between the task graph activation and the sub-task absolute deadline: it is computed as the sum of its artificial offset and its artificial deadline $(\mathsf{O}(v) + \mathsf{D}(v))$.

Figure 2 illustrates the relationship between the activation times, the artificial offsets and deadlines and local deadlines of the sub-tasks in Figure 1. We assume that v_1, v_3, v_9 have been allocated on the same core whereas v_5 and v_7 on another core and v_6 on a third core. Please notice that the right branch of the condition has been taken.

Most of the artificial deadline assignment algorithms distribute the slack computed in Equation (2) to different sub-tasks on every path in a task graph. However, they differ on the way this operation is done (referred as calculate_share in Equation (3)). In Section 7, we will describe the most popular techniques proposed in the literature, and in Section 6.2 we propose our own heuristics.

$$\mathsf{D}(v) = \mathsf{C}(v) + \mathsf{calculate_share}(v, \pi) \tag{3}$$

Once artificial deadlines are computed for all sub-task, we can automatically assign offsets as follows. As source nodes are activated as soon as the task is activated, their offset is set

Pre print

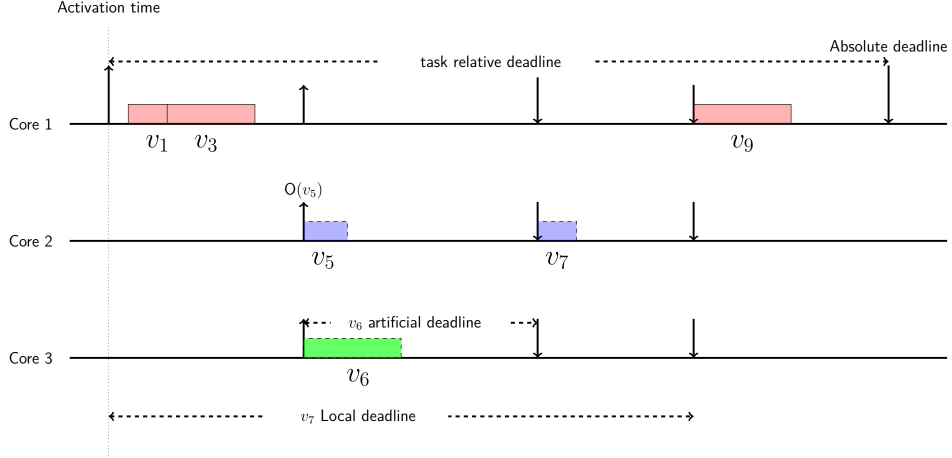


Figure 2 Example of offset and local deadline

to 0. For the other sub-tasks:

$$\mathsf{O}(v) = \begin{cases} 0, & \text{if } \mathsf{preds}(v) = \emptyset \\ \max_{v' \in \mathsf{preds}(v)} \{\mathsf{O}(v') + \mathsf{D}(v')\}, & \text{otherwise} \end{cases} \quad (4)$$

if the sub-task has more than one immediate predecessor, the offset is computed recursively as the maximum between the *local deadlines* of its immediate predecessors (Equation (4)).

Sub-task v 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 3.** *A task is feasible if all its sub-tasks are feasible.*

Proof. By 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 task is feasible. ◀

4 Preemption-aware analysis

In this paper, we consider a sporadic task system. We show in [?] how to compute and reduce preemption costs. For completeness, we state here Lemma 4 and Theorem 6.

▶ **Lemma 4** (Worst case preemption). *Let $\mathcal{V} = \{v_1, v_2, \cdots, v_K\}$ be a set of sub-tasks to be scheduled by EDF on a single core.*

Consider $\mathcal{V}^{\mathsf{pc}} = \{v'_1, v'_2, \cdots, v'_K\}$, where v'_i has the same parameters as v_i, except for its 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.

Lemma 4 accounts the maximum preemption cost in each sub-task execution time. If the system is schedulable in this configuration, then it is schedulable when considering preemption. The lemma is safe but very pessimistic. Pessimism can be reduced by using Theorem 6.

We highlight here the difference between $\mathsf{pc}(v_i)$, which represents the cost to preempt v_i, and pc^i which is the cost that v_i needs to account to preempt other sub-tasks.

▶ **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:*

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

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

▶ **Theorem 6** (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 (5) or (6).*

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

$$\mathsf{pc}^i = \max\{\mathsf{pc}(v) | v \in \mathcal{V} \setminus \mathcal{V}_\tau \wedge D(v) > D(v_i)\}; \tag{5}$$

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

$$\mathsf{pc}^i = 0 \tag{6}$$

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.

Proofs of Lemma 4 and Theorem 6 can be found in [?]. For space reasons, they are not reported in this paper.

To adopt pc^i notation for partitioned scheduling, we revise the symbol to $\mathsf{pc}^i(k)$ to denote the preemption cost of sub-task v_i when allocated onto core k.

▶ **Theorem 7** (Preemption aware volume). *Let $\mathcal{T}$ be a set of tasks, whose sub-tasks have already been allocated on a set of cores. Consider task $\tau \in \mathcal{T}$, and suppose that τ is allocated on a subset of cores denoted by $\mathcal{K}$. Let $\bar{\tau}^k$ denote the subset of sub-tasks of τ allocated on core $k \in \mathcal{K}$.*

Consider now a second configuration for task τ in which all sub-tasks of τ are allocated on the same core $j \in \mathcal{K}$, let us call such a configuration as $\bar{\bar{\tau}}^j$; the other tasks maintain the same configuration of allocation.

Then:

$$\sum_{k \in \mathcal{K}} (\mathsf{vol}(\bar{\tau}^k) + \sum_{v_i \in \bar{\mathcal{V}}^k} \mathsf{pc}^i(k)) \geq \mathsf{vol}(\bar{\bar{\tau}}^j) + \sum_{v_h \in \bar{\bar{\mathcal{V}}}^j} \mathsf{pc}^h(j). \tag{7}$$

In plain words, splitting the allocation of a task on several cores costs more in term of utilization than allocating all sub-tasks in one single core.

Proof. We start by proving that the following inequality is correct:

$$\sum_k \mathsf{vol}(\bar{\tau}^k) \geq \mathsf{vol}(\bar{\bar{\tau}}^j) \tag{8}$$

Pre print

We must consider two cases. First, consider the case of a task not containing any condition-control nodes. According to the definition of $\bar{\tau}^k$, sub-tasks of τ not allocated to k have null execution times. Thus,

$$\sum_{k} \mathsf{vol}(\tau^k) = \mathsf{vol}(\tau)$$

Now consider the case when the task contains conditional branches. Two branches of a conditional node will both contribute each on its core on the task volume, however if they are allocated on the same core, only one of will contribute to the volume. Therefore, the left part of Inequality (8) cannot be inferior to the right part.

Now we prove that the following inequality is also correct:

$$\sum_{k} \sum_{v_i \in \bar{\mathcal{V}}^k} \mathsf{pc}^i(k) \geq \sum_{v_h \in \bar{\bar{\mathcal{V}}}^j} \mathsf{pc}^h(j) \tag{9}$$

By assuming that all sub-tasks are allocated onto one core j, the maximal sequential subset $\mathcal{V}^M$ contains all sub-tasks, thus

$$\sum_{v^h \in \bar{\bar{\mathcal{V}}}} \mathsf{pc}^h(j) = \mathsf{pc}^M(j) + (|\mathcal{V}| - 1) \cdot 0 \tag{10}$$

where $\mathsf{pc}^M(j)$ is the preemption cost of v^M on j (Theorem 6).

Let us analyze the left hand side of Inequality (9).

$$\sum_{k \in \mathcal{K}} \sum_{v_i \in \bar{\mathcal{V}}^k} \mathsf{pc}^i(k) = \mathsf{pc}^M(j) + \sum_{z \in \mathcal{K} \setminus \{j\}} \sum_{v_i \in \bar{\mathcal{V}}^z} \mathsf{pc}^i(z). \tag{11}$$

As the last sum (in Equation (11)) is greater than or equal to 0, the inequality in (9) is proved.

By adding both Inequalities (8) and (9), the Theorem is proved. ◀

▶ **Theorem 8** (0-cost preemption). *Let $\mathcal{T}$ denotes a set of tasks allocated on same core and scheduled using preemptive EDF. $\mathcal{T}$ is scheduled without any preemption if and only if:*

- *all source sub-tasks have the same artificial deadline D^{src};*
- *all other sub-tasks have deadline shorter than D^{src}.*

Proof. *Only-if.* For a given sub-task v, from Theorem 6, and from the fact that all sub-task are allocated on the same processor, we derive that v can only be preempted by a source sub-task. Since all source sub-tasks have deadline larger than the deadline of v, thus no preemption can occur on v.

If. By contradiction, assume that at least one preemption occurs. The preempting sub-task must be a source, thus:

$$\exists v \in \mathcal{V}^{\mathsf{src}}, \exists v' \in \mathcal{V} \quad D(v) < D(v').$$

By assumption, the condition above is not possible, thus proving the second leg of the equivalence. ◀

According to Theorem 8, if it is possible to feasibly assign the same deadline to all source nodes, which is greater to all other resource of a task system allocated onto the same core without any null-predecessor, the preemption cost is 0.

Theorems 7 and 8 will be used to build our allocation and deadline assignment heuristics.

We present now preemption-aware deadline assignment using ILP and further allocation and deadline assignment heuristics.

5 ILP preemption-aware deadline assignment

Finding the optimal solution for overall problem (deadline assignment and allocation that minimizes preemption cost) is very complex due to the extremely large space of parameters to explore.

In this section, we propose a model based on Mixed Integer Linear Programming (MILP) to assign artificial deadlines to sub-tasks, assuming a single core. The MILP model proposed here can be used by the heuristics for allocation on multicore systems of Section 6.1. In Section 6 we will also propose heuristics for deadline assignment as an alternative to the MILP formulation.

5.1 Decision variables and objective function

Let $\mathsf{seq}_l(\tau)$ denote the lth maximal sequential subset of task τ on a given core and let v_l^M denote its v^M, selected according to Theorem 6. Let $\mathsf{D}(v)$ be an integer decision variable expressing the deadline of sub-task v.

Let $p_{(v_i,v_j)}$ be a binary decision variable to express the ability of sub-task v_i to preempt sub-task v_j. According to Theorem 6, only sub-tasks v_l^M in all maximal sequential subsets $(\forall l)$ have to account for maximal preemption costs, all the other sub-tasks account for 0 preemption cost. Thus, the variable p is defined for the combination of sub-task v_l^m for every maximal sequential subset and all sub-tasks as follows:

$$\forall l, \forall v_j \in \mathcal{V} \setminus \mathsf{V}^\tau, p_{(v_l^M,v_j)} = \begin{cases} 1, & \mathsf{D}(v_i) < \mathsf{D}(v_j) \\ 0, & \mathsf{Otherwise} \end{cases} \tag{12}$$

Where $\mathcal{V} = (\bigcup_{\tau\in\mathcal{T}} \mathcal{V}(\tau))$ is the set of all sub-tasks in task set and τ is the task to which v_l^M belongs.

The objective function tries to reduce as much as possible the preemption cost, thus it is modeled as follows:

$$\mathsf{Minimize} \sum_{\tau\in\mathcal{T}} \sum_{l} \sum_{v_j\in\{\mathcal{V}\setminus\mathcal{V}(\tau)\}} \mathsf{pc}(v_j) \; p_{(v_l^M,v_j)} \tag{13}$$

5.2 Preemption and deadline assignment constraints

To express the ability of v_l^M to preempt v_j, we impose the $\mathsf{D}(v_l^M) < \mathsf{D}(v_j)$ that can be linearized as follows:

$$\begin{aligned} &\mathsf{BIG_NUM} \;\; p_{(v_l^M,v_j)} + \mathsf{D}(v_l^M) - \mathsf{D}(v_j) >= 0, \qquad v_i \in \mathsf{V}^\tau \\ &\mathsf{BIG_NUM} \;\; p_{(v_l^M,v_j)} - \mathsf{BIG_NUM} + \mathsf{D}(v_l^M) - \mathsf{D}(v_j) <= 0 \end{aligned} \tag{14}$$

if $\mathsf{D}(v_l^M)$ is shorter than $\mathsf{D}(v_j)$, the $p_{(v_l^M,v_j)}$ is set to 1 so both constraints in Equation (14) can be respected, otherwise it is set to 0.

Other constraints can be linearized in a similar way. Due to space constraints, we do not report here the rest of the linearized constraints.

For each task, we need to impose that the sum of deadlines in each complete path does not exceed the task deadline as follows:

$$\forall \tau, \forall \pi \in \Pi(\tau), \sum_{v\in\pi} \mathsf{D}(v) \leq \mathsf{D}(\tau) \tag{15}$$

Pre print

We highlight that if sub-task v is present in several paths, it has only one decision variable $\mathsf{D}(v)$. Moreover, the deadline of each sub-task need to be greater than its execution time (slack need to be greater or equal to 0), thus $\mathsf{D}(v) \geq \mathsf{C}(v)$.

5.3 Feasibility constraints

In addition to the above constraints, we need to impose the schedulability of the system.

▶ **Theorem 9** (Single core feasibility). *Let $\mathcal{T}$ be a set of task graphs allocated onto a single-core core. Task set $\mathcal{T}$ is schedulable by EDF if and only if:*

$$\sum_{\tau \in \mathcal{T}} \mathsf{dbf}(\tau, t) \leq t, \forall t \leq t^* \tag{16}$$

where dbf is the demand bound function [?] for a task graph τ in interval t. The demand bound function is computed as the maximum 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:

$$\mathsf{dbf}(\tau, t) = \max_{v \in \tau} \sum_{v' \in \tau} \left\lfloor \frac{t - \tilde{\mathsf{O}}(v') - \mathsf{D}(v') + \mathsf{T}(\tau)}{\mathsf{T}(\tau)} \right\rfloor \mathsf{C}(v') \tag{17}$$

where

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

The dbf constraints can be expressed into our ILP as follows:

$$\mathsf{C}(v')^{\mathsf{pc}} = \begin{cases} \mathsf{C}(v'), & \text{if } v' \notin \{v_l^M, \forall l\} \\ \mathsf{C}(v') + \max_{\forall j, v_j \notin \mathsf{V}^\tau} \{p_{(v', v_j)} \cdot \mathsf{pc}(v_j)\}, & \text{otherwise} \end{cases} \tag{18}$$

Schedulability can be tested by applying the constraint presented in Equation (16) for all values of t between 0 and the tasks hyper-period by replacing $\mathsf{C}(v')$ in Equation (17) by the execution time $\mathsf{C}(v')^{\mathsf{pc}}$ computed in Equation (18).

It is time consuming to compute the exact dbf as in Equation (17). Several dbf approximations has been proposed in the literature of real-time systems. One of simplest for conditional DAGs has been presented by Baruah et al. in [?]. We enhanced this approximation to take into account the preemption costs. Our modification is described in Equation (19). First let us define $\mathsf{vol}(\tau)^{\mathsf{pc}}$ as the task volume when accounting for preemption overheads :

$$\mathsf{vol}(\tau)^{\mathsf{pc}} = \mathsf{vol}(\tau) + \sum_{l} \mathsf{max}_{(\forall j, v_j \notin \mathcal{V}(\tau))} \{p_{(v_l^M, v_j)} \cdot \mathsf{pc}(v_j)\}$$

The dbf can be approximated as:

$$\mathsf{dbf}^*(\tau, t) = \begin{cases} 0, & \text{if } t < \mathsf{D}(\tau) \\ \mathsf{vol}(\tau)^{\mathsf{pc}} + u_i^{pc} \times (t - \mathsf{D}(\tau)), & \text{otherwise} \end{cases} \tag{19}$$

Where $u_i^{\mathsf{pc}} = \frac{\mathsf{vol}(\tau)^{\mathsf{pc}}}{\mathsf{T}}$ is the task utilization when taking into account preemption costs.

The approximation described here is only valid when all sub-tasks of the same task are allocated on the same processor. If one or more sub-tasks of the same task are allocated on a different processor, the approximation does not guarantee the respect of the artificial deadlines of the sub-tasks, and hence of the precedence constraints. Therefore, if all sub-tasks of a task are allocated onto the same processor, we use the approximation; otherwise we resort to the exact dbf.

6 Allocation and deadline assignment heuristics

In this section, we present the different heuristics used in this paper to assign deadlines and allocate sub-tasks onto processors so to minimize the impact of the preemption cost on the schedulability of the system. As mentioned in Section 4, to achieve optimal and sub-optimal solutions, deadline assignment of different sub-tasks of different tasks have to be considered at the same time, because the cost of preemption is a function of all sub-tasks allocated on the same core.

Thus, our algorithm consists of 3 steps: (i) group tasks (sub-tasks) by means of Algorithm 2, (ii) assign deadlines using either single core ILP deadline assignment as shown in Section 5, or one of the heuristics described in Algorithm 3 and (iii) re-adjust task groups and allocate each group onto a core. This 3-step approach is described in Algorithm 1.

6.1 Allocation heuristics

Algorithm 1 starts by clustering tasks (Line 3) into separate groups, such that each group has total utilization strictly greater than 1 (except the last one).

If the number of groups is greater than the number of available cores, the system is non feasible as it will be proved in Lemma 11.

Further, task groups are sorted in a non-increasing order of total utilization (Line 7). For each task group, we first assign deadlines using either ILP, or one of the preemption-aware heuristics described later on.

Algorithm 1 Allocation algorithm

```
1:  input : T: set of conditional DAG tasks
2:  parameters :ASSIGN_PARAM(ILP, PREEMP_A, FAIR, PROP)
3:  clusters = cluster_taskset(T)
4:  if |clusters| > m then
5:      return FAIL
6:  end if
7:  sort_clusters(clusters)                         ▷ sort by total utilization
8:  for T^c ∈ clusters do
9:      feasible = False
10:     removed = ∅
11:     while (not feasible) do
12:         assign_deadlines(T^c,ASSIGN_PARAM)
13:         feasible = test_feasibility(T^c)
14:         if (not feasible) then
15:             removed += omit_subtasks(T^c)
16:         end if
17:     end while
18:     insert(removed,T^|clusters|)
19:     if |clusters| > m then
20:         return FAIL
21:     end if
22: end for
23: assign_task_groups_to_cores()
24: return SUCESS
```

Further, for each group the algorithm tests the feasibility of the sub-tasks in the group (except if the ILP approach is used, because it always produces a feasible task set). If the system is not feasible, a sub-task is selected to be removed from the current group (line 15).

When a sub-task is removed from the group, it is replaced by a null sub-task with execution time and preemption costs equal to 0. The removed sub-task is inserted into the removed list and will be allocated to another group later on.

Selection of the sub-task to be removed is done by first selecting a random task in the group, and then a sub-task in the given task by using one of the following heuristics:

- **random heuristics:** The sub-task to omit is selected randomly from all sub-tasks in the task.
- **preemption-aware heuristic:** This heuristic behaves differently when it is applied to a task with no null-sub-tasks, and when it is applied to a task containing at least one null sub-task.
  In the first case, the algorithm selects the sub-task with the largest execution time among all sub-tasks that do not belong to the critical path of the task. If all sub-tasks in the task belong to the critical path, then the last one in the critical path is chosen.
  In the second case (presence of at least one null sub-task), the heuristic tries to avoid creating too many *sequential maximum subsets* (holes). Hence, it looks for null sub-tasks and removes one of their predecessor or successors. Among all candidates, it gives priority to the one with the largest execution time and that does not belong to the critical path. Notice that, when two consecutive sub-tasks are removed, their deadline and offsets might be later reassigned when moving them to a different group.

The system tests iteratively the schedulability until finding a feasible schedule by invoking test_feasibility. test_feasibility uses dbf based tests according to two situations: If all sub-tasks of the same sub-task are allocated on the same core, it uses the dbf approximation described in Equation (19), otherwise it uses the exact dbf described in Equation (17). Since every time we remove a sub-task, the while loop (Line 11) will converge to the case of no sub-tasks, which is obviously feasible. The non-allocated sub-tasks which are contained in removed_task list are added to the last task group to be allocated in the future iterations. Further, the algorithm invokes the clustering algorithm to add the removed sub-tasks (see Algorithm 2).

As a consequence, new clusters may be produced. The algorithm fails at any time the number of clusters is greater than the number of available cores.

We now describe the clustering algorithm. First of all, tasks are sorted according to the following order relationship.

▶ **Definition 10** (Task order function). *For a task τ_i, we denote by $\gamma(\tau_i)$ the* average artificial deadline *of task τ_i, computed as:*

$$\gamma(\tau_i) = \frac{\mathsf{D}(\tau_i)}{\max_{\pi \in \Pi(\tau_i)} |\pi|}$$

where $|\pi|$ denotes the number of sub-tasks in path π, and $\Pi(\tau_i)$ is the set of all paths in τ_i.

Let τ_i, τ_j be two tasks. The order relationship $\tau_i > \tau_j$ is defined as

$$\tau_i > \tau_j \implies \gamma(\tau_i) > \gamma(\tau_j). \tag{20}$$

Notice that $>$ sorts tasks according to their *average deadline*. If two tasks have similar average deadline, then it is likely possible to group them on the same processor: then, as stated by Theorem 8, we can reduce the cost of preemption by assigning the same deadline

Algorithm 2 cluster_taskset

```
1:  input : 𝒯: set of conditional DAG tasks
2:  Output : clusters: set of tasksets
3:  sort_tasks(𝒯)                                   ▷ By > relation
4:  𝒯^current = ∅
5:  for τ ∈ 𝒯 do
6:      if u(τ) ≥ 1 then
7:          add_cluster({τ}, clusters)
8:      else
9:          add_task(τ, 𝒯^current)
10:         if U({𝒯^current} ≥ 1 then
11:             add_cluster(𝒯^current, clusters)
12:             𝒯^current = ∅
13:         end if
14:     end if
15: end for
16: return clusters
```

to their source sub-tasks, and shorter deadlines to following sub-tasks. On the contrary, if we group tasks with very different average deadline on the same core, it is unlikely to assign the same large deadline to all source sub-tasks, leading to a large preemption cost.

Once the tasks have bee sorted according to their average deadline, Algorithm 2 adds tasks one by one to a group until total utilization is greater than 1. Tasks having an utilization greater than 1, are put in their own group[3]. When a group has an utilization greater than 1, a new cluster is created.

▶ **Lemma 11** (Necessary test). *Let $\mathcal{T}$ be a task set and M the number of clusters of $\mathcal{T}$ obtained by algorithm 2.*

If $M > m$, then the task system is not feasible.

Proof. Trivially, if $M > m$, the total utilization exceeds m, so the system is not schedulable. ◀

6.2 Deadline assignment heuristics

The deadline assignment step has a large impact on schedulability and preemption overheads. In fact, a *good* deadline assignment technique can allow us to avoid costly preemption or even reduce the preemption cost to zero as proven in Theorem 8. In this section, we will show how to assign deadlines while taking into account preemption costs.

First we start by defining the *preemption heaviness* of sub-task v as :

$$w(v) = \frac{\mathsf{pc}(v)}{\mathsf{C}(v)}$$

According to their heaviness, we define three classes of sub-tasks:

- **Non preemptive:** sub-tasks with preemption heaviness greater than or equal to 1. Preempting these sub-tasks costs more than waiting for their completion, so they must be assigned the same shortest possible deadline.

[3] We observe that this approach is similar to the federated-scheduling framework.

Pre print

- **Heavy:** sub-tasks with preemption heaviness greater then a given threshold α and less than 1.
- **Preemptive:** sub-tasks with preemption cost less than or equal to α.

Algorithm 3 Preemption-aware deadline-assignment.

```
1: input : T: set of conditional DAG tasks, method: Int
2: base_deadline = 0
3: for (τ ∈ T) do
4:     b_d = max(max_{v∈τ} C(v), D(τ)/max_depth(τ))
5:     base_deadline = max(b_d, base_deadline)
6: end for
7: reduce_deadlines(T)
8: switch method do
9:     case FAIR_DEADLINE :
10:        ∀τ, fair_deadline_single_task(τ)
11:    case PROP_DEADLINE :
12:        ∀τ, prop_deadline_single_task(τ)
13:    case PA_DEADLINE:
14:        ∀τ, pa_deadline_single_task(τ)
```

Algorithm 3 assigns deadlines by taking into account preemption costs. The algorithm has three main steps: the first assigns deadlines to all sources sub-tasks, the second one re-adjusts, if necessary, the source sub-tasks deadlines, and the final step assigns deadlines to the other sub-tasks.

The first step starts by computing the base_deadline. It represents the maximum deadline that source sub-tasks may be assigned, so to eliminate all possible preemption according to Theorem 8. It is computed as the maximum between the largest execution time among all sub-tasks of all tasks in the group, and the maximum average deadline $\gamma(\tau)$ among all tasks in the group.

Assigning base_deadline to source sub-tasks does not ensure schedulability, thus a necessary test is applied to quickly eliminate unfeasible solutions.

The necessary test computes the slack in the critical path of every tasks, and checks that it is still positive:

$$\forall \tau, D - \mathsf{base_deadline} > \sum_{v^* \in \pi^*} C(v^*) - C(v') \tag{21}$$

where v' is the source of the critical path $\pi^*(\tau)$.

If the test in the previous equation fails, the task group is not feasible when assigning the base_deadline to source sub-tasks. Therefore base_deadline is decremented iteratively until Condition 21 becomes true.

After this iteration, the deadline of each source sub-task is set to the maximum between the base_deadline and the sub-task execution time (the per-sub-task slack can not be negative). If the execution time of the critical path is less than the task deadline (necessary condition even on a unlimited number of cores), this second step will converge.

The third step assigns deadlines to the rest of the sub-tasks. It may use the already existing heuristics such as fair, proportional deadline assignment described in Section 7 or the preemption-aware deadline assignment described in Algorithm 4. In the case a group

has no null sub-task, all heuristics behave in the same way regarding preemption cost (see Theorem 6). However, in case of a null sub-task, a well-designed heuristic can reduce preemption cost.

Algorithm 4 is a novel heuristic for assigning artificial deadlines. It starts by selecting all heavy sub-tasks. Further, it selects d_min, the minimum deadline that has been already assigned in a previous step. d_min is an upper bound to heavy sub-task deadlines, otherwise these tasks can be preempted by at least one source sub-task. Further, another upper bound d_b_min is computed as the minimum $\gamma(\tau)$. This step is similar to source sub-tasks deadline assignment, however the minimum is selected instead of the maximum. In fact, if this heavy tasks deadline is greater than this value, at least one of the sub-tasks for which the deadline is not-yet assigned will have a smaller deadline, hence be able to preempt at least one heavy task.

The minimum between the two upper bounds d_min and d_b_min is selected as a new upper bound. Further we ensure that this upper bound is greater than the maximum sub-task execution time (Line 6) of heavy sub-tasks. If it is the case, the maximum execution time among heavy sub-tasks is selected. Further, heavy sub-tasks deadlines are reduced in a way similar to source sub-tasks deadline assignment. Further, the deadlines of light non-source sub-tasks are assigned using either fair or proportional deadline (line 11). We highlight that already assigned deadlines are not reassigned again (except when the deadlines are canceled inside the omit function in Algorithm 1).

Algorithm 4 pa_deadline_single_task(τ)

```
1:  input : 𝒯: set of conditional DAG tasks, method
2:  heavy_list = select_tasks_with w(v) > α
3:  d_min = min{D_v}                                   ▷ Already assigned deadlines
4:  d_b_min = min{ D(τ_i) / max_{π∈Π(τ_i)} |π| }       ▷ for all τ ∈ 𝒯
5:  c_max = max_{v∈heavy_list}{C(v)}
6:  d_c = max{c_max, min{d_min, d_b_min}}
7:  for v ∈ heavy_list do
8:      D(v) = d_c
9:  end for
10: reduce_deadlines(𝒯)
11: switch method do
12:     case FAIR_DEADLINE :
13:         ∀τ, fair_deadline_single_task(τ)
14:     case PROP_DEADLINE :
15:         ∀τ, prop_deadline_single_task(τ)
16: return SUCCESS
```

7 Related work

Several task models have been proposed in the literature to express data dependency and parallelism within a real-time task, most of them are based on DAGs (directed acyclic graphs), e.g. [?, ?, ?, ?]. DAGs have received a lot of attention in preemptive global (e.g. [?, ?, ?, ?]) and partitioned (e.g. [?, ?, ?]) scheduling.

One of the most effective techniques to schedule DAGs on multicore platforms is to assign

intermediate deadlines and offsets[4] to sub-tasks in order to enforce precedence constraints. The advantage of such techniques is that a set of dependent sub-tasks is converted into a set of independent sub-tasks with offsets, for which well-know and efficient schedulability analysis exists. However, optimal assignment of intermediate deadlines and offset is a difficult problem.

The most popular heuristic algorithms are based on the idea of dividing the slack time along each path among all its sub-tasks according to some simple rule. Two among the many alternative heuristics are:

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

$$\mathsf{calculate_share}(v, \pi) = \frac{\mathsf{Sl}(\pi, \mathsf{D}(\tau))}{|\pi|} \tag{22}$$

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

$$\mathsf{calculate_share}(v, \pi) = \frac{\mathsf{C}(v)}{\mathsf{C}(\pi)} \cdot \mathsf{Sl}(\pi, \mathsf{D}(\tau)) \tag{23}$$

The share of every sub-task is computed according to a non-increasing order of paths by cumulative execution time. Authors of [**?**] studied the deadline assignment problem in distributed real-time systems. They formalized the problem and identified the cases where deadline assignment methods have a strong impact on system performances. They proposed *Fair Laxity Distribution* (FLD) *and Unfair Laxity Distribution* (ULD) and studied their impact on the schedulability. In [**?**], authors analyze the schedulability of a set of DAGs using global EDF, global rate-monotonic (RM), and federated scheduling. Yifun wu et al. in [**?**] propose techniques to set offsets and deadlines using ILPs. Qamhieh et al. in [**?**] proposed a sufficient schedulability test of a set of DAG tasks onto a multicore platform. They assigned intermediate deadlines and offsets according to path length using techniques similar to Equation (21).

All the above-cited works consider preemption costs to be negligible. In the presence of high and variable preemption cost, the previous techniques may be ineffective. A radical approach is to consider non-preemptive systems as those found in [**?**, **?**]. Bertogna et al. [**?**, **?**, **?**, **?**] propose limited preemption models as a viable alternative between the two extreme cases of fully preemptive and non-preemptive scheduling.

To reduce the preemption costs, two main techniques have been proposed. In [**?**], a task can not be preempted up to a given priority, called *Preemption Threshold*. Thus, each task is assigned a priority and a preemption threshold, and the preemption takes place only when the priority of the arriving task is higher than the threshold of the running task. On the other hand, Baruah in [**?**] proposed *deferred preemption*. According to this method, when a high priority task is activated on a core where a low priority task is running, a function is evaluated on-line to define the longest interval the current task can continue to execute non-preemptively without compromising the respect of real-time constraints. Finally, fixed preemption points have been introduced to forbid a task to get preempted out of well-defined preemption points specified in the code. A complete survey about such techniques can be found in [**?**].

[4] In this paper, we refer to them as *artificial* deadlines and offsets.

For all these preemption-aware techniques the preemption cost itself is still considered negligible. Some of these techniques can be used to reduce/avoid *on-line* preemption cost, however they cannot be used in the case of very high preemption costs.

Phavorin et al. in [?] have shown that single processor EDF is not optimal when considering preemption costs. This work is the closest one to our work. However, they use techniques to build off-line static schedules, whereas our techniques is based on EDF. Moreover, we use assume C-DAGs, thus our sub-tasks are dependent, whereas Phavorin et al. consider only independent L&L tasks. Finally, they consider a single core platform, we are interested in partitioned scheduling over multicores.

8 Results and discussions

In this section, we evaluate the performance of our deadline assignment heuristics and allocation strategies. We compare the combination of several heuristics proposed in this paper against fair and proportional deadline assignment combined with the bin-packing heuristics: Best Fit (BF) and Worst Fit (WF). We adapted BF and WF to take into account the preemption costs evaluated using Theorem 6.

We compare schedulability rate, preemption cost reduction efficiency and practical complexity of the techniques cited above on a platform of 4 identical cores.

8.1 Task Generation

We conducted our experiments on a large number of randomly generated task sets. First, the generation algorithm starts by producing n utilizations whose sum is equal to x (varies in every experiment) by using the UUniFast [?] algorithm, n is randomly selected between $[8, 12]$.

For each utilization, the algorithm uses again UUniFast-discard to distribute the task utilization to n_v sub-task, so to obtain per-sub-task utilizations. n_v is randomly selected between 7 and 15. The sub-task utilization is multiplied by the task period to compute the sub-task execution time. Further, two approaches to assign preemption costs and define task topologies are used:

1. In the first approach, the per-sub-task preemption cost is generated randomly according to a probability $\mathsf{P}_{\mathsf{pc}} = 0.7$: thus 70% of the sub-tasks are assigned preemption cost in the interval $[0\%, 20\%]$ of the sub-task execution time; and 30% of the sub-tasks have preemption cost in interval $[70\%, 120\%]$ of their execution time. Sub-tasks are connected randomly without creating cycles.
2. In the second approach, the per-sub-task preemption cost is computed as a fixed percentage of the sub-task execution time. In the different experiments, this fixed cost is chosen in the list $\{0\%, 30\%, 60\%\}$. Further, sub-tasks are randomly divided assigned intoto $L = 5$ layers. Sub-tasks of layer l are randomly connected only to sub-tasks of layer $l + 1$.

We remark that the second approach has been designed to stress our heuristic. In fact, as the number of sub-task is set between 7 and 15 and it is randomly distributed across 5 layers, the number of sub-tasks in a layer can very often be equal to 1. This is actually unfavorable for our heuristic because, in the case a task is not feasible on a single processor, the algorithm is forced to split the critical path and generate several maximal sequential subsets.

Moreover, as the load is *fairly* distributed between layers, fair deadline assignment heuristic have better chances to achieve *good* deadline assignment compared to our heuristic

Pre print

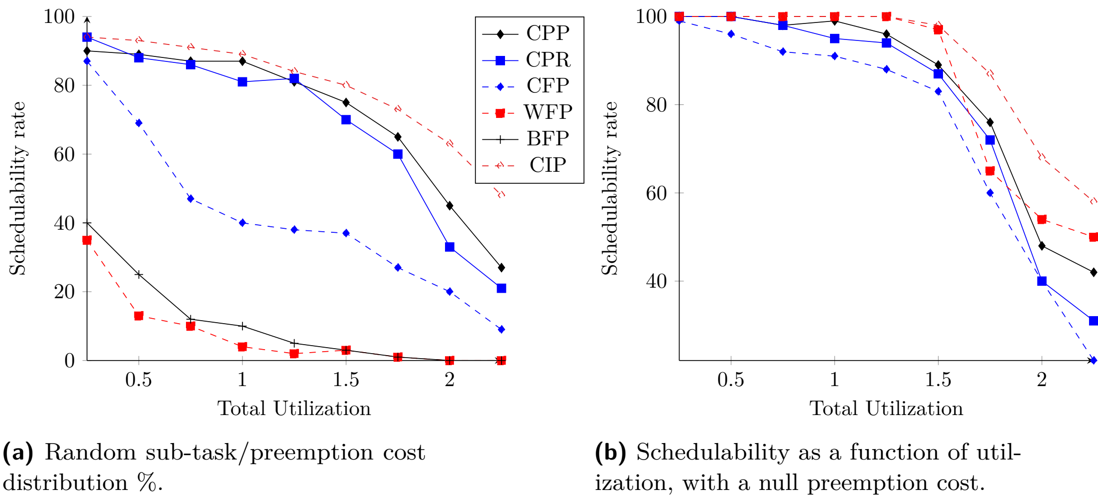
that assigns large deadlines to source tasks and over-constrains the sub-tasks in the following layers. The per-task utilization is limited to 60% in the second generation method.

Once a DAG has been generated, we transform it into a C-DAG by randomly inserting conditional nodes between sub-tasks to simulate tasks' dynamic behavior, thus creating new paths without increasing the task utilization. The task period is selected randomly from a predefined list of periods $\{50, 80, 100, 150, 200, 300, 400, 500, 600, 800, 1200\}$, so to establish an upper bound to the hyperperiod. The task deadline is selected randomly in the interval $[0.75 \cdot \mathsf{T}, 0.85 \cdot \mathsf{T}]$.

8.2 Simulation results and discussions

We vary the baseline utilization from 0 to 4 by a step of 0.25. Every point in the following figures represents the average value of 100 experiment. In all experiments, the standard deviation is between 2% and 3% of the average value (except for Figure 5a, which will be discussed later on).

The results presented in this section are the combination of several heuristics proposed in this paper and in the literature. Each combination is denoted by 3 letters: (i) the task allocation heuristic can be either C for clustering, B for best fit or W for worst fit; (ii) the deadline assignment heuristic can be either capital P for preemption aware heuristic, or F for fair deadline assignment, or I for ILP; (iii) sub-task selection (Line 15 in Algorithm 1), it can be either R for random heuristic or P for preemption aware selection heuristic.



(a) Random sub-task/preemption cost distribution %.

(b) Schedulability as a function of utilization, with a null preemption cost.

Figure 3 Schedulability rate VS total utilization : Random and controlled generation.

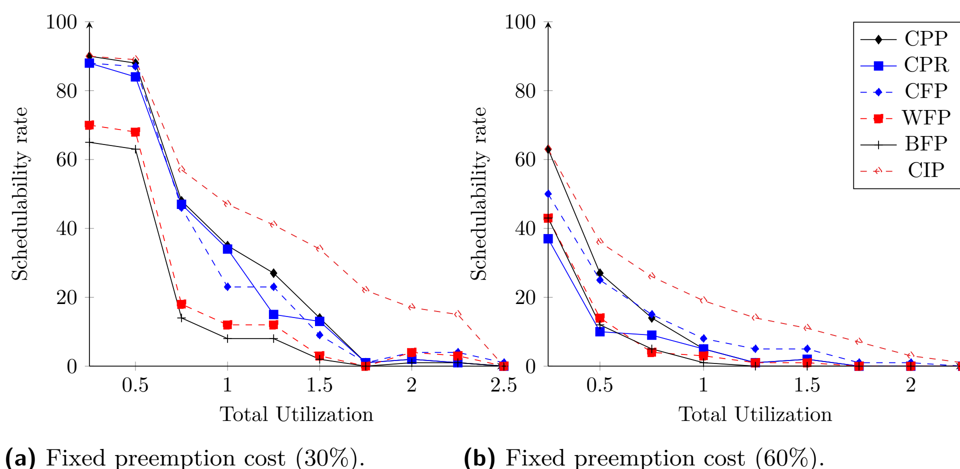
In Figure 3a, tasks are generated using the first method. The figure shows the schedulability rate of the following combinations: CPP, CPR, CFR , WFP, BFP and CIP heuristics as a function of total utilization.

CIP uses clustering for allocation, preemption aware selection heuristic to omit sub-tasks and an exact solution to assign per-group deadlines, thus it presents the highest schedulability rates. CPP and CPR combines the preemption-aware heuristics proposed in this paper, and they presents very high schedulability rates. CFP takes advantage of the clustering properties that allow grouping sub-tasks with a "fair" possible deadline distribution.

Even when combined with optimal deadline assignment techniques, BF and FF show a low performances, because (i) preemption cost depends on the allocation, (ii) these heuristic

add sub-tasks individually, and no global state is considered.

In Figure 3b, tasks are generated using the second method and preemption cost equal to zero. The goal of the experiments reported in this figure is to study the effectiveness of our approaches in the absence of preemption costs. The figure reports the schedulability as a function of total utilization for CPP, CPR, WFP (BFP is omitted to avoid surcharging the figure as it is outperformed by WFP). With the increase of the workload, heuristics based on fair deadline assignment perform slightly better than preemption-aware based heuristics. In fact, the preemption-aware heuristics try to reduce the number of preemptions even when the preemption cost is null, and they tend to over-constrain unnecessarily the non-source sub-tasks. CIP outperforms all other heuristics, as it combines clustering with optimal deadline assignment.



(a) Fixed preemption cost (30%). **(b)** Fixed preemption cost (60%).

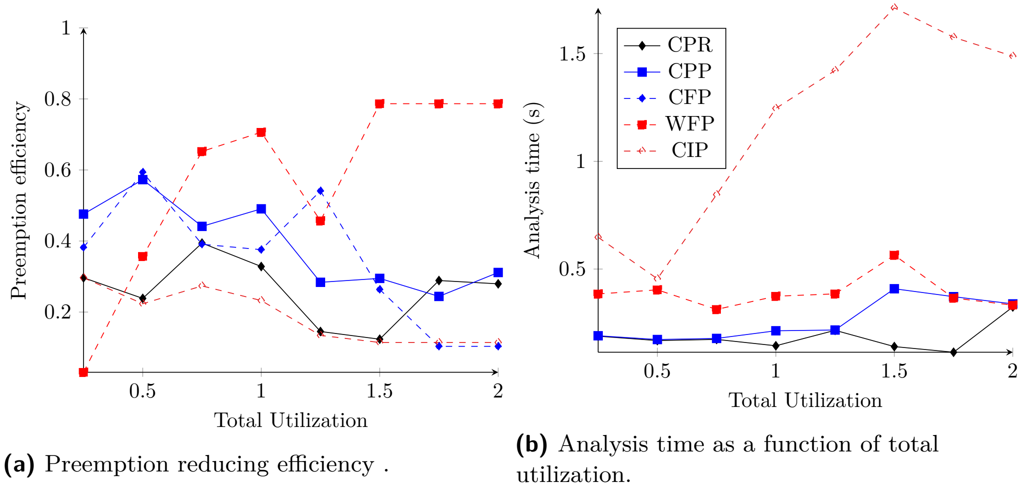
Figure 4 Schedulability rate VS total utilization with fixed preemption overhead.

In Figure 4a and Figure 4b, task are generated according to the second method, with a fixed preemption cost of 30% and 60% of the sub-task execution time, respectively. The schedulability falls sharply even for low utilization rates: as the preemption cost increases, reducing the number of preemptions becomes essential for schedulability. Again, CIP outperforms all other heuristics. We observe that at low workloads, CPP and CPR provide performance close to CIP, however as the workload increases, CIP dominates all the others. In contrast, BF and WF schedulability rates fall more sharply and they are not able to achieve more than 10% schedulability rate even at a very low utilization.

Figure 5a represent the *preemption cost reducing efficiency* as a function of total utilization for schedulable task sets. The preemption reduction efficiency of vertex v_i is defined as: $\epsilon(v_i) = \frac{\mathsf{pc}^i(h)}{\mathsf{pc}(v_i)}$. In the figure we show the total preemption efficiency (the sum for all sub-tasks). It quantifies the effectiveness of a heuristic in reducing the preemption costs: the lower, the better. Only schedulable task sets are considered in this figure. Please notice that, since the number of schedulable task sets decreases substantially for high utilization, the standard deviation increases from 10% on the left part of the figure, to more than 50% in the right part of the figure, so those points are less meaningful.

As expected, in general clustering based approaches are more effective in reducing the preemption cost compared to WF-based approaches. The inversion on the very low utilizations can be explained by observing that WF distributes tasks onto different cores first, thus sub-tasks of the same task may be allocated onto the same core with less concurrence. On the other hand, clustering based approaches tend to cluster all tasks onto the same core,

Pre print



(a) Preemption reducing efficiency .

(b) Analysis time as a function of total utilization.

Figure 5 Preemption reduction efficiency and analysis execution time

thus the preemption cost is higher for very low utilizations. When the load increases, WF fits more tasks on the same core, potentially increasing the number of preemption compared to our heuristics that are designed to reduce the number and the cost of preemption. CIP is more efficient as it assigns optimal deadlines with the goal of minimizing preemption costs reduce.

Figure 5b shows the execution time of the analysis for schedulable tasks set under clustering based and WF based approaches. Even if the theoretical complexity of clustering-based approaches seem to be greater than the classical bin-packing heuristic, in practice they are more efficient. In fact, clustering based approaches group tasks only once and assign them artificial deadlines and offset before proceding with allocation, wheras, when using bin-packing based heuristics, at each allocation the deadline assignment algorithm is invoked again. We observe that the good performance shown by CIP are not for free. In fact, the execution time increase considerably with the increase of utilization. The clustering heuristic is called more often to re-assign non-allocated sub-tasks to cores.

9 Conclusions and future work

In this paper we propose technique to allocate C-DAG tasks onto identical multicore platforms, by accounting for task preemption. Since the cost of preemption can be very large in modern GPU, our technique reduces the number of preemptions by setting appropriate artificial deadline to sub-tasks and by allocating tasks with similar deadlines on the same core. Results of our extensive synthetic experiments show a significant reduction on total preemption cost when combining preemption-aware allocation with preemption-aware deadline assignment techniques.

The work presented in this paper can be easily extended to heterogeneous platforms with multiple ISA and multiple capacities as in [?]. We also plan to investigate other premption-reducing techniques, such as the deferred preemption proposed by Baruah et al. [?]. Finally, we plan to extend our approach to separately take into account saving and loading context, thus allowing tighter bounds on preemption costs.