Template schedule construction for global real-time scheduling on unrelated multiprocessor platforms

Antoine Bertout*, Joël Goossens†, Emmanuel Grolleau‡, and Xavier Poczekajlo†
*LIAS, Université de Poitiers, ISAE-ENSMA, Poitiers, France, antoine.bertout@univ-poitiers.fr
‡LIAS, ISAE-ENSMA, Université de Poitiers, Chasseneuil-Futuroscope, France, grolleau@ensma.fr
†Université libre de Bruxelles, Brussels, Belgium, firstname.lastname@ulb.ac.be

***Abstract*—The seminal work on the global real-time scheduling of periodic tasks on unrelated multiprocessor platforms is based on a two-step method. First, the workload of each task is distributed over the processors and it is proved that this first step success ensures the existence of a feasible schedule. Then, using this workload assignment as an input, a template schedule construction method is presented. In this work, we review the seminal work and show by using a counter-example that this second step is incomplete. Thus, we propose and prove correct a novel and efficient algorithm to build the template schedule.**

***Index Terms*—real-time scheduling, unrelated multiprocessor platforms, heterogeneous multiprocessors, graph theory**

I. Introduction

In the two last decades, the chips founder market has been very active in developing heterogeneous multiprocessor system-on-chip platforms (MPSoCs). These heterogeneous MPSoCs are widely used in everyday embedded systems, from the smartphones to infotainment processors in the automotive domain. In the academia, multiprocessor systems are generally classified into three categories [1], [2]. (1) *Identical*: all the processors are identical (same speed) and execute the tasks at the same execution rate; (2) *Uniform*: each processor is characterised by a speed relative to a standard speed, e.g., a processor of speed 2 executes any task at twice the rate of a processor of speed 1; (3) *Unrelated*: the rate of execution of a task depends on both the processor and the task. When tasks are to be executed on the same processor, then the schedule is said to be *partitioned*, while if they are allowed to migrate, the schedule is said to be *global*. Real-time scheduling consists in finding a scheduling strategy that will allow every task, which is periodically releasing a job to process, to be completed by its deadline. A schedule is said to be *feasible* if each job released by each task meets its *deadline*. A constructive polynomial complexity method has been proposed to address feasibility in the context of unrelated multiprocessor platforms in [3]. It is constructive because it builds a feasible template schedule if such schedule can exist. The schedule is obtained in two steps: a workload assignment —allocating each task to one or several processor portions—, then a template schedule construction from this assignment.

The research is done in the context of the SOFIST project, supported by Project ARC (Concerted Research Action) of Federation Wallonie-Bruxelles. This research is also supported by the European Unions Horizon 2020 research and innovation program under grant agreement N. 826610.

In this work, we present a counter-example showing that the second step is incomplete, in the seminal paper [4] in the context of processor affinities. Also, we propose a new and efficient method to construct the template schedule and thoroughly prove its correctness. Although the overall soundness of these works regarding the feasibility problem, these flaws in the template schedule construction compromise its applicability. This would prevent correct and concrete implementations generalising the template schedule, relying on techniques such as the deadline partitioning [5].

A. Related work

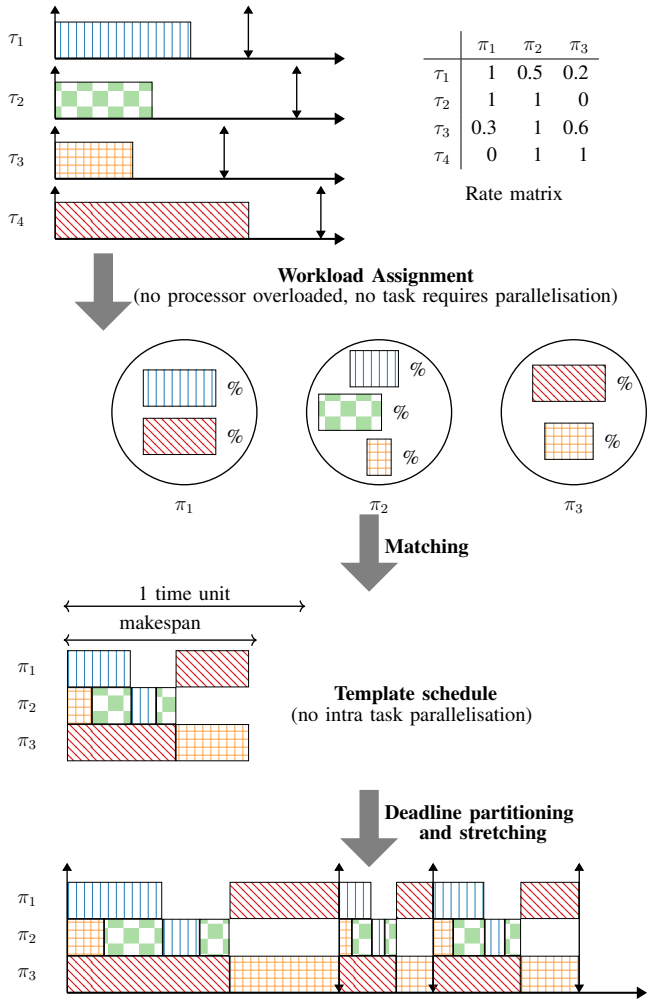
In operational research, a pioneer work on the scheduling of jobs on unrelated multiprocessor platforms was proposed in [6]. In the real-time community, the scheduling on multiprocessor systems has been the subject of intensive studies [1], [2]. In the case of unrelated multiprocessor platforms, apart from [3] revisited in this paper, most studies have been focusing on the partitioned scheduling problem. These studies are mainly proposing heuristics addressing both feasibility and quality of service, as underlined in the survey [7]. Global scheduling has been recently studied in [8], where the proposed method can be applied to two types of processors only. In this specific context, the template schedule is built using Mc Naughton wrap-around rule [9]. More recently, a heuristic to schedule periodic tasks on unrelated multiprocessor platforms has been proposed in [10], with in general, an exponential complexity.

B. This work

This paper summarises in Section II the process proposed in [3], [4], illustrated by a simple guideline example. Then, in Section III, we focus on the template schedule building step. We exhibit a counter-example showing that the seminal method, fails to construct a correct template schedule. We propose an efficient algorithm to fix this problem and prove its correctness in Section IV. Finally, we conclude in Section V.

II. Seminal work

In this section, we briefly review the seminal work from [3] and its context. A two-step method is proposed in that work to build an offline pattern of schedule. This pattern, called template schedule, may be repeated over time to constitute a feasible real-time schedule. Each step is depicted top-down



	π_1	π_2	π_3
τ_1	1	0.5	0.2
τ_2	1	1	0
τ_3	0.3	1	0.6
τ_4	0	1	1

Fig. 1: Building a schedule step by step

in the Figure 1. A first step called workload assignment, is obtained by solving a Linear Programming (LP) problem from the input tasks and platform models. Solving the LP problem computes optimally the processor ratios assigned to tasks in polynomial time, forming an assignment matrix. Note that this step success ensures the system feasibility. The second step rests on the workload assignment matrix to build a template schedule. This step is called *matching* in Figure 1, because it corresponds to iteratively solving matching problems in a bipartite graph, as detailed in Section III. As the seminal work is focused on the feasibility, repeating the template schedule every unit of time is theoretically acceptable. Papers [8], [10] also rely on the notion of the template schedule, although it is differently built. Nevertheless, the authors propose a more applicable solution in stretching the template schedule into consecutive intervals of time delimited by absolute deadlines (referred to as deadline partitioning [5]), in a third step. Using the stretching method strengthens the need to correct the seminal method. After introducing the model, we first define the workload assignment (see Section II-B) and then briefly explain the template construction (see Section II-C), with an emphasis on the matching phase where the correction is later applied (see Section III). An example will be given as a guideline throughout the different sections.

A. Seminal model

a) Task model: The workload is modelled by a set of n *periodic tasks* $\Gamma \doteq \{\tau_1, \tau_2, \ldots, \tau_n\}$ (symbol $\doteq$ means *is equal by definition to*). Each *task* is defined by two parameters (C_i, T_i) where C_i is the *worst-case execution time* on a *fictional processor* —chosen arbitrarily— for every task, and T_i is the release *period*. Each task releases a *job* every period T_i. The first job of a task is released at $t = 0$, the k^{th} at $t = k \times T_i$ and has to complete by $(k+1) \times T_i$ (tasks are said to have implicit deadlines). During its lifespan, a job may *be preempted* or *migrate* from a processor to another, with no time penalty. The processor *utilisation* of a task on an *fictional processor* is defined as $u_i \doteq C_i/T_i$.

Guideline example. We have two tasks τ_1 and τ_2 with following parameters:

τ_i	C_i	T_i	u_i
τ_1	4	2	2
τ_2	3	1	3

b) Platform model: In this work, we consider *unrelated multiprocessor platforms*. An *unrelated platform* $\Pi \doteq \{\pi_1, \pi_2, \ldots, \pi_m\}$ consists of m processors. In this paradigm, the *execution rate* of a task τ_i depends on the processor π_j where it is being executed and is denoted as $r_{i,j}$. A processor π_j executing a job of τ_i for t time units will process $r_{i,j} \times t$ units of its execution time. If $r_{i,j} = 0$, then τ_i cannot be executed on π_j, this couple task/processor is incompatible.

Guideline example. We have three processors and define the following rates $r_{i,j}$ for the task set:

τ_i	$r_{i,1}$	$r_{i,2}$	$r_{i,3}$
τ_1	1	3	0
τ_2	0	5	1

We observe that τ_1 cannot be executed on π_3 and is executed three times faster on π_2 than on π_1.

B. Seminal workload assignment: LP-Feas

The scheduling algorithm from [3] is divided into several steps. Its first step is to split and distribute the execution time of each task over the processors. To do so, the LP problem *LP-Feas* is defined. *LP-Feas* splits each task utilisation into one or several portions and assigns them to the processors, with respect to their capacities.

Formally, the LP problem *LP-Feas*(Γ, Π) is the following:

$$\text{Minimise } \ell, \text{ where:} \tag{1}$$

$$\sum_{j=1}^{m} x_{i,j} \times r_{i,j} = u_i \qquad i = 1, 2, \ldots, n \tag{2}$$

$$\sum_{j=1}^{m} x_{i,j} \leq \ell \qquad i = 1, 2, \ldots, n \tag{3}$$

$$\sum_{i=1}^{n} x_{i,j} \leq \ell \qquad j = 1, 2, \ldots, m \tag{4}$$

The LP problem must ensure that each task is fully executed. The value $x_{i,j}$ denotes the ratio of π_j to be assigned to τ_i. In other words, it represents the required amount of execution time of τ_i on processor π_j. Therefore, the sum of every utilisation portion of a given task times the speed of the assigned processors must be equal to the task utilisation (Equation (2)). The sum of the $x_{i,j}$ represents the total portion of each time unit where τ_i will be executed per time unit. ℓ represents the portion of a time unit where processors are continuously busy and then Equation (3) avoids the parallel execution of tasks. It is trivial that ℓ must be lower or equal to 1 to avoid deadline misses. The last constraint of Equation (4) ensures that processors cannot execute more task workload than their capacity.

Theorem 1 from [3] introduces a major result on the feasibility of a task set on an heterogeneous unrelated platform.

Theorem 1 (from [3])**.** *The LP problem LP-Feas*(Γ, Π) *has a solution with* $\ell \leq 1$ *if and only if the task set* Γ *is feasible on the platform* Π*.*

The value ℓ is called the *makespan* and the objective function of *LP-Feas* (Equation (1)) aims at minimising it. All the processors will be idle in $[\ell, 1)$.

Guideline example. Using our example task set and platform, the following given assignment respects *LP-Feas* constraints:

- $x_{1,1} = x_{1,2} = 0.5$, $x_{1,3} = 0$.
- $x_{2,2} = x_{2,3} = 0.5$, $x_{2,1} = 0$ and $\ell = 1$.

Therefore, the processor π_2 will be equally shared on both tasks.

C. Seminal template schedule construction

It is shown in [3] that a feasible solution of *LP-Feas* (with $\ell \leq 1$) ensures the possible construction of a template schedule on one unit of time.

In the following, we use the notion of *full processors* at time t to denote the processors that will be busy in the interval $[0, t)$, with $0 \leq t \leq \ell$. Symmetrically, the *urgent tasks* at time t are the tasks that must be executed continuously in $[0, t)$. Formally, a processor π_j is *full* at t iff $\sum_{i=1}^{n} x_{i,j} = t$. A task τ_i is *urgent* at t iff $\sum_{j=1}^{m} x_{i,j} = t$. Using these two notions, the template schedule is built iteratively in reverse. It is important to note that t is the time relative to the template schedule which starts at $t = \ell$ and ends at $t = 0$. Also, a task (resp. processor) may become urgent (resp. full) at a given time. By definition, there is at least one urgent task and/or one full processor at $t = \ell$, but there is none for $t > \ell$. At each iteration made at time t, the algorithm assigns some tasks (including all the urgent tasks) to some processors (including all the full processors) for a duration δ in the newly formed interval $[t - \delta, t)$ in the template schedule. This duration is chosen such that no unassigned task or unassigned processor can become urgent within this interval, otherwise the schedule would not be feasible. The set of assignments is called a *matching*. The resulting matching is composed of pairs (τ_i, π_j), representing the assignment of the task τ_i on the processor π_j in the interval $[t - \delta, t)$. The computation of the matching proposed in [3], [4] is detailed in Section III. Once the matching is found and δ is computed, time t is decreased by δ and the workload assignment matrix is updated: for every (τ_i, π_j) in the matching, the corresponding $x_{i,j}$ is decreased by δ. Intuitively, δ is the largest value that respects the following constraints: a) no task τ_i must be assigned on π_j for more than $x_{i,j}$; b) no unassigned task becomes urgent in the interval $[t - \delta, t)$; c) no unassigned processor becomes full in the interval $[t - \delta, t)$; d) and obviously, by construction $t - \delta \geq 0$.

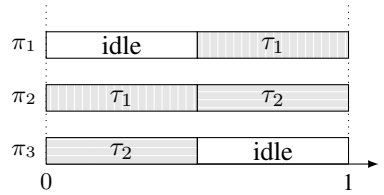


Fig. 2: Resulting template schedule for the guideline example

Iterations are performed starting from $t = \ell$ until $t = 0$, with a new matching and a new δ computed at each iteration. By construction, at time $t = 0$, all the tasks have been fully assigned to processors, with no intra-parallelism.

Guideline example. Be aware that in this example we do not apply the matching algorithm from [3] and [4]. Instead, we apply our algorithm presented and proved correct in Section IV. We do so because the seminal algorithm is flawed, as shown in Section III. No detail are given here on the matching algorithm to focus on the template schedule construction.

The initial workload assignment $n \times m$ matrix X, based on the previously computed x_{ij} values, is used to start the construction of the template schedule at time $t = \ell = 1$: $X_{t=\ell=1} = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \end{bmatrix}$.

At time $t = 1$, $\sum_{j=1}^{m} x_{1,j} = 0.5 + 0.5 = 1$. Therefore, τ_1 is urgent. The same applies to τ_2. Also, we can see that π_2 is full. In order to respect the constraints a) to d), τ_1, τ_2 must be assigned to a processor and π_2 must have a task assigned. The matching algorithm (from Section IV) matches τ_1 on π_1 and τ_2 on π_2. Once the matching has been computed, we define δ. Since $x_{1,1} = x_{2,2} = 0.5$, it may be at most 0.5. Also, π_3 becomes full at $t = 0.5$. For those reasons, $\delta = 0.5$. Knowing δ, we now update the workload assignment matrix by subtracting 0.5 from both $x_{1,1}$ and $x_{2,2}$. The updated matrix is $X_{t=0.5} = \begin{bmatrix} 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$.

At $t = 0.5$, both tasks remain urgent and the processor π_2 is still full. The processor π_3 becomes full. Here, the only possibility is to match τ_1 on π_2 and τ_2 on π_3, for a duration of 0.5. The updated workload assignment matrix is null, therefore the construction is finished. The resulting template schedule is given on Figure 2.

In the following section, we investigate why the seminal matching algorithm may fail to perform a correct matching.

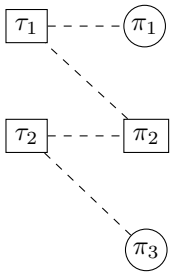
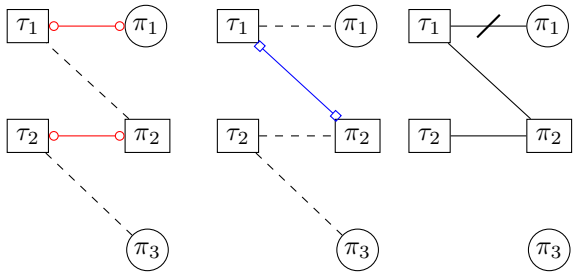


Fig. 3: Bipartite graph corresponding to the workload assignment matrix at time $t = 1$.



(a) Matching M_τ (b) Matching M_π (c) $M_\tau \cup M_\pi$

Fig. 4: Application of the algorithm in [3], [4]

III. Issue with the seminal matching algorithm

In this section, we present the matching algorithm from [3] and [4] which is based on graph theory. This algorithm takes place at each iteration of the template construction, described in Section II-C. The goal is to find an assignment at a given time between the tasks and processors. By definition, each full processor must be assigned to a task, and each urgent task must be assigned to a processor. Also, non-urgent tasks (resp. non-full processors) can be assigned as well, to full processors (resp. to urgent tasks). To compute this assignment, the problem is represented as a bipartite graph $G = (\Gamma \cup \Pi, E \subseteq \Gamma \times \Pi)$, where the partition of vertices Γ (resp. Π) corresponds to the tasks (resp. processors). There exists an edge between vertices τ_i and π_j if, and only if, τ_i may be assigned to π_j. Formally, $(\tau_i, \pi_j) \in E$ iff $x_{i,j} > 0$. Such bipartite graph for our guideline example at time $t = 1$ is depicted on Figure 3. In this figure, edges are represented by dashed lines, urgent tasks and full processors symbolised by square nodes and, non-urgent and non-full processors symbolised by circle nodes. In the following, we construct an assignment through a matching in the bipartite graph. In this respect, we first recall usual definitions of graph theory that will be used in the remainder of the paper.

The *degree* of a vertex v in a graph is the number of edges that are connected to v. A finite *walk* is a sequence of edges $(e_1, e_2, \ldots, e_{n-1})$ which connects a sequence of vertices $(v_1, v_2, \ldots, v_n)$ such that $e_i = (v_i, v_{i+1})$ for $i = 1, 2, \ldots, n-1$. A *trail* is a walk where no edge is repeated. A *path* is a trail without repeated vertices. A graph *traversal* is the process of visiting each vertex of a graph. A *connected component* is a sub-graph in which any two vertices are connected to each other by paths. A *matching* in a bipartite graph G is a subset of its edges $M \subseteq E$ without common vertices. A vertex v is said to be *saturated* (or matched) by M iff an edge in M connects v.

Paper [3] states that a matching saturating every urgent task and every full processor can always be found. We summarised this result in Theorem 2.

Theorem 2 (Fact 2 combined with Theorem 2 from [3])**.** *Giving a bipartite graph built from a workload assignment matrix such that the makespan is not greater than one, i) it is always possible to find a matching that saturates all the urgent tasks and ii) it is always possible to find a matching that saturates all the full processors.*

Using the Theorem 2, the idea is to combine both matchings in order to obtain a matching that saturates both the urgent tasks and the full processors. We now describe the overall matching algorithm proposed in [3], [4]. $\Gamma_u \subseteq \Gamma$ denotes the set of urgent tasks and $\Pi_f \subseteq \Pi$ the set of full processors. The whole matching algorithm is divided in four steps. The first three steps are from [3] and the last step has been added in [4].

1) Determine a matching M_τ from all vertices in Γ_u to a subset of the vertices in Π — by Theorem 2, this can always be done.
2) Determine a matching M_π from all vertices in Π_f to a subset of the vertices in Γ — by Theorem 2, this can always be done.
3) If an urgent task-vertex (i.e., one in Γ_u) appears in this second matching as well, then discard the edge that it was matched to in the initial Γ_u-to-Π matching.
4) (From [4]) If a full processor remains matched with two tasks with these remaining edges, then discard the edge matching it with a non-urgent task, and retain only the edge that matches it with an urgent task

The authors propose the following claim on that algorithm:

Claim 1 (Fact 3 from [3] corresponding to Fact 2 from [4])**.** *What remains after application of the algorithm is a matching that satisfies the following properties: each full processor is matched, and each urgent task is matched. (There may be additional matched vertices, corresponding to non-full processors and non-urgent tasks, as well.)*

This algorithm is applied on the guideline example at time $t = 1$ represented on Figure 4). We observe that the two first steps of the algorithm are successfully achieved. The matching M_τ (edges with circle tips in Figure 4(a)) saturates all the urgent tasks, while the matching M_π (edges with square tips in Figure 4(b)) saturates all the full processors. We now apply step (3) of the algorithm because τ_1 is an urgent task appearing both in M_τ and M_π. Consequently, the edge (τ_1, π_1) is discarded from M_τ (the edge is stroke out in Figure 4(c)). The step (4) proposed in [4] is not applicable since we have a full processor matched with two urgent tasks. As a result,

π_2 is both paired with τ_1 and τ_2, meaning that it is supposed to be allocated both to τ_1 and τ_2. Consequently, we do not obtain a matching, which contradicts Claim 1.

We have shown that both cleaning phases (steps (3) and (4)) are incomplete and may lead to unfeasible schedules. Therefore, we propose in the following section a new algorithm to perform a correct matching.

IV. Corrected matching algorithm

A. Problem statement

To simplify and to make our solution more generic, we rely on the following definition.

Definition 1. *An* important *vertex is either a vertex corresponding to an urgent task or to a full processor.*

Thus, the problem can be formulated as:

Given a bipartite graph having in each partition a subset of ***important*** *vertices. Determine a matching saturating all important vertices. (There may be additional non-important vertices in the matching, as well.)*

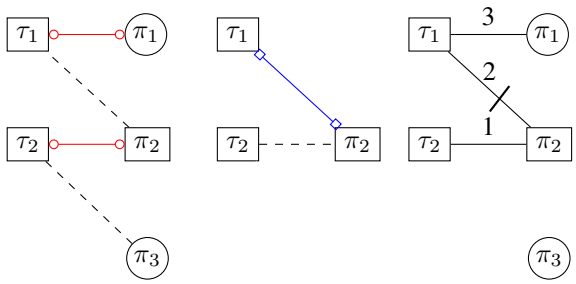
As far as we know, this problem is not addressed in the "traditional" literature of graph theory [11], [12]. The closest problem we found is the assignment problem with seniority and job priority constraints [13], [14] but these works are not straightforwardly applicable to the problem raised here. Therefore, we propose in the following a correction of the procedure introduced in [3].

B. Proposed solution

In our solution, we keep the two first steps proposed in [3], [4], but add as a third step a novel and efficient cleaning phase of the graph resulting from the union of the two matchings. Starting from a graph $G = (\Gamma \cup \Pi, E \subseteq \Gamma \times \Pi)$ obtained from the workload assignment matrix as illustrated in Figure 3, we apply the three following steps:

1) Determine a matching M_τ saturating the urgent tasks — by Theorem 2, this can always be done.
2) Determine a matching M_π saturating the full processors — by Theorem 2, this can always be done.
3) Let $G' = (\Gamma \cup \Pi, \{M_\tau \cup M_\pi\} \subseteq E)$. For each connected component g of G': let v_1 in g be an important 1-degree vertex if it exists or any 2-degree vertex otherwise. Traverse g from v_1 and discard every visited edge $e_i = (v_i, v_{i+1})$ having an even edge index. The first visited edge is $e_1 = (v_1, v_2)$, the second one is $e_2 = (v_2, v_3)$, etc.

Steps (1) and (2) are polynomially solvable using a *maximum cardinality matching* algorithm, as the one in [15]. It is applied to both sub-graph of G: $G_\tau = (\Gamma_u \cup \Pi, E_\tau = E \cap (\Gamma_u \times \Pi))$ and $G_\pi = (\Gamma \cup \Pi_f, E_\pi = E \cap (\Gamma \times \Pi_f))$. G_τ contains only the urgent task vertices (and all the processors vertices), when G_π contains only the full processor vertices (and all the task vertices). The Figure 5(a) (resp 5(b)) illustrates the matching M_τ on G_τ (resp. M_π on G_π).



(a) Matching M_τ (b) Matching M_π (c) $M_\tau \cup M_\pi$

Fig. 5: Proposed matching algorithm

Theorem 2 ensures that a maximum cardinality matching algorithm will find a matching saturating every urgent task vertex and every full processor vertex, respectively.

Step (3) is illustrated in Figure 5(c): in the guideline example, the edge (τ_1, π_2) with the even label number 2 is discarded, giving a correct matching saturating all the full processors and all the urgent tasks. This step simply ensures that i) every important vertex, which was paired in one of the two matchings in step (1) or (2), remains paired, and ii) the maximum degree of a vertex in G' is 1.

C. Proof of correctness of the algorithm

For the sake of the proof and without loss of generality, we consider G' introduced in Step (3) as a directed bipartite graph where edges in M_τ are oriented from an urgent task to a paired processor and edges in M_π are oriented from a full processor to a task. Notice that the directed version of G' only differ from G' in that the union of M_τ and M_π, as sets of directed edges, retains edges with the same vertices in opposite direction (e.g. (τ_i, π_j) and (π_j, τ_i)).

Property 1. *In the directed version of G' built from the union of matchings $M_\tau \cup M_\pi$, every important vertex has exactly one outgoing edge, while every non-important vertex has no outgoing edge.*

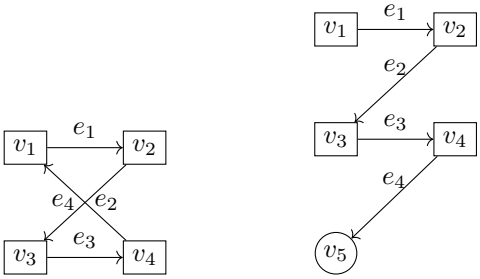
Proof. By definition 1, an important vertex corresponds to either an urgent task or a full processor. By construction of both matching, the outgoing edges are only created from important vertices. The property follows. □

Property 2. *The maximum degree of the directed version of graph G' built from the union of matchings $M_\tau \cup M_\pi$ is 2.*

Proof. Since this graph is obtained by the union of two matchings where each vertex is present once, this graph has vertices with a degree of at most 2. □

Those two properties will be used to prove Theorem 3 in the following.

Theorem 3. *Applying the step (3) of our algorithm to G' ensures a* correct matching *in the resulting graph, i.e. all important vertices have a degree of 1 (therefore every urgent task or full processor is saturated), and the maximum degree of the resulting graph is 1.*



(a) Cycle (b) Path

Fig. 6: Two possible types of connected components

Proof. Without loss of generality, we consider only one connected component of G' in this proof. Step 3 of the algorithm will process every connected component individually. 0-degree vertices are not in the matching and therefore not considered.

By property 2, G' has a maximum degree of 2 and a minimum degree of 1 (as it is a connected component). As G' is connected, there exists a trail connecting all vertices (namely from v_1 to v_n) in G'. Property 1 ensures that this trail vertices $v_1, \ldots, v_{n-1}$ are necessarily important. For the ending vertex v_n, there are only two possibles cases:

Case 1 v_n is important. Thus, it has an outgoing edge and this trail is necessarily a *cycle*, i.e. a trail where no other vertices are repeated but the starting vertex v_1.

Case 2 v_n is non-important. Thus, it has no outgoing edge and this trail is a path.

Both cases are illustrated in Figure 6 and addressed separately hereafter.

Case 1 $v_1 - v_n$ ***cycle:*** It is known that a bipartite graph contains no odd cycles. Consequently, there is a unique cycle in G', $(e_1 = (v_1, v_2), e_2 = (v_2, v_3), \ldots, e_{n-1} = (v_{n-1}, v_n), e_n = (v_n, v_1))$ containing an even number of n edges. It is now clear that discarding every edge with an even index let every vertex with a degree of 1. It results that all vertices are degree 1 vertices, so Case 1 is proved.

Case 2 $v_1 - v_n$ ***path:*** Starting from the important 1-degree vertex v_1, we discard every even indexed edge. The result depends on the path length parity:

- The number of edges is even, with the sequence $(e_1 = (v_1, v_2), e_2 = (v_2, v_3), \ldots, e_{2k-1} = (v_{2k-1}, v_{2k}), e_{2k} = (v_{2k}, v_{2k+1}))$. At the end of Step 3, the last edge is also discarded, then only the last vertex becomes 0-degree, which does not matter because it is a non-important vertex. Every other vertex is of degree 1.
- The number of edges is odd, with the sequence $(e_1 = (v_1, v_2), e_2 = (v_2, v_3), \ldots, e_{2k-2} = (v_{2k-2}, v_{2k-1}), e_{2k-1} = (v_{2k-1}, v_{2k}))$. At the end of Step 3, the last edge of the sequence is preserved and every vertex is of degree 1.

All important vertices are of degree 1 and the non-important vertex has a maximum degree of 1, so Case 2 is also proved and Theorem 3 is demonstrated. □

V. Conclusion

We have revisited the seminal work of [3] proposing a polynomial time constructive feasibility test for global real-time scheduling on unrelated multiprocessor platforms. Using a simple example we have shown that, without compromising the main result on the feasibility, the algorithm building the template schedule was flawed. We have proposed a novel and efficient polynomial time algorithm to correctly build the template schedule, that we thoroughly prove correct. Even if the problem may appear straightforward, the erroneous method was reused in a later work [4] which encouraged us to provide a detailed and illustrated proof. This correction allows the seminal work to be implemented and used to build a feasible offline schedules jointly with the deadline partitioning scheme. Moreover, the result could be used in a more general problem of graph theory with priority vertices that must be saturated.

References

[1] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM computing surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[2] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

[3] S. Baruah, "Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms," in *Real-Time Systems Symposium*. IEEE, 2004, pp. 37–46.

[4] S. Baruah and B. Brandenburg, "Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities," in *Real-Time Systems Symposium*. IEEE, 2013, pp. 160–169.

[5] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *22nd Euromicro Conference on Real-Time Systems*. IEEE, 2010, pp. 3–13.

[6] E. L. Lawler and J. Labetoulle, "On preemptive scheduling of unrelated parallel processors by linear programming," *Journal of the ACM (JACM)*, vol. 25, no. 4, pp. 612–619, 1978.

[7] J. Singh and N. Auluck, "Real time scheduling on heterogeneous multiprocessor systemsa survey," in *4th International Conference on Parallel, Distributed and Grid Computing (PDGC)*. IEEE, 2016, pp. 73–78.

[8] H. S. Chwa, J. Seo, J. Lee, and I. Shin, "Optimal real-time scheduling on two-type heterogeneous multicore platforms," in *Real-Time Systems Symposium*. IEEE, 2015, pp. 119–129.

[9] R. McNaughton, "Scheduling with deadlines and loss functions," *Management Science*, vol. 6, no. 1, pp. 1–12, 1959.

[10] S. Moulik, R. Devaraj, and A. Sarkar, "Hetero-sched: A low-overhead heterogeneous multi-core scheduler for real-time periodic tasks," in *20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2018, pp. 659–666.

[11] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems: Revised Reprint*. SIAM, 2012.

[12] L. Lovász and M. D. Plummer, *Matching theory*. American Mathematical Soc., 2009, vol. 367.

[13] A. Volgenant, "A note on the assignment problem with seniority and job priority constraints," *European journal of operational research*, vol. 154, no. 1, pp. 330–335, 2004.

[14] G. Caron, P. Hansen, and B. Jaumard, "The assignment problem with seniority and job priority constraints," *Operations Research*, vol. 47, no. 3, pp. 449–453, 1999.

[15] J. E. Hopcroft and R. M. Karp, "An n^5/2 algorithm for maximum matchings in bipartite graphs," *SIAM Journal on computing*, vol. 2, no. 4, pp. 225–231, 1973.