1 ALLOCATION AND ANALYSIS

In this work, we consider partitioned scheduling. Each engine has its own scheduler and a separate ready-queue. Subtasks 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 allows to reduce the overhead due to thread migration compared to global scheduling. The analysis presented here is modular, so engines may have different scheduling policies. In this paper, we restrict to preemptive-EDF.

Let assume n task specifications to be allocated onto m engines. Only concrete graphs can be allocated to engines. However, we often use *allocating task specification* to express the whole process of allocating one of its concrete digraphs.

1.1 Alternative patterns

Definition 1. *Concrete task τ is schedulable if and only if all its sub-tasks at arrival j finish their execution within the time interval $[a_{ij}, a_{ij} + \mathcal{D}(i)], \forall j \in N$ where a_{ij} is the j^{th} instance arrival time.*

Definition 2. *Task specification τ_i is schedulable, if :*

$$\exists \tau_{id} \in \Omega_i, \tau_{id} \textit{is schedulable} \tag{1}$$

According to 2, it is sufficient to find one concrete task τ_d among all the concrete tasks of task specification τ, so it can be feasibly allocated. Thus, it is necessary to generate and test the allocation of all concrete tasks a given task specification. Algorithm shows how concrete tasks are generated.

Algorithm 1 generate_concrete_task

```
1: input: τ Task
2: alta_node = select_one_alternative_node()
3: if (alta_node == ∅) then
4:     gr_list=explode_node_to_graphs()
5:     for (τ ∈ gr_list) do
6:         concrete_task_generation(task);
7:     end for
8: else
9:     add_to_conc_list(τ, Ω)
10: end if
```

Algorithm 1 takes as input one task specification, and selects one of its alternative nodes. Further, it explodes the nodes to several copies of the same graph by excluding all other alternative possibilites and calls back the same algorithm. The recursivity ends when a task specification does not contain any alternative control nodes, (becomes a concrete task).

Definition 3. *Let $\tau_{d'}, \tau_{d''}$ be two concrete tasks of specification task τ*

The order relation $>$ is defined as:

$$\tau_{d'} > \tau_{d''} \Rightarrow \sum_{v \in \tau_{d'}} \mathsf{C}(v) \geq \sum_{v \in \tau_{d''}} \mathsf{C}(v) \tag{2}$$

The order relation $>$ allows to sort concrete task according to their total execution time, thus allows prefering concrete tasks with less overhead. Further we will define relation order $>>$ to compare two concrete tasks based on the available engine number.

Algorithm 2 generate_task_separation

```
1: input: τ: Task                          ▷ specification or concrete
2: for (tag ∈ tags) do
3:     for v ∈ V) do
4:         if (v.Tag == tag) then
5:             add_to_current_separation(v)
6:         else
7:             add_to_current_separation(∅)
8:         end if
9:     end for
10: end for
```

Order relations $>$, $>>$ will be used later later in Section 1.6 to select the concrete task to allocate for a given task specification when several concrete tasks can feasibility be allocated.

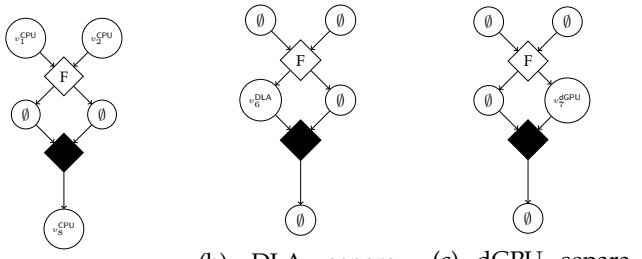
1.2 Graph separation

One concrete task may contain sub-tasks with different tags. Two sub-tasks with different tags are allocated onto different cores. We recall that in this paper we consider partitioned scheduling, analysis of each engine is independant. Hence, we need to *separe* vertices with the same tag before proceeding to its allocation. We call this operation separation.

Definition 4. *Let $\tau_d = (\mathcal{V}, \mathsf{E})$ denote a concrete task. The set $\mathcal{S}(\tau_d) = \{\tau_{d_1}^{tag^1}, \tau_{d_2}^{tag^2}, \cdots, \tau_{d_S}^{tag^S}\}$ is a separation of concrete task τ_d if and only if:*

- *Every separation $\tau_{d_s}^{tag^s}$ is an isomorphe graph regarding τ_d*
- *The number of tags of each separation is equal to **one***
- *The union of the same vertex in all separations is equal to the original vertex. Thus, each vertex is either empty or is equal to the original vertex.*

Each concrete task generates at least as separations as tags in the target architecture. Algorithm 2 shows how the minimum number of task separations is generated. The algorithm acts as a filter on each tag. The vertices having a different tag regarding the current one are set to empty. An empty vertex is a vertex having an execution time that equals $\mathsf{C}(\emptyset) = 0$, otherwise the vertex is added to the current task separation.



(a) CPU separation (b) DLA separation (c) dGPU separation

Fig. 1: minimal set of sepecification for concrete task at Figure

Figure 1 shows the generated task separations using Algorithm 2. As the concrete task has three tags, three task separations are generated. The first one contains only vertices having CPU tag, respectivly the second the DLA and the thrid one the GPU. Each of the task separations will be allocated on one or multiple engines having the same tag.

Definition 5. *Let $\tau_{d'}, \tau_{d''}$ be two concrete tasks of specification task τ. Let* t_l *denote the list of ordered tag in an non-decreasing order of number of cores.*

The order relation $>>$ is defined as:

$$\tau_{d'} >> \tau_{d''} \Rightarrow \sum_{v \in \tau_{d'}(t\_l(0))} \mathsf{C}(v) > \sum_{v \in \tau_{d''} t\_l(0)} \mathsf{C}(v) \quad (3)$$

1.3 Artifial deadlines and offsets

The respect of a real-time task constraints depends on the task allocation which is defined as the problem of allocating sub-tasks onto different engines. As these sub-tasks share buffers, they are forced to respect an execution order. Analyzing the behavior of dependant sub-tasks is complex. To reduce allocation complexity, lot of works have proposed to insert artificial offsets and deadlines to sub-tasks so they can be analyzed separately with respect to their initial execution order. We use similar techniques because the analysis is more complex in the presence of highly heterogeneous engines such as those we are interested in. Deadline assignement techniques are applied to concrete tasks because applying it to task specification may enforce some sub-task to have tight deadline because the critical path of a concrete path is shorter or equal to the critical path of its task specification.

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 τ such that :

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

.

Definition 6. *Let P_x denote the path x of task τ and Π denotes the set of all paths of task τ.*

The critical path $\mathcal{P}$ of task τ is define as:

$$\mathcal{P} = P_x \in \Pi, \max_x \{ \sum_{v \in P_x} \mathsf{C}(v) \} \quad (4)$$

Informally, the critical path is defined as the path having the highest execution time. Let assume $\mathcal{P} = \{v_1, v_2, \cdots, v_l\}$ denote the critical path of concrete task τ_d. We define slack time $\mathsf{S}_{\mathcal{P}}$ of task τ_d as the following:

$$\mathsf{S}(\mathcal{P}) = \mathcal{D}(\tau_d) - \sum_{v \in \mathcal{P}} \mathsf{C}(v) \quad (5)$$

Further, to assign artificial deadline, $\mathsf{S}(\mathcal{P})$ is distributed on different vertices as following:

$$\mathsf{D}(v) = \mathsf{C}(v) + \mathsf{calculate\_share}(\mathsf{v}, \mathcal{P}) \quad (6)$$

The calculate share function computes the slack for every single vertex. This slack can be shared according to two heuristics:

- **Fair distribution:** assigns vertex slack as the ratio of the original slack $\mathsf{s}(\mathcal{P})$ by the number of vertices in the path ($|\mathcal{P}|$) (Equation ((7)))

$$\mathsf{calculate\_share}(v, \mathcal{P}) = \frac{\mathsf{S}(\mathcal{P})}{|\mathcal{P}|} \quad (7)$$

- **Proportional distribution:** assigns single slack according to the contribution of the vertex execution time in the path length, the the share is computed as in Equation (8).

$$\mathsf{calculate\_share}(v, \mathcal{P}) = \frac{\mathsf{C}(v)}{\mathsf{C}(\mathcal{P})} \cdot \mathsf{S}(\mathcal{P}) \quad (8)$$

Recurrently, other paths are evaluated according to an non-decreasing order of their vertice execution time. Already assigned deadlines are not assigned again and their respective slack is substracted from the original slack of each path $P_x \in \Pi$.

Further offsets of each vertex are computed as:

$$\mathsf{O}(v) = \begin{cases} 0, & \mathsf{preds(v)} = \emptyset \\ \max\{\mathsf{O}(v_{\mathsf{pr}}) + \mathsf{C}(v_{\mathsf{pr}})\}, & \mathsf{pr} \in \mathsf{preds(v)} \text{ Otherwise} \end{cases} \quad (9)$$

Definition 7. *Vertex $v \in \mathcal{V}_\tau$ is feasible, if each arrival j finishes its execution within the interval bounded by its arrival time $a(v) = j \cdot \mathsf{T}(\tau) + \mathsf{O}(v)$ and its deadline $a(v) + \mathsf{D}(v)$.*

Theorem 1. *A concrete task (resp. task separation) is feasible, if all vertices within the task are feasible.*

Proof. It is easy to proof that exit-sub tasks have an absolute deadline not greater than the task absolute deadline, thus if exit sub-tasks are feasible (by assumption), thus task is feasible. □

According to Theorem 1, the analysis of every sub-task can be done independently from each other, if they all respect their arrival time and deadlines.

Theorem 2. *Let τ_d be a concrete digraph and $\mathcal{S}(\tau_d)$ be the set of its task separations.*

If $\forall \tau_{d_s} \in \mathcal{S}(\tau_d)$ is feasible, τ is feasible

Proof. First, we would like to recall that vertices having an empty vertex as predecessor are forced to take its deadline and offset into account in their deadline/offset computation (as shown in Section 1.3.

If all task separation are feasible, it means that even with their blocking time due to an empty vertex, the exit nodes finish before the concrete task relative deadline $\mathcal{D}(i)$. Thus, the original concrete task is feasible. □

1.4 Allocation of task specifications

1.5 Single core analyis

1.6 Schedulability analysis and on-line conditional graphs

Algorithm 3 Full algorithm

for $\tau_i \in \mathcal{T}$ **do**
  Ω_i=generate_concrete_task(τ_i)
  for $(\tau_{id} \in \Omega_i)$ **do**
    assign_deadlines_offset(conc_list)
    $\mathcal{S}(\tau_{id})$=generate_task_separation(Ω_i)
    if (**then**feasible_sequential($\mathcal{S}(\tau_{id})$))
      add_to_feasible_list(f_list)
    end if
  end for
  if (|f_list| > 0) **then**
    sort(f_list) ▷ Sort f_list using to "$>$" or "$>>$"
    Allocate f_list(0) to selected_engines
  else
    sort(Ω_i) ▷ sort according to order relations
    $(\tau_{id'}, \tau_{id''}) = paralleliz\_concrete(\Omega_i(0))$
    if (**then**$\tau_{id'} \neq 0$)
      allocate
        $toselectedcores$ **else**
        return FAIL
      end if