Multiprocessor Scheduling of Precedence-constrained Mixed-Critical Jobs

Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga

Verimag Research Report n^o TR-2014-11

November 2014

Reports are downloadable at the following address
http://www-verimag.imag.fr

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Multiprocessor Scheduling of Precedence-constrained Mixed-Critical Jobs

Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga

November 2014

Abstract

The real-time system design targeting multiprocessor platforms leads to two important complications in real-time scheduling. First, to ensure deterministic processing by communicating tasks the scheduling has to consider precedence constraints. The second complication factor is mixed criticality, *i.e.,* integration upon a single platform of various subsystems where some are safety-critical (*e.g.,* car braking system) and the others are not (*e.g.,* car digital radio). Therefore we motivate and study the multiprocessor scheduling problem of a finite set of precedence-related mixed criticality jobs. This problem, to our knowledge, has never been studied if not under very specific assumptions. The main contribution of our work is an algorithm that, given a global fixed-priority assignment for jobs, can modify it in order to improve its schedulability for mixed-criticality setting. Our experiments show an increase of schedulable instances up to a maximum of 30% if compared to classical solutions for this category of scheduling problems.

Keywords: real-time, mixed critical, scheduling, multiprocessor

Reviewers:

How to cite this report:

```
@techreport {TR-2014-11,
  title = {Multiprocessor Scheduling of Precedence-constrained Mixed-Critical Jobs},
  author = {Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga},
  institution = {{Verimag} Research Report},
  number = {TR-2014-11},
  year = {2014}
}
```

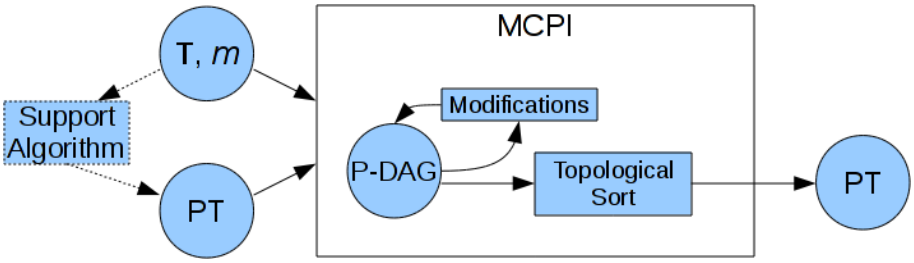


Figure 1: Proposed algortirhm MCPI. $\mathbf{T}$ stands for task graph and PT for priority table.

1 Introduction

The real-time system design targeting multi and many-core platforms leads to two important issues. Firstly, to ensure deterministic processing by communicating tasks one has to consider scheduling problems with precedence constraints, *i.e., task graphs*. Such tasks often have multiple execution rates and hence their jobs have different arrival times and deadlines [1]. However, the precedence constrained scheduling theory for multiple processors usually considers common arrival times and deadlines of connected jobs. Luckily many practical applications are not sporadic but synchronous-periodic, so they can be modeled by a finite task graph that represents one hyperperiod and enables simple static analysis. We abstract from job periodicity and consider just a static set of jobs with arbitrary statically known arrival times, deadlines, and precedence relations.

Modern technology opens the possibility to integrate upon a single chip various subsystems which required multiple chips and boards in the past, which offers power and weight savings. However, this integration leads to the second issue we raise here – the mixed criticality. The point is that some subsystems are safety critical [2]; therefore, according to current industry standards, one cannot let other subsystems share resources with them, to avoid that their errors and faults have consequences for the safety critical subsystems. The current industry practice assumes complete time or space isolation of subsystems having different levels of criticality, which reduces the benefits of integration. It is much more efficient [3] to let the scheduler use the resources in a flexible way during the normal operation, and only when faults occur give all the resources entirely to the safety critical subsystems, to provide them ample means for fault recovery. In addition, one needs to protect highly critical subsystems from timing misbehavior, especially execution time overruns of less critical ones [4]. For static sets of jobs on single processor, the basic principles and results of corresponding scheduling policies were presented in [3], whereas we investigate extensions towards precedence constraints and multiple processors.

For mixed criticality scheduling problems Audsley approach can be used for correct priority assignment [1], but this approach is mainly restricted to uniprocessor scheduling [5]. This is because Audsley approach is based on the assumption that the completion time of the job with the least priority may be computed ignoring the relative priority of the other jobs. This assumption is no longer true in multiprocessors systems. Audsley approach can still be used, by using pessimistic formulas to compute the completion time of the least priority job [5]. However, in the case of finite set of jobs, it can be hard to find a formula with an acceptable level of pessimism. The main contribution of this paper is the *Mixed Criticality Priority Improvement* (MCPI) algorithm, that overcomes the limitation of Audsley approach in multiprocessor system. MCPI, in fact, assigns priorities starting from the highest. This allows us to compute exact completion times. The drawback of this approach is that, unlike Audsley approach, just picking up a job that meets the deadline is not enough for correctness. For this reasons we need an heuristic to help us to select a "good" job in each step. Fig. 1 shows an overview of MCPI. The algorithm takes as input the task graph $\mathbf{T}$, the number of processors m and a priority table PT. The latter may be generated by any known multiprocessor algorithm. We call this algorithm *support algorithm*. The algorithm is based on the concept of *Priority Direct Acyclic Graph* (P-DAG), which defines a partial order on the jobs showing *sufficient* priority constraints needed to obtain a certain schedule. We build such a structure by adding, at each step, jobs from PT, starting from the one with the highest priority. Each time we add a job, we apply a modification to the priority order given by table PT, to increase the schedulability of safety critical scenarios. When

the construction of the P-DAG is terminated, we generate a new priority table by topological sort of the P-DAG.

The paper is organized as follows. Section 2.1 gives an introduction to the formalism of multiprocessor scheduling in Mixed Critical System. Section 3 defines P-DAGs and their properties. The MCPI algorithm is then described in Section 4. In Section 5 we discuss the related work and in Section 6 we give experimental results. Finally in Section 7 we discuss conclusions and future work.

2 Scheduling Problem

2.1 Problem Definition

In a dual-criticality Mixed-Critical System (*MCS*), a *job* J_j is characterized by a 5-tuple $J_j = (j, A_j, D_j, \chi_j, C_j)$, where:

- $j \in \mathbb{N}_+$ is a unique index
- $A_j \in \mathbb{Q}$ is the arrival time, $A_j \geq 0$
- $D_j \in \mathbb{Q}$ is the deadline, $D_j \geq A_j$
- $\chi_j \in \{\text{LO}, \text{HI}\}$ is the job's criticality level
- $C_j \in \mathbb{Q}_+^2$ is a vector $(C_j(\text{LO}), C_j(\text{HI}))$ where $C_j(\chi)$ is the WCET at criticality level χ.

We assume that $C_j(\text{LO}) \leq C_j(\text{HI})$[3]. We also assume that the LO jobs are forced to complete after $C_j(\text{LO})$ time units of execution, so $(\chi_j = \text{LO}) \Rightarrow C_j(\text{LO}) = C_j(\text{HI})$. A *task graph* $\mathbf{T}$ of the MC-scheduling problem is the pair $(\mathbf{J}, \rightarrow)$ of a set $\mathbf{J}$ of K jobs with indexes $1 \ldots K$ and a functional precedence relation $\rightarrow \subset \mathbf{J} \times \mathbf{J}$. The criticality of a precedence constraint $J_a \rightarrow J_b$ is HI if $\chi(a) = \chi(b) = \text{HI}$. It is LO otherwise.

A *scenario* of a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$ is a vector of execution times of all jobs: $(c_1, c_2, \ldots, c_K)$. If at least one c_j exceeds $C_j(\text{HI})$, the scenario is called *erroneous*. The *criticality of scenario* $(c_1, c_2, \ldots, c_K)$ is the least critical χ such that $c_j \leq C_j(\chi), \quad \forall j \in [1, K]$. A scenario is *basic* if for each $j = 1, \ldots, K$ either $c_j = C_j(\text{LO})$ or $c_j = C_j(\text{HI})$.

A (preemptive) schedule $\mathcal{S}$ of a given scenario is a mapping from physical time to $\mathbf{J}_\epsilon \times \mathbf{J}_\epsilon \times \ldots \times \mathbf{J}_\epsilon = \mathbf{J}_\epsilon^m$ where $\mathbf{J}_\epsilon = \mathbf{J} \cup \{\epsilon\}$, where ϵ denotes no job and m the number of processors available. Every job should start at time A_j or later and run for no more than c_j time units. A job may be assigned to only one processor at time t, but we assume that job migration is possible to any processor at any time. Also for each precedence constraint $J_a \rightarrow J_b$, job J_b may not run until J_a completes. A job J is said to be *ready* at time t iff:

1. all its predecessors completed execution before t
2. it is already arrived at time t
3. it is not yet completed at time t

The online state of a run-time scheduler at every time instance consists of the set of completed jobs, the set of *ready jobs*, the progress of ready jobs, *i.e.,* for how much each of them has executed so far, and the current criticality mode, χ_{mode}, initialized as $\chi_{mode} = \text{LO}$ and switched to 'HI' as soon as a HI job exceeds $C_j(\text{LO})$. A schedule is *feasible* if the following conditions are met:

Condition 1. *If all jobs run at most for their LO WCET, then both critical (HI) and non-critical (LO) jobs must complete before their deadline, respecting all precedence constraints.*

Condition 2. *If at least one job runs for more then its LO WCET, than all critical (HI) jobs must complete before their deadline, whereas non-critical (LO) jobs may be even dropped. Also LO precedence constraints may be ignored.*

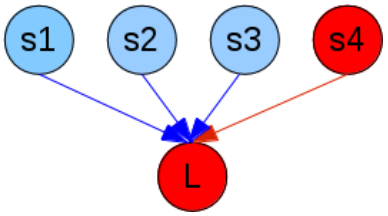


Figure 2: The graph of an airplane localization system illustrating LO→HI dependencies.

The reason why we allow to have precedences from LO jobs to HI jobs can be seen in the example of Fig. 2. There we have a task graph of the localization system of an airplane, composed of four sensors (jobs s1-s4) and the job L, that computes the position. Data coming from sensor s4 is necessary and sufficient to compute the plane position with a safe precision, thus only s4 and L are marked as HI critical. On the other hand, data from s1, s2 and s3 may improve the precision of the computed position, thus granting the possibility of saving fuel by a better computation of the plane's route. So we do want job L to wait for all the sensors during normal execution, but when the systems switch to HI mode we only wait for data coming from s4.

Based on the online state, a *scheduling policy* deterministically decides which ready jobs are scheduled at every time instant on m processors. A scheduling policy is *correct* for the given task graph $\mathbf{T}$ if for each non-erroneous scenario it generates a feasible schedule. A scheduling policy is *predictable*, if an earlier completion of a job may not delay the completion of another job.

A task graph $\mathbf{T}$ is *MC-schedulable* if there exists a correct scheduling policy for it. A *fixed-priority* scheduling policy is a policy that can be defined by a priority table PT, which is a vector specifying all jobs in a certain order. The position of a job in PT is its *priority*, the earlier a job is to occur in PT the higher the priority it has. Among all ready jobs, the fixed-priority scheduling policy always selects the m highest-priority jobs in PT. A priority table PT defines a total ordering relationship between the jobs. If job J_1 has higher priority than job J_2 in table PT, we write $J_1 \succ_{PT} J_2$ or simply $J_1 \succ J_2$, if PT is clear from the context. In this paper we assume *global* fixed-priority scheduling which allows unrestricted job migration. A priority table PT is required to be *precedence compliant i.e.,* the following property should hold:

$$J \rightarrow J' \Rightarrow J \succ_{PT} J' \tag{1}$$

The above requirement is reasonable, since we may not schedule a job before its predecessors complete. The use of fixed-priority in combination with the adopted precedence aware definition of ready job is called in literature *List Scheduling*.

We combine list scheduling with *fixed priority per mode* (FPM), a policy with two tables: PT_{LO} and PT_{HI}. The former includes all jobs. The latter only HI jobs. As long as the current mode is LO, this policy performs the fixed priority scheduling according to PT_{LO}. After a switch to the HI mode, this policy drops all pending LO jobs and applies priority table PT_{HI}. Since scheduling after the mode switch is a single-criticality problem, such a table can be obtained by using classical approaches. Therefore, we focus on producing the table PT_{LO}, in the following simply denoted as PT.

Fixed-priority (FP) policy (without precedences), is predictable [6], while list scheduling (with precedences) is not, therefore, for the online scheduling we modify this policy to ensure predictability as described in Sec. 4.2. For predictable policies it is sufficient to restrict the offline schedulability check to simulation of basic scenarios [3]. To be more specific [7], firstly, we check the scenario with execution times $c_j = C_j(\text{LO})$, *i.e.,* the LO scenario. Secondly, for each HI job J_h, we check the scenario where the jobs that completed before J_h have $c_j = C_j(\text{LO})$, while the other jobs (including J_h) have $c_j = C_j(\text{HI})$. Such a scenario is denoted HI$[J_h]$. We check these scenarios offline under list scheduling, and then use their start times as arrival times online.

2.2 Characterization of Problem Instance

To characterize the performance of scheduling algorithms one uses the utilization and the related demand-capacity ratio metrics. For a job set $\mathbf{J} = \{J_i\}$ and an assignment of execution times c_i the appropriate metric is load [8]:

$$\ell load(\mathbf{J}, c) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i \in \mathbf{J}:\ t_1 \le A_i \wedge D_i \le t_2} c_i}{t_2 - t_1}$$

For a multiprocessor system there does not exist a necessary and sufficient schedulability bound on load, whereas it exists for *uniprocessor systems*: $\ell load \le 1$. For m-processor system the corresponding bound is only *necessary, but not sufficient* [9]: $\ell load \le m$. In Section 5 we also discuss sufficient conditions on load for fixed priority scheduling.

From the problem instance $\mathbf{T}(\mathbf{J}, \rightarrow)$ it is convenient to derive the following graphs:

HI-criticality graph $\mathbf{T}_{\text{HI}}(\mathbf{J}_{\text{HI}}, \rightarrow_{\text{HI}})$, where the nodes and edges are the subset of HI jobs and precedences of HI criticality level

MIX-criticality graph $\mathbf{T}_{\text{MIX}}(\mathbf{J}_{\text{MIX}}, \rightarrow)$, where the jobs in $\mathbf{J}_{\text{MIX}}$ are obtained from the original set of jobs $\mathbf{J}$ by modifying only job deadlines: $D_{\text{MIX}i} = D_i - (C_i(\text{HI}) - C_i(\text{LO}))$.

For static mixed-criticality jobs, [10] and [11] propose the following characterization of mixed-criticality load:

$$Load_{\text{LO}}(\mathbf{T}) = \ell load(\mathbf{J}, C(LO))$$
$$Load_{\text{HI}}(\mathbf{T}) = \ell load(\mathbf{J}_{\text{HI}}, C(HI))$$
$$Load_{\text{MIX}}(\mathbf{T}) = \ell load(\mathbf{J}_{\text{MIX}}, C(LO))$$

The necessary schedulability condition for load on m identical processors then generalizes to mixed criticality as follows: $Load_{\text{LO}}(\mathbf{T}) \le m \wedge Load_{\text{HI}}(\mathbf{T}) \le m$. However, it was noticed in [11] that in the LO scenario the jobs should meet deadlines $D_{\text{MIX}j}$, otherwise deadlines D_j can be missed in a HI scenario, so they made this condition stronger by replacing $Load_{\text{LO}}$ by $Load_{\text{MIX}}$.

Lemma 2.1 (Necessary condition for schedulability)**.** *Mixed-critical problem instance* $\mathbf{T}$ *is schedulable only if*

$$Load_{MIX}(\mathbf{T}) \le m \wedge Load_{HI}(\mathbf{T}) \le m \tag{2}$$

In MIX-criticality graph $\mathbf{T}_{MIX}$ *we should have for all jobs:*

$$A_i + C_i(LO) \le D_{MIXi}$$

whereas in HI-criticality graph $\mathbf{T}_{HI}$ *we should have:*

$$A_i + C_i(HI) \le D_i$$

For practical reasons, we refine the load to a new metric:

$$stress(\mathbf{J}, c) = \max_{0 \le t_1 < t_2} \frac{m}{\min\{m, |\mathbf{J}'|\}} \cdot \frac{\sum_{\mathbf{J}' = J_i | t_1 \le A_i \wedge D_i \le t_2} c_i}{t_2 - t_1}$$

The $m/|\mathbf{J}'|$ scale factor is used to consider the fact that if there are $j < m$ ready jobs then only j processors can be used to schedule them.

Based on $stress$, one can define $Stress_{\text{LO}}$, $Stress_{\text{HI}}$ and $Stress_{\text{MIX}}$. One can also rewrite the necessary conditions (2) using stress, but that would not make them stronger. Nevertheless in general, we have $stress \ge \ell load$, therefore we use it as a more 'realistic' metric of 'complexity' of the scheduling problem, as for the problem instances of growing complexity it approaches the critical bound m faster than the load.

The formulas of Load and Stress introduced above do not take into account *precedence constraints*. To solve this issue, we define ASAP arrival times and ALAP deadlines, known in the task graph theory [12], but so far mainly used to derive priority tables rather than to compute the load[1].

[1] In literature the word ALAP is usually used for latest arrival

For a task graph with execution times c, ASAP arrival time A^* is the earliest time when a job can possibly start:

$$A_j^* = \max_i \left(A_j, A_i^* + c_i \mid J_i \text{ are predecessors of } J_j\right)$$

Dually, ALAP deadline D^* is the latest time when a job is allowed to complete:

$$D_j^* = \min_i \left(D_j, D_i^* - c_i \mid J_i \text{ are successors of } J_j\right)$$

It is trivial that substituting ASAP arrival time and ALAP deadline to the job parameters does not change the schedulability of the task graph, so the necessary conditions in Lemma 2.1 remain valid, whereas the lemma becomes, in general, stronger. It should be noted that, by definition, to compute $Load_{\text{MIX}}$ one should do the ASAP/ALAP calculation in MIX-criticality graph $\mathbf{T}_{\text{MIX}}$ using $C(LO)$, whereas for $Load_{\text{HI}}$ it should be done in graph $\mathbf{T}_{\text{HI}}$ using $C(HI)$. Therefore ASAP arrival and ALAP deadlines for the same job are *mode-dependent*, and one should use these mode-dependent values also for the second part of Lemma 2.1, where we check the properties of individual jobs. In the sequel, unless mentioned otherwise, we assume in the algorithms and analysis that the load and stress values are computed using ASAP and ALAP values.

3 Priority DAG

In this section we will introduce the idea of Priority DAG (P-DAG). Informally it is a graph that defines a partial order on the jobs showing *sufficient* priority constraints needed to obtain a certain schedule. This structure makes it easier to reason on priorities than a priority table, since the latter is a total order and thus contains also *unnecessary* priority constraints. We will imply for the rest of this section that we are using preemptive list scheduling and we always refer to the basic LO scenario. A priority table PT defines a total order on the set of jobs $\mathbf{J}$ of $\mathbf{T}$. A priority table PT defines one and only one schedule $\mathcal{S}$ when applying list scheduling on m processors, we indicate it with the following notation: $PT \vDash_m \mathcal{S}$.

Consider a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$, a number of processors m and the graph $G = (\mathbf{J}, \triangleright)$, where $\triangleright$ is a partial order relation defined on $\mathbf{J}$.

Definition 1 (P-DAGs and their Equivalence)**.** *We call $\mathbf{PT}(G)$ the set of all priority tables that can be obtained by a topological sort of G. G is a P-DAG on m processors for schedule $\mathcal{S}$ iff:*

$$\forall PT,\ PT \in \mathbf{PT}(G) \Rightarrow PT \vDash_m \mathcal{S} \tag{3}$$

Two P-DAGs giving the same schedule are called equivalent*.*

Definition 2 (Canonical P-DAG)**.** *A* Canonical *P-DAG for a schedule $\mathcal{S}$ is a P-DAG G:*

$$\forall PT,\ PT \in \mathbf{PT}(G) \Leftrightarrow PT \vDash_m \mathcal{S} \tag{4}$$

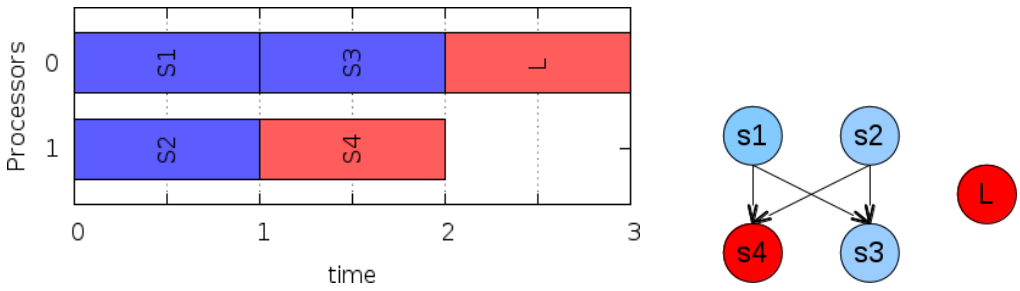
Let $\mathcal{S}$ be the schedule of a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$ produced by a priority table PT on m processors. Given two jobs J_1 and J_2, we say that J_1 *blocks* J_2 ($J_1 \vdash_{\mathcal{S}} J_2$) if in the schedule $\mathcal{S}$ there is a point in time t where J_2 is ready but not running while J_1 is running. It's trivial that:

$$J_1 \vdash_{\mathcal{S}} J_2 \Rightarrow J_1 \succ_{PT} J_2 \tag{5}$$

Lemma 3.1. *Given a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$, a table PT and a number of processors m. Consider the* blocking *relation $\vdash_{\mathcal{S}}$, where $\mathcal{S}$ is such that $PT \vDash_m \mathcal{S}$. Then $G = (\mathbf{J}, \vdash_{\mathcal{S}})$ is a canonical P-DAG for $\mathcal{S}$.*

Proof. We need to prove that (4) holds. Let us first prove that G is actually a P-DAG (*i.e.,* (3) holds). This trivially comes from the observation that during the execution of the schedule $\mathcal{S}$, we only need to define a priority when a job blocks another. So the priorities defined by $\vdash_{\mathcal{S}}$ are *sufficient* to generate $\mathcal{S}$.

To prove that the priorities defined by $\vdash_{\mathcal{S}}$ are also *necessary*, let us suppose by contradiction that there exist a table PT' such that $PT' \vDash_m \mathcal{S}$ and $PT' \notin \mathbf{PT}(G)$. The latter means that $\exists\ J_1, J_2$ so that $J_1 \vdash_{\mathcal{S}} J_2$ and $J_1 \not\succ_{PT'} J_2$. By the first statement and by (5), we have $J_1 \succ_{PT'} J_2$ that contradicts the second statement. □



(a) Schedule (b) P-DAG

Figure 3: The figures of Example 3.1.

Example 3.1. *Let us consider the tasks of Fig 2, where* $\mathbf{J}$ *is defined as follows:*

Job	*A*	*D*	χ	$C(LO)$	$C(HI)$
s1	*0*	*3*	*LO*	*1*	*1*
s2	*0*	*3*	*LO*	*1*	*1*
s3	*0*	*3*	*LO*	*1*	*1*
s4	*0*	*4*	*HI*	*1*	*3*
L	*0*	*6*	*HI*	*1*	*3*

consider the priority table $PT = \{s1 \succ s2 \succ s3 \succ s4 \succ L\}$. *On two processors* PT *produces the schedule* $\mathcal{S}$ *shown in Fig. 3(a). From the figure is easy to derive the blocking relation* $\vdash_{\mathcal{S}}$. *We have:* $s_1 \vdash s_3$, $s_2 \vdash s_3$, $s_1 \vdash s_4$, $s_2 \vdash s_4$. *Notice that L is never blocked, because, due to precedence constraints, it is never ready until time 2, when all its predecessors complete. From the blocking relation* $\vdash_{\mathcal{S}}$, *we can derive the canonical P-DAG* $G = (\mathbf{J}, \vdash_{\mathcal{S}})$, *shown in Fig. 3(b).*

Also, the following is trivial:

Lemma 3.2. *If adding an edge to a P-DAG* G *does not introduce a cycle, the resulting graph* G' *is still a P-DAG and it is equivalent to* G. *Also* $\mathbf{PT}(G') \subseteq \mathbf{PT}(G)$.

Definition 3 (Redundant edges)**.** *An edge* (J_1, J_2) *of a P-DAG* G *is called* redundant *iff there exists another path in* G *from* J_1 *to* J_2.

Removing redundant edges from a P-DAG G will not have any effect on $\mathbf{PT}(G)$. The following trivially follows from Lemmas 3.1 and 3.2:

Lemma 3.3. *Consider a task graph* $\mathbf{T} = (\mathbf{J}, \rightarrow)$ *and a graph* $G = (\mathbf{J}, \triangleright)$. *Let* $\triangleright^*$ *be the transitive closure of* $\triangleright$ *and* $\mathcal{S}$ *be a schedule generated by a priority table* $PT \in \mathbf{PT}(G)$. *Then* G *is a P-DAG iff:*

$$J' \vdash_{\mathcal{S}} J'' \Rightarrow J' \triangleright^* J'', \ \forall J', J'' \in \mathbf{J} \tag{6}$$

We are interested in generating P-DAGs that are shaped like forests (*i.e.,* a set of unconnected trees). The reason why we want such a structure will be clear in Section 4, where we use the properties of forest to prove some properties of our algorithm.

We propose in this section an algorithm that generates a forest-shaped P-DAG. We will first explain the algorithm and then prove its correctness. The algorithm is shown in Fig. 4, it takes a task graph and a precedence compliant priority table as input and proceeds as follows. The highest priority job J^{Curr} is removed from the table PT and added to the graph G. Then we simulate a run on the list scheduler in the basic LO scenario for the jobs included so far in G with their precedence relation taken from $\mathbf{T}$ and using as priority table a topological sort of G. During this simulation we keep note of the jobs that block J^{Curr}. Based on the computed blocking relation, in the next steps we inspect the trees of (forest) graph G, looking for the trees that contain a job that is known to block the job J^{Curr}. For each such tree, its root is connected by a new edge in graph G to the job J^{Curr}.

1: **Algorithm:** $Forest\_PDAG$
2: **Input:** **task graph T**
3: **Input:** **priority table** PT
4: **Output:** **P-DAG** G
5: $G = (\emptyset, \emptyset)$
6: **while** $PT \neq \emptyset$ **do**
7: $J^{Curr} \leftarrow PopHighestPriority(PT)$
8: $G.\mathbf{J} \leftarrow G.\mathbf{J} \cup \{J^{Curr}\}$
9: $\vdash \leftarrow SimulateListSchedule(\mathrm{LO},\ (G.\mathbf{J}, \rightarrow),\ \mathbf{PT}(G))$
10: **for** *all trees* $ST' \in G$ **do**
11: **if** $\exists\ J' \in ST' \colon J' \vdash J^{Curr}$ **then**
12: $G.\rhd \leftarrow G.\rhd \cup \{(root(ST'), J^{Curr})\}$
13: **end if**
14: **end for**
15: **end while**
16: **return** G

Figure 4: The forest P-DAG generation algorithm

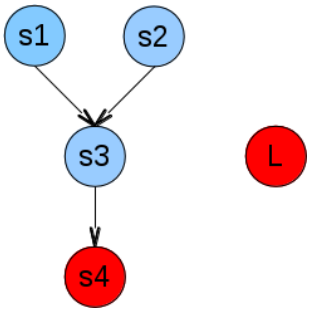


Figure 5: Forest P-DAG

Example 3.2. *Consider the task graph and the priority table of Example 3.1. We will apply* $Forest\_PDAG$ *algorithm to them. In the first step the algorithm picks up* s_1*, the highest priority jobs from* PT*, and will add it to the graph. In the second iteration, we pick up* s_2*, since it is not blocked by any job, we continue without adding any arc. Then we pick up* s_3*, that is blocked by both* s_1 *and* s_2*, so we add the arcs* (s_1, s_3) *and* (s_2, s_3)*. At the next iteration we pick up job* s_4*, that is also blocked by both* s_1 *and* s_2*, so we add an arc from the root of the tree that contains the blocking jobs (*i.e., s_3*) to* s_4*. In the final iteration we pick up job* L*, that is not blocked by any job, thus we add it to the graph without inserting any arcs from it. The resulting graph is shown in Fig. 5.*

Theorem 3.4. *Let* G *be the graph generated by the* Forest_PDAG *algorithm. Then* G *is a P-DAG and a forest.*

Proof. We will prove both by induction, by showing that at the n-th step, the statement is true for the partial graph G_n and for priority table PT_n, where both are composed of the first n elements of PT.

Basic step. The basic step is trivial. We have a priority table $PT_1 = \{J_1\}$ with one element and a graph $G_1 = (\{J_1\}, \emptyset)$. A graph of one element is a forest and the only possible topological sort of G_1 gives PT_1.

Inductive step. We know by inductive hypothesis that G_{n-1} is a P-DAG and can generate PT_{n-1}. Also G_{n-1} is a forest. We only add edges to J_n from the root of unrelated subtrees, this operation may only generate another tree, thus G_n is a forest. Also, since J_n has no successors in G, during a topological sort of G_n we can give J_n the n-th position in the priority table, same position it has in PT_n. For the other jobs, the partial graph that we have to explore is exactly G_{n-1}, so we can generate PT_{n-1} from it.

1: **Algorithm:** $MCPI$
2: **Input:** **task graph** $\mathbf{T}$
3: **Input:** **priority table** SPT
4: **Output:** **priority table** PT
5: $SPT \leftarrow PTTransform(SPT)$
6: $CheckLOscenarioSchedulability(\mathbf{T}, SPT)$
7: $G \leftarrow MCPI\_PDAG(\mathbf{T}, SPT, \emptyset)$
8: $PT \leftarrow TopologicalSort(G)$
9: **if** $anyScenarioFailure(PT, \mathbf{T})$ **then**
10: **return** (FAIL)
11: **end if**

Figure 6: The MCPI algorithm

Since by construction up to the $(n-1)$-th element PT_n and PT_{n-1} are equal, we can generate PT_n by topological sort of G_n.

□

4 Algorithm

We define here the *Mixed Criticality Priority Improvement* (MCPI) algorithm. It is basically an algorithm to compute offline job priorities under list scheduling, while online we use precedence-unaware global fixed priority with adapted arrival times. As previously discussed, our aim is to overcome the limitation of Audsley approach in multiprocessor systems, by assigning priorities starting from the highest. This allows us to compute the exact job completion times. We first discuss the offline computation of priorities and then we describe the online policy.

4.1 Offline priorities computation

As shown in Fig. 1, MCPI takes as input a priority table, produced by a support algorithm. If we use support algorithm *ALGO*, we indicate that with the following notation: MCPI(*ALGO*). A panoramic of the possible support algorithms is given in Section 5.

We use FPM policy, *i.e.,* we have to generate two tables, one for LO mode and one for HI mode. As previously discussed, scheduling in HI mode is a single criticality problem. Thus we will compute PT_{HI} for HI with the support algorithm using $C(\text{HI})$ and graph $\mathbf{T}_{\text{HI}}$. MCPI is just used to compute PT_{LO}, which we will simply denote as PT. To construct such a PT, MCPI takes the priority table generated by the support algorithm and tries to improve the HI scenarios schedulability by increasing the priorities of HI jobs as much as possible without undermining the LO schedulability.

The pseudocode of the algorithm is given in Fig. 6. The algorithm takes as inputs the support priority table SPT and the task graph $\mathbf{T}$. We require SPT to satisfy precedence compliant property (1). In the case where the support algorithm does not imply property (1), we apply a transformation to SPT such that (1) will hold. The transformation is done as follows. We repeatedly scan the priority table, from the highest to the least priority. For each job J that has higher priority than some of its predecessors in $\mathbf{T}$, we raise the priority of those predecessors moving them immediately before J, keeping their relative order. This procedure is illustrated in Fig 7, where we show the task graph, the priority table and its modifications.

We then check LO scenario schedulability. If the schedulability holds, it will be kept as an invariant during the execution. Subroutine $MCPI\_PDAG$ generates a forest P-DAG, based on the support priority table SPT. It is a modified version of $Forest\_PDAG$. Then we obtain a priority table from G by using the well-known $TopologicalSort$ procedure (see *e.g.,* [13]), which traverses the trees in G from the leafs to the roots while adding the visited nodes to PT. Finally, the subroutine $anyScenarioFailure$ checks the schedulability. The check is done by a simulation over the set of all scenarios $HI[J_h]$, as explained in Section 2.1.

In Fig. 8 subroutine *MCPI_PDAG* is shown. It takes as inputs the task graph $\mathbf{T}$, the support priority

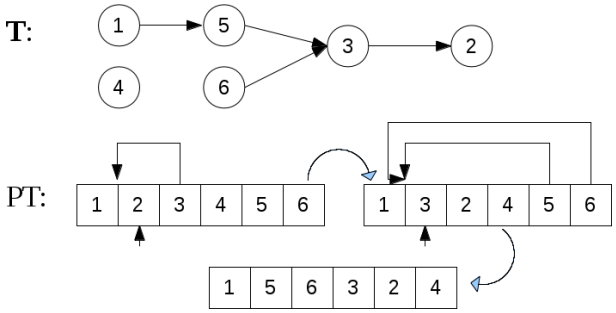


Figure 7: The initial PT transformation

1: **Algorithm:** $MCPI\_PDAG$
2: **Input:** **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
3: **Input:** **priority table** SPT
4: **In/out:** **forest P-DAG** $G(\mathbf{J}', \triangleright)$
5: **if** $\mathbf{T} \neq \emptyset$ **then**
6: $J^{\text{curr}} \leftarrow SelectHighestPriorityJob(\mathbf{T}.\mathbf{J}, SPT)$
7: $\vdash \leftarrow SimulateListSchedule(\text{LO},\ (G.\mathbf{J}', \rightarrow),\ \mathbf{PT}(G) \frown J^{\text{curr}}\ )$
8: $G.\mathbf{J}' \leftarrow G.\mathbf{J}' \cup \{J^{\text{curr}}\}$
9: **for** *all trees* $ST \in G$ **do**
10: **if** $\chi(J^{\text{curr}}) = \text{LO}$ **then**
11: **if** $\exists\ J' \in ST\colon J' \vdash J^{\text{curr}} \vee J' \rightarrow J^{\text{curr}}$ **then**
12: $ConnectAsRoot(ST, J^{\text{curr}})$
13: **end if**
14: **else**
15: $ConnectAsRoot(ST, J^{\text{curr}})$
16: **end if**
17: **end for**
18: **if** $\chi(J^{\text{curr}}) = \text{HI}$ **then** $PullUp\ (J^{\text{curr}}, G, \mathbf{T}, SPT)$
19: $\mathbf{T}.\mathbf{J} \leftarrow \mathbf{T}.\mathbf{J} \setminus \{J^{\text{curr}}\}$
20: $MCPI\_PDAG(\mathbf{T}, SPT, G)$
21: **end if**

Figure 8: The algorithm for computing priority tree in MCPI

Algorithm: $PullUp$
Input: **job** J
In/out: **forest P-DAG** G
Input: **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
Input: **priority table** SPT
$PREC \leftarrow LOpredecessors(J, G)$
while $PREC \neq \emptyset$ **do**
    $J' \leftarrow SelectLeastPriorityJob(PREC, SPT)$
    $PREC \leftarrow PREC \setminus \{J'\}$
    if $CanSwap(J, J', G, \mathbf{T}, SPT)$ **then**
        $PREC \leftarrow PREC \cup LOpredecessors(J', G)$
        $TreeSwap(J, J', G)$
    end if
end while

Figure 9: The pull-up subroutine

Algorithm: $CanSwap$
Input: **HI job** J
Input: **LO job** J'
Input: **forest P-DAG** G
Input: **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
Input: **priority table** SPT
if $J' \rightarrow^* J$ **then**
    return False
end if
$TreeSwap(J, J', G)$
$\mathbf{J}'' \leftarrow G.\mathbf{J}' \cup \mathbf{T}.\mathbf{J}$
$PT'' \leftarrow \mathbf{PT}(G) \frown (SPT \mid\prec J)$
$allDeadlinesMet \leftarrow SimulateListSchedule(\text{LO}, (\mathbf{J}'', \mathbf{T}. \rightarrow), PT'')$
return $allDeadlinesMet$

Figure 10: The subroutine for checking the feasibility of a priority swap

table SPT, and the graph G generated so far (that will be empty at the beginning). This subroutine is very similar to the algorithm of Fig. 4. It selects the highest priority unassigned job of table SPT and adds it to the graph G. Then it evaluates the relation $\vdash$ by simulation. In this simulation we assume that the selected job has the lowest priority and the other priorities are determined by P-DAG G. Notation $PT \frown J$ means concatenation of job J in the lowest-priority (*i.e.,* the last) position in the priority table PT. After that:

if $\chi(J^{\text{curr}}) = \textbf{LO}$ we add an arc to J^{curr} from all the roots of the trees ST present in G where $\exists\, J' \colon J' \vdash J^{\text{curr}}$. We also add an arc to J^{curr} from the root of the subtrees ST present in G where $\exists\, J' \colon J' \rightarrow J^{\text{curr}}$. This makes J^{curr} the new root of ST.

if $\chi(J^{\text{curr}}) = \textbf{HI}$ an arc to J^{curr} from the roots of all the trees present in G is added.

The reason why we add extra arcs, compared to the procedure shown in Fig. 4 is to ensure safety of further modifications of G. These modifications are done by subroutine $PullUp$ when called on J^{curr}. This subroutine is the core of the algorithm. It modifies the P-DAG generated so far trying to improve the HI schedulability of the initial priority order. Notice that if this subroutine were not called, the algorithm would just generate a P-DAG of the initial priority table SPT. After the $PullUp$, we just remove the current job from the working set and the subroutine is called again recursively.

Procedure $PullUp$ is described by the pseudocode in Fig. 9. The idea behind this subroutine is to try to improve the schedulability of HI scenarios by raising the priorities of HI jobs, "swapping" their position in the graph with LO jobs while keeping the LO scenario schedulability an invariant.

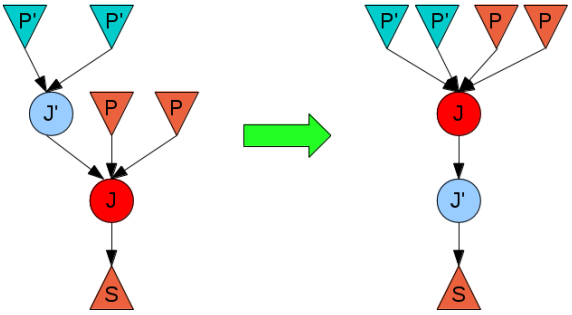


Figure 11: The effect of a Swap. The red triangle marked with S represent the successors of J, while the triangle marked with P and P' are, respectively the predecessors of J and J'.

Procedure $LOpredecessors(J, G)$ returns the set of direct predecessors of LO criticality: $\{J_s \mid J_s \rhd J,\ \chi_s = \text{LO}\}$. At each step in Fig. 9 we pick the least priority predecessor from the working set $PREC$, then subroutine $CanSwap$ checks if J and J' can swap priorities. If so, we perform the swap and extend the working set. The subroutine proceeds until this set is empty. As shown in Figure 10, subroutine $CanSwap$ uses a private copy of graph G to perform a tentative swap modification and then evaluates its impact by reconstructing the whole original job instance as $\mathbf{J}''$. It also constructs a complete PT'' by letting the jobs already included in G to have the most significant priorities, obtained according to G, and the jobs not yet included in G to have the least priorities according to SPT. Note that the latter jobs are identified by the 'trailer' part of SPT that has less priority than the current HI job J, we denote that 'trailer' part as $(SPT \mid\prec J)$. Note that thus we check the whole job set of the problem instance and not only the jobs whose priorities have been changed. This is required on a multi-processor because, unlike in single-processor case, changing the priorities of a pair of jobs may impact the schedulability of not only these jobs but of all jobs that have less priority. We accept the swapping only if it does not lead to a deadline miss for any job. This way, we maintain the schedulability in LO mode as an invariant of the algorithm. Note that $CanSwap$ immediately rejects to swap J and J' if $J' \rightarrow^* J$, to maintain the precedence compliance of priorities.

Procedure $TreeSwap(J, J', G)$ performs the following modification on graph $G(\mathbf{J}', \rhd)$:

1. $J' \rhd J$ is transformed into $J \rhd J'$
2. $\forall J_p\colon J_p \rhd J',\ J_p \rhd J'$ is transformed into $J_p \rhd J$
3. $if\ \exists J_s\colon J \rhd J_s,\ J \rhd J_s$ is transformed into $J' \rhd J_s$

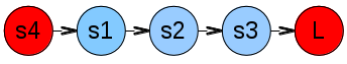
The swap is illustrated in Fig. 11.

When the swap is done, the $PullUp$ subroutine updates the set $PREC$ to take into account point 2 in the $TreeSwap$ definition and reiterates.

Example 4.1. *Consider again the instance and the priority table of Example 3.2. Let us apply MCPI on them. The table PT is already precedence compliant, so $PTTransform$ will not modify it. Then we check LO schedulability, by simulation. The result of the simulation of the LO scenario is the Gantt chart of Fig. 3(a), where it is easy to check that no jobs miss its deadline.*

Then we apply subroutine MCPI_PDAG. *In the first iteration we add $s1$ to G. It is not blocked by any other job, so we proceed with the second iteration. $s2$ is added to G, again we do not have any blocking. Next we add job $s3$, and we have the following blocking relations: $s1 \vdash s3$ and $s2 \vdash s3$. Thus we add the following edges to G: $s1 \rhd s3$ and $s2 \rhd s3$. Then we add $s4$. Since it is a HI job, we add the edge $s3 \rhd s4$, since $s3$ is the root of the only tree of G.*

Since $s4$ is a HI job, we run $PullUp$ on it. First we swap it with $s3$, after checking that after this operation the jobs will still meet their deadlines. Then we swap it also with $s1$ and $s2$. The result of PullUp *subroutine is shown in Fig. 12. Finally we add job L to the graph and the edge $s3 \rhd L$. Since $s3 \rightarrow L$, we may not swap further, thus obtaining the following P-DAG:*



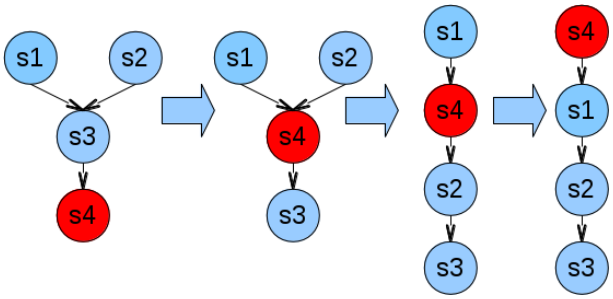


Figure 12: The effect of subroutine $PullUp$ on job $s4$.

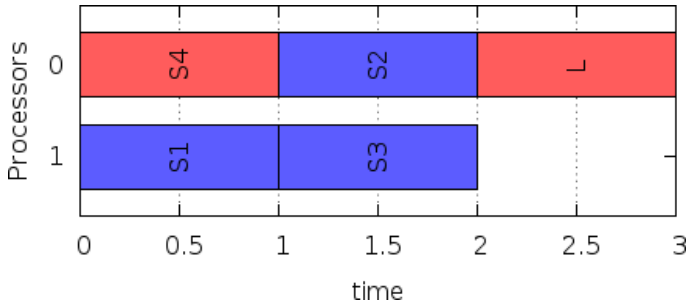


Figure 13: The schedule obtained by MCPI in Example 4.1.

From topological sort we obtain the priority table $PT = \{s4 \succ s1 \succ s2 \succ s3 \succ L\}$. *The priority table thus obtained leads to the schedule of Fig. 13. The reader may easily verify that using the initial priority assignment, the schedule will fail in scenario HI*$[s4]$, *where* $s4$ *will run for 3 time units, while using the table generated by MCPI the task graph is schedulable in HI*$[s4]$ *and HI*$[L]$.

Theorem 4.1. *The Graph produced by* $MCPI\_PDAG$ *procedure is a forest P-DAG.*

Proof. $MCPI\_PDAG$ proceeds similarly to *Forest_PDAG*, whose correctness was already shown by Theorem 3.4. There are only two differences:

1. more edges are added at each step
2. the *swap* modification is performed

Since, by Lemma 3.2, with extra edges added, G still remains a P-DAG, we observe, by Theorem 3.4, that $MCPI\_PDAG$ ensures that G is a P-DAG at least until the first swap.

To complete the proof we have to show that after the *swap* operation G remains to be a P-DAG. Let us assume by contradiction that, after a swap $TreeSwap(J^s, J^p, G)$, the resulting graph $G' = (\mathbf{J}', \triangleright)$ is no longer a P-DAG. Notice that J^s is a HI job, and thus after inserting it G becomes a connected tree. Also, after the first swap, G is still a tree, such that J^p is the new root and job J^s is the root of a subtree that contains all jobs except J^p (see Fig. 11). After multiple swaps, we will have a tree composed of a chain of LO jobs in the upper part, connected to a subtree that has J^s as root. This is illustrated in Fig. 14.

On the left side of the figure we have a tree with HI job J as root. After swapping J with J',J'' and J''' (in this order), we obtain the tree on the right side. This tree is composed of a chain of J',J'' and J''' and a subtree whose root is J.

By Lemma 3.3 and the contradicting hypothesis, we have that G' can generate a table PT' that leads to a schedule $\mathcal{S}$ such that:

$$\exists J', J'' \colon J' \vdash_{\mathcal{S}} J'' \ \wedge \ J' \not\triangleright^* J''$$

For $TreeSwap(J^s, J^p, G)$ all the possible $J' \vdash J''$ relations that were not present before the swap are such that either $J'' = J^p$ or $J^p \rightarrow^* J''$. This is because, by lowering J^p priority (*i.e.,* shifting forward its execution), it might enter in the execution window of another job and get blocked by it. The same holds for its successors in $\mathbf{T}$.

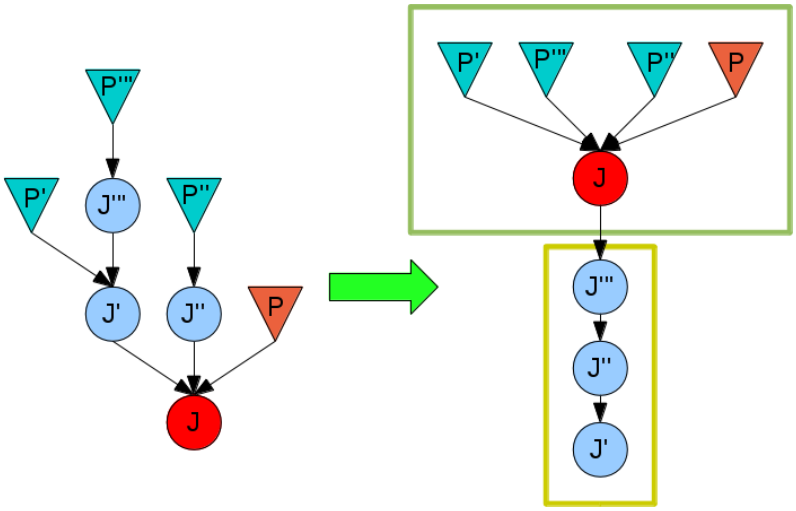


Figure 14: The effect of multiple Swaps.

For J^p, we can then rewrite our contradicting hypothesis as follows:

$$\exists J' : J' \vdash_{\mathcal{S}} J^p \ \wedge \ J' \not\triangleright^* J^p$$

After the swap, J^p is the root of a subtree ST. So $\forall J \in ST,\ J \triangleright^* J^p$. All jobs J' that are not in ST are in the chain below J^p, this means that $\forall J',\ J' \not\triangleright^* J^p \Rightarrow J^p \triangleright^* J'$ which implies that $\forall PT' \in \mathbf{PT}(G), J' \not\vdash_{\mathcal{S}} J^p$.

Let us now consider jobs J'' such that $J'' \rightarrow^* J^p$. An invariant of our algorithm is precedence compliance, *i.e.*, $J^p \rightarrow^* J'' \Rightarrow J^p \triangleright^* J''$. This means that all such J' are in the chain below J^p. The same reasoning as in the previous case holds. □

Theorem 4.2. *Let k be the number of jobs in '$\mathbf{J}$', E the number of precedence edges in '$\rightarrow$' and m the number of processors. The computational complexity of MCPI is*

$$O(Ek^2 + mk^3 \log k) \tag{7}$$

Proof. One of the main contributions to the computational complexity of the algorithm is given by the high number of list schedule simulations. Thus we will first compute the complexity of one simulation. List scheduling must keep track for each job of whether or not all its predecessors have terminated. This can be implemented by assigning each job a counter of the number of predecessors and decrementing it when a predecessor terminates. Since there are E predecessor-successor pairs, maintaining these counters gives a total contribution to the complexity linear with E. Also, each simulation sorts the jobs by the arrival times, in $O(k \log k)$ time. Every simulation examines the jobs in the order of arrivals, adding each arriving job to the priority queue data structure [13]. At most once per job arrival or termination the m greatest-priority jobs need to be selected in the queue. Once per job termination, a job is removed from the queue. Because the addition, removal and selection operations are done $O(k)$ times and each priority-queue operation costs $O(\log k)$, the cost of all queue operations is $O(mk \log k)$. The complexity of one simulation is thus:

$$O(E + mk \log k) \tag{8}$$

Let us now analyze the algorithm of Figure 7 line by line. Routine $PTTransform$ has a complexity of $O(k^2)$. This is because we run through the linked list of all jobs, and for each of them we move all the predecessors that have a lower priority in front of the current job, and the maximum number of predecessors for each job is $O(k)$. $CheckLOscenarioSchedulability$ does one simulation, thus it has a complexity of (8). $MCPI\_PDAG$ gives the highest contribution, and its complexity will be discussed later. $TopologicalSort$ has complexity $O(k)$ [13]. Finally, $anyScenarioFailure$ does $O(k)$ simulations, and thus its complexity is $O(k(E + mk \log k))$.

Let us now analyze routine $MCPI\_PDAG$. This is a recursive subroutine that is called exactly k times. This subroutine, after some $O(1)$ operations, performs a simulation, which gives a total contribution of $O(k(E + mk \log k))$. Then for each subtree a $ConnectAsRoot$ operation is performed. One such operation has a linear complexity in jobs, because we have to find the root of a subtree. There are $O(k)$ subtrees, thus this operation yields a total contribution of $O(k^3)$. Finally we have to analyze the complexity of $PullUp$ (Figure 9). In this subroutine there is a while loop that is executed once for each LO predecessor of the current job, thus a $O(k)$ number of times inside $PullUp$ and $O(k^2)$ number of times per one one execution of $MCPI$. All the operations performed in the subroutine are $O(1)$ except for $CanSwap$, which performs a simulation and thus it has complexity (8). Thus the $CanSwap$ subroutine gives the main total contribution to the complexity of the algorithm, executing in total $O(k^2)$ simulations of complexity (8), which gives the result given in (7).

□

Notice that for large practical problem instances it can be expected that $m \ll k$, and also m is usually considered as a constant given by the platform. Also, even if in general $E = O(k^2)$, having a quadratic number of precedence edges is really unrealistic in parallel programs, as this situation is likely to seriously restrict the possibility of parallel execution. If we consider only the cases where the number of job inputs and outputs is bounded by a constant then the number of precedence edges would grow linearly with the number of jobs. Under the assumptions mentioned here Formula (7) can be simplified as follows:

$$O(k^3 \log k) \tag{9}$$

4.2 Predictable Online Policy

The online policy should be predictable, in the sense that lowering the execution times may not increase the termination time. List scheduling is, in general, non-predictable. Therefore, online we execute a predictable policy that behaves the same way as list scheduling in basic scenarios. Recall that offline we check schedulability by simulating all basic scenarios. For each of them we record all jobs start time in a table and provide the table to the online policy. Online, we keep track of the current basic scenario, assuming LO when in LO mode and HI$[J_h]$ when job J_h causes a switch to HI mode. We assume that jobs arrive not at their nominal arrival times, but at their offline start times specified in the table of the current scenario. The modified arrival times ensure that precedences are satisfied. Therefore, while preserving the precedence constraints, instead of list scheduling our online policy uses the default classical global fixed priority scheduling, which is known to be predictable.

5 Related Work: Discussion and Analysis

Although our scope is finite set of jobs, most of the literature concerns with instances that have an infinite set of jobs, generated by periodic or sporadic tasks. Periodic tasks are said to be synchronous if the offsets between the first arrival of different tasks are statically known. The deadlines can be implicit (*i.e.,* equal to the period), constrained (*i.e.,* less or equal to the period) or pipelined (*i.e.,* larger than period).

Our work can be applied for scheduling the hyperperiod of *periodic synchronous non-pipelined*(*i.e.,* implicit or constrained-deadline) tasks with *precedence constraints*. However, we still consider general real-time policies, even if not originally designed for such systems, as they can be reused as *starting point for our priority-improvement algorithm*. We are particularly interested in the policies tailored for multiprocessor systems, assuming *global fixed priority* for jobs.

5.1 Multiprocessor Scheduling

Whereas for uniprocessor scheduling a fixed-job-priority algorithm (EDF) is optimal, for multiprocessor case, dynamic job priorities are essential for optimality[5]. Moreover, the EDF heuristic can be very inefficient for multiprocessors. In seminal work of Dhall and Liu [14] it was shown that the *best*, *i.e.,* maximal, load that can be guaranteed for any schedulable job instance for EDF on multiprocessors is no better than for EDF on uniprocessor. For arbitrarily small $\epsilon > 0$ one can find a feasible job instance with load $1 + \epsilon$

that is not schedulable by EDF. For this, let us consider m small-deadline jobs with utilization ϵ/m each and one job with utilization 1 and a large deadline. If the last job, which has a large utilization, was given the highest priority then the schedule would be feasible.

In [15] it was shown that in general implicit-deadline periodic task sets under global *fixed priority* for jobs have the following best guaranteed utilization: $(m+1)/2$. Roughly speaking, the fixed priority scheduling can be guaranteed to find a multiprocessor schedule if the system is loaded by no more than *one half*, and even this is only possible if job priorities are well calculated, *e.g.,* the plain EDF cannot provide this guarantee, as explained earlier. Therefore, EDF modifications have been proposed to provide this guarantee. The main idea of several such algorithms is so-called 'separation' of jobs, *i.e.,* separating those that have low and high contribution to load. One of such algorithms is fpEDF, formulated for periodic tasks [15], and later on generalized to sporadic tasks under name *EDF-DS*, where DS stands for *density separation* (see [5] for references). In our notation, this algorithm computes job density as $\delta_i = C_i/(D_i - A_i)$ and it differs from EDF by always giving the jobs with $\delta_i > th$ the highest priority, for a certain threshold th. Ties are broken arbitrarily. For the other jobs, the priority is the default EDF. Obviously, this strategy resolves the Dhall-effect counterexample mentioned earlier. However this approach does not give any schedulability assurance in the case of finite sets of jobs. Experiments shows that it can even decrease the schedulability using a threshold $th = 1/2$. For such cases, experiments suggest to use a higher threshold to improve schedulability. In the experiments presented here the threshold is set to: $th = 1$, *i.e.,* only density-one jobs get the highest priority, whereas in future work we will investigate other thresholds.

5.2 Precedence-constrained Scheduling

The *list scheduling* can be seen as generalization of fixed-priority scheduling by handling precedence constraints using *synchronization* between dependent jobs, *i.e.,* including wait for predecessor completion into the condition of job 'ready' status. Synchronization is essential for multiprocessors, whereas for single processor systems it may be sufficient to require precedence compliance of the priority [16, 1]. In both cases, it is generally recognized that the definition of EDF heuristics should be adjusted by using *ALAP deadlines* D^* instead of the nominal deadlines for priority assignment. For example, the list scheduling knows so-called 'ALAP' and b-level heuristics [12]. Single-processor scheduling uses this approach for priority assignment with adjusted deadlines [16]. Sometimes the ALAP-adjusted EDF is a part of an optimal strategy, see [12] for further references.

5.3 Mixed-critical Scheduling

There are many works on mixed-critical scheduling for uniprocessor systems without precedence constraints. These works compute priorities either by a variant of Audsley approach or by improving the EDF priorities. Our previous work, MCEDF [7] algorithm can be seen as a combination of the two, also based on P-DAG. Another uniprocessor algorithm should be mentioned, OCBP, which applies Audsley approach to obtain optimal priority tables for FP (fixed priority) policy. Unlike OCBP, MCEDF exploits FPM scheduling policy and, due to this advantage, it has been shown to dominate OCBP [7]. On the other hand, it should be also mentioned that OCBP was generalized to precedence-constrained instances in [1]. Possible dominance of some precedence-aware MCEDF extension over the precedence-aware OCBP is subject of future work.

Compared to MCEDF, in the present paper we extended the P-DAG analysis to support precedence relation and multiple processors. Moreover we abandon Audsley approach replacing it by more elaborate priority improvement in P-DAGs, while, by the following observation, we also offer a generalization of MCEDF.

Observation 5.1. *For single processor and without precedence constraints, a slightly modified MCPI(EDF) is equivalent to MCEDF and both algorithms are optimal among those that put HI jobs in EDF order.*

The mentioned modification of our algorithm leads to a slight improvement in schedulability, which is practically invisible in the random experiments but is necessary to establish the properties mentioned above. Note that the above observation also implies dominance of the modified MCPI(EDF) over OCBP

m	jobs	arcs	step	δ	σ_s	instances	EDF	EDF-DS	MCPI(EDF)	MCPI(EDF-DS)	diff(%)	diff-DS(%)
2	30	20	0.005	0.01	3.2	128800	20924	21023	27375	27467	**30.83%**	**30.65 %**
4	60	40	0.02	0.05	6	50500	6839	6887	8263	8310	**20.82%**	**20.66 %**
8	120	80	0.05	0.125	12	31575	3065	3082	3521	3538	**14.88%**	**14.80%**

Table 1: Experimental results.

for the case of no precedence constraints. We describe the modification, formalize the above claim, and prove it in Appendix A.

EDF sets the PT in the increasing deadline order, therefore the EDF improvement strategies perform *deadline modification* of HI jobs, reducing their deadlines to improve their priorities *w.r.t.* LO jobs and re-use EDF schedulability analyzes for the modified problem instance. One of the strategies for deadline modifications scales the relative deadline of all HI jobs by the same factor $x, 0 < x < 1$. This strategy was generalized for multiprocessors in [17], where it was combined with EDF-DS.

There are only a few works on precedence-constrained mixed-criticality scheduling. For single processor, [1] generalizes Audsley approach based algorithm OCBP to support precedence constraints for synchronous systems. In [18], multiprocessor list scheduling algorithm was proposed. However, it is restricted to jobs that all have the same arrival and deadline times. Finally, [19] consider pipelined scheduling for task graphs. However, they implicitly assume that the deadlines are large enough, such that they can be ignored during the problem solving, as only period (throughput) constraints were considered and not deadline (latency) ones.

5.4 Analysis

From the analysis of literature we make the following choices. For multiprocessor scheduling we use density separation, *i.e.,* EDF-DS, for the construction of FPM priority tables: PT_{LO} and PT_{HI} To represent the state-of-the art approach to mixed critical multiprocessor scheduling, we apply deadline modification to the HI jobs, but instead of the deadline scaling, we use the deadlines D_{MIX}, which anyway should be met in the LO mode. In fact, we base the construction of the LO priority table on the MIX-task graph, $\mathbf{T}_{\text{MIX}}$. In this graph we calculate ALAP deadlines. The resulting values for $D_{\text{MIX}}{}^*$ are substituted as the 'deadlines' when calculating the job density and deadline-based priority in the context of EDF-DS. The resulting LO priority table serves as input for MCPI. For fair comparison with related work in the experiments, we use this table as the reference to evaluate the improvement brought by the MCPI into this table. For the HI table PT_{HI} we use the ALAP deadlines calculated in HI task graph $\mathbf{T}_{\text{HI}}$.

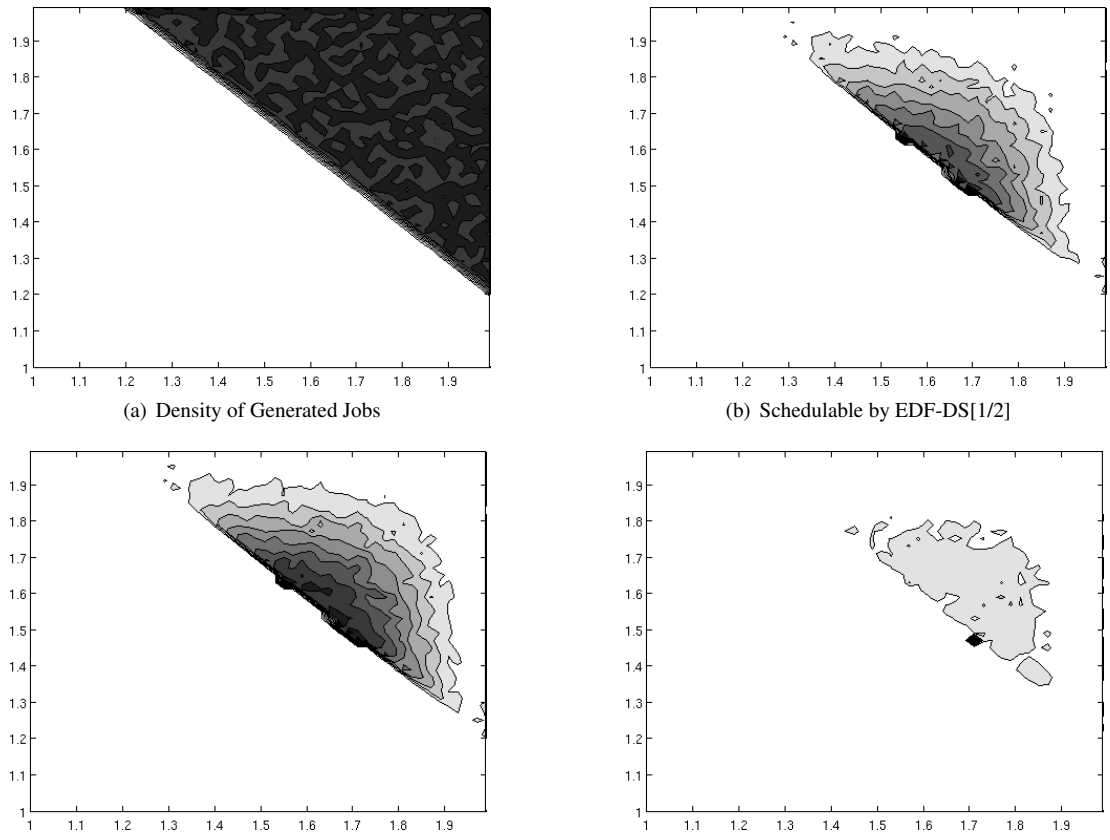
6 Implementation and Experiments

We evaluated the schedulability performance of MCPI comparing it with those the performance of the support algorithms. We randomly generated task graphs with integer timing parameters. Every task graph was generated for a target LO and HI stress pair. The method to generate the random problem instances is similar to the one used in [7]. We restricted our experiments to "hard" task graphs, *i.e.,* those satisfying the following formula:

$$Stress_{\text{LO}}(\mathbf{T}) + Stress_{\text{HI}}(\mathbf{T}) \geq \sigma_s \tag{10}$$

The reason of this choice is that task graphs under that line are relatively easy to schedule. We ran multiple job generation experiments, ranging the target of $Stress_{\text{LO}}$ and $Stress_{\text{HI}}$ in the area defined by (10) with a fixed step s. Per each target, ten experiments were run, generating the points lying near the target with a certain tolerance δ. The result of the experiments are shown in Table 1. We ran experiments for 2, 4 and 8 processors. For each generated task graph, we checked the schedulability of EDF, EDF-DS, MCPI(EDF), MCPI(EDF-DS). All algorithms were applied using the FPM scheduling policy, the ALAP and ASAP arrivals and deadlines, based upon modified deadline D_{MIX} in the LO mode, as described in Section 5. From the result we can see that MCPI gives a big improvement in schedulability compared to the support algorithm, reaching a maximum of 30.83%.

Fig. 15 and Fig. 16 give the contour graph of the density of the generated points in grayscale, where black is the maximum value and white is 0. The horizontal axis is $Load_{\text{LO}}$, the vertical is $Load_{\text{HI}}$. We used



(a) Density of Generated Jobs

(b) Schedulable by EDF-DS[1/2]

(c) Schedulable by MCPI

(d) Schedulable by MCPI and not by EDF-DS[1/2]

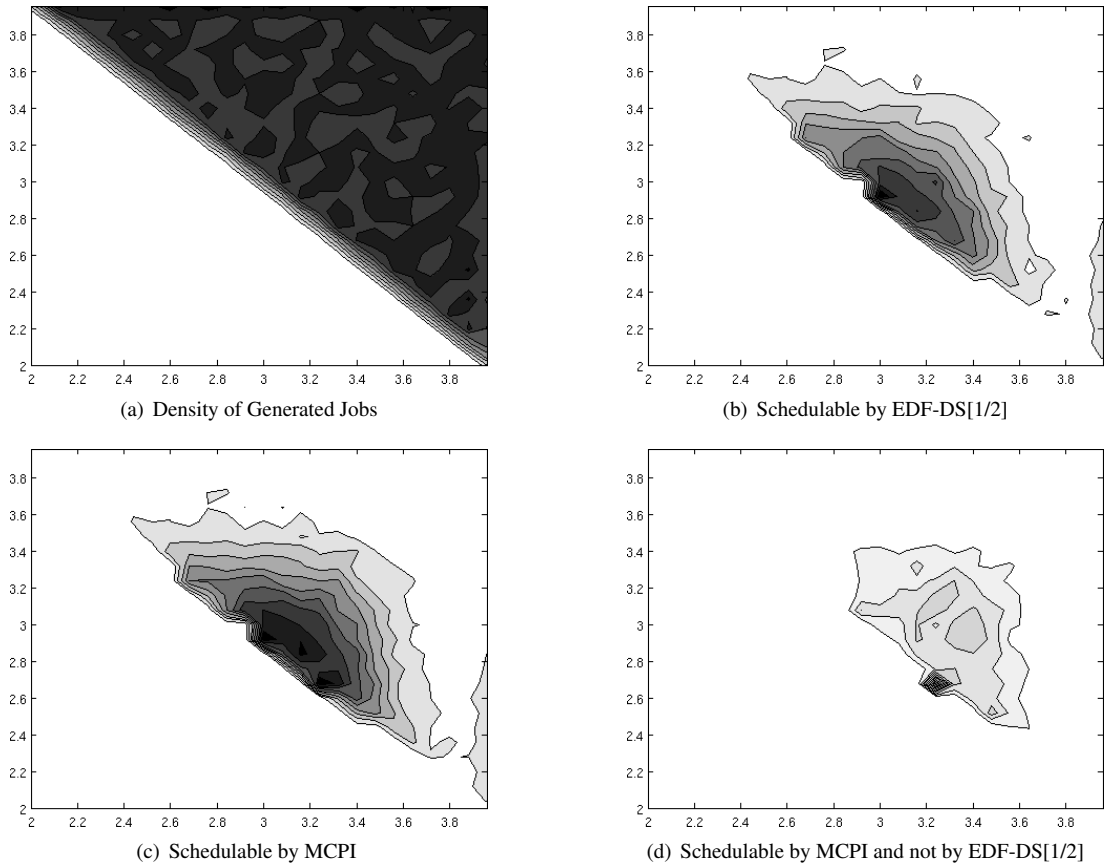
Figure 15: The contour graphs of random task graphs for 2 processors. The horizontal axis is $Load_{\text{LO}}$, the vertical is $Load_{\text{HI}}$.

Load in the axes because it better reflects required parallelism. Figures from Fig. 15(a) to Fig. 15(d) refer to the experiments made for 2 processors. In particular Fig. 15(a) shows the density of the generated task graphs, Fig. 15(b) shows the percentage of instances schedulable by EDF-DS among the generated ones. Likewise Fig. 15(c) shows the percentage of task graphs schedulable by MCPI (EDF-DS) and Fig. 15(d) shows the percentage of task graphs schedulable by MCPI (EDF-DS) and not schedulable by EDF-DS. As expected the schedulability decreases while the distance from the axis origin increase. Fig. 15(d) is particularly interesting, because it shows how MCPI increases the schedulability over the support algorithm when the load increases. Notice that approximately around point $(1.7, 1.7)$ the density is higher, suggesting that around this point MCPI is more effective.

Figures from Fig. 16(a) to Fig. 16(d) show respectively the same information of figures from Fig. 15(a) to Fig. 15(d), but referred to experiments on 4 processors. From those graph we have confirmation of the conclusions made above. Also in Fig. 16(d) we have an area where MCPI is particularly effective, approximately around point $(3.3, 3.1)$.

7 Conclusions

We addressed the problem of multi-processor scheduling of mixed criticality task graphs in synchronous systems. The advantage of our algorithm over state of the art was demonstrated by experiments on a large set of synthetic benchmarks, demonstrating a good improvement in schedulability.



(a) Density of Generated Jobs

(b) Schedulable by EDF-DS[1/2]

(c) Schedulable by MCPI

(d) Schedulable by MCPI and not by EDF-DS[1/2]

Figure 16: The contour graphs of random task graphs for 4 processors. The horizontal axis is $Load_{\text{LO}}$, the vertical is $Load_{\text{HI}}$.

In multi-processor scheduling is hard to apply the Audsley approach, previously proven effective for single-processor mixed-critical scheduling with precedence constraints [1]. Therefore in our algorithm, MCPI , we assign the priorities in a different order. Nevertheless, MCPI still generalizes an Audsley-approach compliant algorithm MCEDF [7], when applied to single-processor instances without precedences.

In future work, we plan to extend the algorithm for multiple criticality levels and to support pipelining.

References

[1] S. Baruah, "Semantics-preserving implementation of multirate mixed-criticality synchronous programs," in *RTNS'12*, pp. 11–19, ACM, 2012. 1, 5.2, 5.3, 5.3, 7

[2] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Stanfill, D. Stuart, and R. Urzi, "White paper: A research agenda for mixed-criticality systems," Apr. 2009. 1

[3] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Trans. Comput.*, vol. 61, pp. 1140 –1152, aug. 2012. 1, 2.1, 2.1

[4] C. Ficek, N. Feiertag, and K. Richter, "Applying the AUTOSAR timing protection to build safe and efficient ISO 26262 mixed-criticality systems," in *ERTSS'2012*, 2012. 1

[5] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, Oct. 2011. 1, 5.1

[6] R. Ha and J. W. S. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *Proc. Int. Conf. Distributed Computing Systems*, pp. 162–171, Jun 1994. 2.1

[7] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Mixed critical earliest deadline first," in *Euromicro Conf. on Real-Time Systems*, ECRTS'13, pp. 93–102, IEEE, 2013. 2.1, 5.3, 6, 7, A, A.3

[8] J. W. S. Liu, *Real-Time Systems*. Prentice-Hall, Inc., 2000. 2.2

[9] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pp. 9 pp.–329, Dec 2005. 2.2

[10] H. Li and S. Baruah, "Load-based schedulability analysis of certifiable mixed-criticality systems," in *Intern. Conf. on Embedded Software*, EMSOFT '10, pp. 99–108, ACM, 2010. 2.2

[11] T. Park and S. Kim, "Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems," in *Intern. Conf. on Embedded software*, EMSOFT '11, pp. 253–262, ACM, 2011. 2.2

[12] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, pp. 406–471, Dec. 1999. 2.2, 5.2

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001. 4.1, 4.1, 4.1

[14] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978. 5.1

[15] S. K. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," *IEEE Trans. Comput.*, vol. 53, pp. 781–784, June 2004. 5.1

[16] J. Forget *et al.*, "Scheduling dependent periodic tasks without synchronization mechanisms," in *RTAS'10*, pp. 301–310. 5.2

[17] H. Li and S. K. Baruah, "Outstanding paper award: Global mixed-criticality scheduling on multiprocessors," in *24th Euromicro Conference on Real-Time Systems, ECRTS 2012*, 2012. 5.3

[18] S. Baruah, "Implementing mixed-criticality synchronous reactive systems upon multiprocessor platforms." 5.3

[19] E. Yip, M. Kuo, P. S. Roop, and D. Broman, "Relaxing the Synchronous Approach for Mixed-Criticality Systems," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2014. 5.3

Appendices

A MCEDF and MCPI(EDF): Modifications and Optimality

In this appendix we formalize and prove Observation 5.1, which states that the MCEDF algorithm, proposed in [7], and a slightly modified MCPI(EDF) have equivalent schedulability. Note that because all these algorithms use LO-mode schedules to construct the priority tables, under the 'scheduling' we always mean the LO-mode scheduling unless mentioned otherwise.

A.1 Modified Version of MCPI

In this subsection we formulate a modified version of MCPI that is 'closer' to MCEDF. Additional experiments show results that are statistically indistinguishable from those of MCPI. In a later subsection we show that the modified MCis also equivalent to MCEDF, thus completing the argument of equivalence between MCPI and MCEDF.

The modified version differs from MCPI by replacing subroutine *MCPI_PDAG* by the subroutine *modMCPI_PDAG*, shown in Figure 17. Also the *PullUp* subroutine is replaced by *modPullup*, where swap is defined differently.

The modified MCPI is based on the following concept.

Definition 4 (Potential Interference Relation)**.** *Given task graph* $\mathbf{T}(\mathbf{J}, \rightarrow)$*, number of processors* m *and a subset* $\mathbf{J}' \subseteq \mathbf{J}$*, we say that an equivalence relation* $\overset{\mathbf{J}'}{\sim}$ *on set* $\mathbf{J}'$ *is a 'potential interference' relation if it has the following property:*

$$\forall J_1, J_2 \in \mathbf{J}'.\ \exists PT : J_1 \vdash_{PT} J_2 \ \Rightarrow\ J_1 \overset{\mathbf{J}'}{\sim} J_2$$

whereby we consider LO-mode m*-processor list schedules on maximal task subgraph with nodes* $\mathbf{J}'$*.*

In modified MCPI we exploit the fact that if a potential interference relation is known then any two unrelated jobs can be kept in two different subtrees of a P-DAG even when one modifies the priorities in one of the subtrees, *e.g.,* if one performs the priority swap operations. In general, there exist multiple potential interference relations, as joining two equivalence classes would lead to a new potential interference relation. Therefore, the (unique) maximal such relation is the total equivalence. The (unique) minimal potential interference relation can be obtained by union of blocking relations under all possible PT's, followed by transitive and reflexive closure, however it is a costly computation due to exponential number of PT's. Instead of computing this minimum, we over-approximate it by exploiting the following theorem (given without proof).

Theorem A.1 (Single-Processor Interference)**.** *In preemptive list scheduling, a potential interference relation for single processor is also a potential interference relation for* m *processors.*

As it turns out (see the next section), calculating the minimal potential interference on a single processor can be done by a fast (almost linear) algorithm (the 'makespan'). Therefore, we use the single-processor approximation, though it can be very rough one, and more refined approximations could be defined.

Observation A.2 (Optimality Requirement on Interference Approximation)**.** *For the single-processor optimality results established in this section, the only requirement that we place on interference-relation over-approximation algorithm is that it gives the exact minimal interference relation at least for single-processor problem instances without precedence constraints (and which can be always ensured by using makespan).*

The modified MCPI algorithm differs from the original MCPI by the way it handles the HI jobs in the P-DAG construction. It employs a potential interference relation to try to put these jobs in separate sub-trees when pulling them up the P-DAG, reducing the 'job-chaining' in the P-DAG which we showed in Fig. 14. Note that the modified MCPI would behave exactly as the basic one if one used the worst

1: **Algorithm:** $modMCPI\_PDAG$
2: **Input:** **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
3: **Input:** **priority table** SPT
4: **In/out:** **forest P-DAG** $G(\mathbf{J}', \rhd)$
5: **if** $\mathbf{J} \neq \emptyset$ **then**
6: $J^{\text{curr}} \leftarrow SelectHighestPriorityJob(\mathbf{J}, SPT)$
7: $\vdash \leftarrow SimulateListSchedule(\text{LO},\ (G.\mathbf{J}', \rightarrow),\ \mathbf{PT}(G) \frown J^{\text{curr}})$
8: $G.\mathbf{J}' \leftarrow G.\mathbf{J}' \cup \{J^{\text{curr}}\}$
9: **for** *all trees* $ST \in G$ **do**
10: **if** $\chi(J^{\text{curr}}) = \text{LO}$ **then**
11: **if** $\exists\ J' \in ST:\ J' \vdash J^{\text{curr}} \ \vee\ J' \rightarrow J^{\text{curr}}$ **then**
12: $ConnectAsRoot(ST, J^{\text{curr}})$
13: **end if**
14: **else**
15: **if** $\exists\ J' \in ST:\ J' \overset{\mathbf{J}'}{\sim} J^{\text{curr}} \ \vee\ J' \rightarrow J^{\text{curr}}$ **then**
16: $ConnectAsRoot(ST, J^{\text{curr}})$
17: **end if**
18: **end if**
19: **end for**
20: **if** $\chi(J^{\text{curr}}) = \text{HI}$ **then** $modPullUp(J^{\text{curr}}, G, SPT)$
21: $modMCPI\_PDAG(\ \mathbf{T}(\ \mathbf{J} \setminus \{J^{\text{curr}}\},\ \rightarrow),\ SPT,\ G(\mathbf{J}', \rhd))$
22: **end if**

Figure 17: The algorithm for computing P-DAG in MCPI - modified version

interference approximation (*i.e.,* the total equivalence), but, obviously, such an approximation would not satisfy the optimality requirement in general.

The P-DAG construction algorithm for the modified MCPI is shown in Figure 17. We see that the HI jobs now are connected only to the trees that may potentially interfere with them and hence also with each other when priorities are modified by the priority improvement. The modified priority improvement, $modPullUp$ calls a modified version of $TreeSwap$, denoted $modTreeSwap$.

Definition 5 (Modified Swap)**.** *Let $G(\mathbf{J}', \rhd)$ be a forest P-DAG, let $J_{LO} \rhd J_{HI}$ and let $\mathbf{J}''$ represent the subset of jobs whose priorities can be potentially higher than or equal to J_{HI} after the swap is performed:*

$$\mathbf{J}'' = \{J_{HI}\} \cup \{J' \mid J' \rhd^* J_{HI}\} \setminus \{J_{LO}\}$$

Subroutine $modTreeSwap(J_{HI}, J_{LO}, G)$ performs the following 'swap' transformation on graph G:

1. *$J_{LO} \rhd J_{HI}$ is transformed into $J_{HI} \rhd J_{LO}$*
2. *$\forall$ tree ST : $root(ST) \rhd J_{HI} \ \vee\ root(ST) \rhd J_{LO}$*
   (a) ***if*** *$\exists J' \in ST : J' \overset{\mathbf{J}''}{\sim} J_{HI} \ \vee\ J' \rightarrow J_{HI}$*
   then *in the new G: $root(ST) \rhd J_{HI}$*
   (b) ***else*** *in the new G: $root(ST) \rhd J_{LO}$*
3. ***if*** *$\exists J_s$: $J_{HI} \rhd J_s$* ***then*** *$J_{HI} \rhd J_s$ is transformed into $J_{LO} \rhd J_s$*

The difference of the modified 'swap' operation from the basic version used by algorithm in Fig. 9 is only in the second rule above. The basic version does not distinguish the two cases in the second rule and always 'plugs' the subtrees ST into J_{HI}. The modified version only plugs the subtree into J_{HI} if after priority modifications the subtree can be involved in a blocking relation with J_{HI} and/or other subtrees. Such a blocking relation would invalidate its property of being an independent tree in a P-DAG. If under no circumstances such a blocking relation can appear, the tree is plugged to the lower-priority job J_{LO}. As a result, instead of a LO-job chain below the HI job shown in Fig. 14 we may see a stem to which side trees may be plugged. Because, due to the check involving the '$\overset{\mathbf{J}''}{\sim}$' relation, the side subtrees can never block any job from the subtrees higher in the stem, the proposed modification to the swapping procedure may

1: **Algorithm:** $modPullUp$
2: **Input:** **job** J
3: **Input:** **priority table** SPT
4: **In/out:** **forest P-DAG** G
5: $DONE = \emptyset$
6: **while** $LOpredecessors(J, G) \neq DONE$ **do**
7: $J' \leftarrow SelectLeastPriorityJob(\ (\ LOpredecessors(J, G)\ \setminus\ DONE),\ SPT)$
8: $DONE \leftarrow DONE \cup \{J'\}$
9: **if** $CanSwap(J, J', G)$ **then**
10: $modTreeSwap(J, J', G)$
11: $DONE \leftarrow\ DONE\ \cap\ LOpredecessors(J, G)$
12: **end if**
13: **end while**

Figure 18: The modified pull-up subroutine

never lead to possible violations of the P-DAG property. Therefore, we adapt, without proof, Theorem 4.1 to the modified version of MCPI:

Theorem A.3. *The Graph produced by $modMCPI\_PDAG$ procedure is a P-DAG. Moreover, after each basic step of the algorithm – the initial connection of a new job J^{curr} and a tree swap – the intermediate graph G is a P-DAG as well.*

The modified $PullUp$ procedure, which exploits the new swap operation, is illustrated in Fig. 18. Unlike the basic $PullUp$, Fig. 9, instead of keeping a list of predecessors that were not yet considered for swapping we keep its complement – the list of predecessors that have already been considered for swapping ('done'). Like in the basic algorithm, we do not re-try to swap any such predecessor again, not even after another job has been swapped successfully. The main reason for this restriction is computational complexity, we would like to keep the worst case number of swap trials linear in the total number of jobs.

Note that the modified algorithm can potentially directly connect two LO jobs by P-DAG edge that is incompatible with SPT priority order, which can happen if a swap will fail and another one will succeed. Note also that we use the same $CanSwap$ procedure as before. We could have used a modified subroutine $modCanSwap$, which would differ from the basic $CanSwap$ only in that it evaluates the schedulability of the P-DAG obtained from modified swap transformation, not the basic one. However, one can easily show that the schedulability of both P-DAGs is equivalent (because they differ only in the way they connect the non-interfering subtrees). However, as discussed earlier, the two different swapping methods result in different P-DAG structures, and, as we see in some examples, this may have essential effect on the HI jobs that are pulled through the same region of the P-DAG later than the current HI job.

A.2 Single-processor Scheduling and Busy Intervals

An important concept in the context of single-processor scheduling is busy interval. However, because this concept is better studied and understood for the case of no precedence constraints, we will define it using the notion of 'modeling job set'.

Definition 6 (Modeling job set)**.** *Given a task graph $\mathbf{T}(\mathbf{J}, \rightarrow)$, its modeling job set $\mathbf{J}^*$ is the set of jobs whose arrival times are calculated as ASAP times A_j^* for LO mode.*

Observation A.4 (Modeling job set identity when no precedences)**.** *The modeling job set is identical to the original job set if the task graph has no precedence edges.*

Definition 7 (Predecessor-closed subset)**.** *Given a task graph $\mathbf{T}(\mathbf{J}, \rightarrow)$, a subset of jobs $\mathbf{J}' \subseteq \mathbf{J}$ is predecessor-closed if including a job in this subset implies including all its predecessors.*

The following theorem is given without proof:

Theorem A.5 (Modeling list schedule by fixed priority)**.** *Given a predecessor-closed subset and considering single processor schedules, then using any priority table that is priority-compliant one gets the same basic LO scenario schedules from:*

1. *the list scheduling of the corresponding maximal subgraph*
2. *fixed-priority scheduling of the corresponding modeling subset*

Definition 8 (Busy Interval)**.** *A busy interval for a predecessor-closed subset of jobs* $\mathbf{J}'$ *is a time interval in a fixed-priority schedule for the corresponding modeling job subset* $\mathbf{J}'^*$*. It is defined as a maximal time interval* $(\tau_1, \tau_2]$ *where at least job is ready for execution. By abuse of terminology, we apply the term 'busy interval' also to the subset of jobs running in that interval, and denote it* BI*.*

Note that the time interval in the definition is half-open because the jobs that arrive at time t count ready only for the time instances strictly later than t.

It is obvious that on a single processor either the start time nor the length of a busy interval depends on the exact priority assignment, because the former corresponds to the earliest job arrival and the latter is equal to the sum of $C_i(\mathrm{LO})$ of all jobs in the interval. In general, a job set $\mathbf{J}'$ can be partitioned into multiple BI subsets, because some jobs may arrive at or later than the end of a busy interval of some other jobs. To calculate the busy intervals one can simulate the fixed-priority policy on the modeling job set for any priority table and apply the definition of busy interval to find the time bounds and the contents of job subsets that belong to the same busy interval.

The following lemma is easy to prove:

Lemma A.6 (Least priority in a busy interval)**.** *Given a job set* $\mathbf{J}'$ *and any of its busy interval* BI *with time interval* $(\tau_1, \tau_2]$*. In fixed-priority scheduling for job set* $\mathbf{J}'$*, the least-priority job running in this* BI *terminates at time* τ_2 *and is blocked by at least one other job in* BI *(if there are any).*

The following theorem can be easily derived from the above lemma:

Theorem A.7 (Minimal Potential Interference in Busy Intervals)**.** *Given a job set* $\mathbf{J}'$ *without precedences, then the set of busy intervals* BI *are the equivalence classes of the corresponding minimal potential interference relation* $\overset{\mathbf{J}'}{\sim}$ *on single processor.*

Corollary A.8 (Potential interference with precedences)**.** *Given a task graph and a predecessor-closed subset of jobs* $\mathbf{J}'$*. The set of busy intervals* BI *correspond to equivalence classes of some (not necessarily minimal) potential interference relation* $\overset{\mathbf{J}'}{\sim}$ *on single processor.*

We cannot claim minimality in the second case above because currently we are not sure about it.

Observation A.9 (Implementation of Modified MCPI based on Busy Intervals)**.** *It can be easily shown that modified MCPI evaluates relation* $\overset{\mathbf{J}'}{\sim}$ *only for predecessor-closed subsets. Therefore, it can use busy intervals for this evaluation. Because to obtain busy intervals one requires to do just one extra simulation for a subset of jobs where at least one simulation needs to be done anyway, the modified MCPI based on busy intervals has the same computational complexity as the basic MCPI. Moreover, by the theorem above, busy intervals satisfy our requirement of giving exact evaluation of the minimal single-processor interference for the case without precedence constraints.*

A.3 Recalling the MCEDF Algorithm

MCEDF is defined, which are, compared to the assumptions of this paper, restricted in two different ways:

(1) assume *single processor* platform, $m = 1$

(2) assume *no precedence* between jobs, $\rightarrow = \emptyset$

Due to the second restriction, we consider just *job sets* $\mathbf{J}$ rather than *task graphs* $\mathbf{T}$ as problem instances, and simple fixed-priority policy instead of list scheduling.

In this section we recall the definition of MCEDF from [7], while adapting this description to the terminology and notations of this paper.

1: **Algorithm:** $MCEDF\_PDAG$
2: **Input:** **job set J**
3: **Input:** **node** J_{parent}
4: **In/out:** **P-DAG** G
5: **Input:** **EDF-compliant priority table** SPT
6:
7: $\mathbf{B} \leftarrow PartitionIntoBIs(\mathbf{J})$;
8: **for** all $BI \in \mathbf{B}$ **do**
9: $J^{\text{least}} \leftarrow AssignJobLeastPriority(BI, SPT)$
10: $G.\mathbf{J} \leftarrow G.\mathbf{J} \cup \{J^{\text{least}}\}$
11: **if** $J_{parent} \neq \emptyset$ **then**
12: $G.\rhd \leftarrow G.\rhd \cup \{(J^{\text{least}}, J_{parent})\}$
13: **end if**
14: $\mathbf{J}' \leftarrow BI \setminus \{J^{\text{least}}\}$
15: $MCEDF\_PDAG(\mathbf{J}', J^{\text{least}}, G)$
16: **end for**

Figure 19: The MCEDF algorithm for computing P-DAG

The MCEDF algorithm has the same basic structure as MCPI. The top-level structure of MCEDF basically repeats that of MCPI given in Fig. 6, except that it does not need to call *PTTransform*. Though it is not explicit in the original description of MCEDF, similarly to MCPI, it also uses a support priority table SPT. The point is that it requires that SPT must be *EDF-compliant*, but if all jobs have different deadlines (which is often the case) than there exists only one unique such table for the given job set. For the theoretical properties studied in [7] the choice of the particular EDF-compliant table does not have an influence, but in this description we emphasize the SPT table as are final goal is to prove the equivalence of (modified) MCPI and MCEDF under the condition that they use the *same* SPT.

Though, like modified MCPI, MCEDF also employs busy intervals, it uses an essentially different algorithm for constructing the P-DAG, shown as subroutine $MCEDF\_PDAG$ in Figure 19.

The P-DAG construction algorithm splits the instance into BI's and assigns one of the jobs in each BI the least priority. Recall from Lemma A.6, that in a busy interval $(\tau_1, \tau_2]$, the job assigned the least priority will complete at time τ_2.

The two candidates to be assigned the least priority by MCEDF are $J^{\text{low}}_{\text{LO}}$ and $J^{\text{low}}_{\text{HI}}$, which are the least SPT-priority jobs LO resp. HI jobs among those in the busy interval. Note that since the SPT table is EDF compliant, those two jobs are also latest-deadline among the LO resp. HI jobs present in the given BI.

Subroutine $AssignJobLeastPriority$ selects the least priority job according to the following rule.

- **if** $\exists J_j \in BI : \chi_j = \text{LO} \ \wedge \ J^{\text{low}}_{\text{LO}}.D \geq \tau_2$
- **then** $J^{\text{least}} \leftarrow J^{\text{low}}_{\text{LO}}$
- **else** $J^{\text{least}} \leftarrow J^{\text{low}}_{\text{HI}}$

This rule prefers to assign the least priority to $J^{\text{low}}_{\text{LO}}$ if BI has some LO jobs and if a latest-deadline one among them would not miss its deadline. Otherwise the algorithm has no other choice but to select a HI job. Thus, the algorithm greedily avoids assigning a HI job the least priority, and does so only when this cannot be avoided. Intuitively, this is similar to trying to pull a HI job as high as possible in the P-DAG in the context of MCPI.

After assigning the least priority in the given BI the algorithm continues recursively with sub-instances $\mathbf{J}' = BI \setminus \{J^{\text{least}}\}$. Removing a job from a BI reveals further fragmentation into busy intervals, which become direct children of BI in the P-DAG. In those new BI's the same algorithm is used to find the least-priority job and to construct the subtree further from the roots to the leafs.

The final G P-DAG of the MCEDF algorithm has multiple subtrees whose root nodes correspond to the BI's of the complete problem instance $\mathbf{J}$. If we consider each subtree separately as a subset $\mathbf{J}'$ and

remove the root of the subtree from the subset, then, by construction, different children of the root would correspond to the busy intervals of the given subset. We can consider this process again and again and will see further fragmentation into busy intervals of jobs that have a higher priority than the root of the subtree.

A.4 The Properties of MCEDF and the Modified MCPI

In this subsection under MCPI will always mean the modified version of MCPI and we assume that both algorithms use the same EDF-compliant support priority table SPT.

The following lemma establishes for MCPI a property that is true for MCEDF by construction.

Lemma A.10. *In MCPI, as in MCEDF, each tree of the P-DAG contains jobs from one and only one busy interval.*

Proof. (sketch) For MCPI, we argue that this property is true by demonstrating that at each basic step of the algorithm: the initial connection of a new job to the P-DAG and the swapping. When a LO-job is connected to a P-DAG, the criterion is to connect it to the trees that block the given job when it has the least priority. Since they block the given job then they must be in the same busy interval. When a HI job is initially connected, the property holds by construction, as we use splitting into busy intervals to evaluate $\overset{\mathbf{J}'}{\sim}$.

Now consider the swapping. After the swapping, the current HI job forms one same busy interval with the subtrees connected to it by the same argument as the ones we used for the initial connection. The LO job which was swapped forms one busy interval with the current HI job tree and other trees that are plugged to it by observation that this was already the case before the swap and the busy intervals do not change when priority assignment changes. □

Lemma A.11 (Per-criticality EDF Compliance of P-DAG)**.** *In the P-DAG G of MCPI(EDF) or MCEDF, consider any P-DAG path between two jobs of the same criticality: $J_i \rhd^* J_j$. This path can only join J_i and J_j in the direction that is compliant with their relative priority in SPT. Mathematically:*

$$\forall i,j \;.\; \chi_i = \chi_j \;\;\wedge\;\; J_i \rhd^* J_j \;\;\Rightarrow\;\; J_i \succ_{SPT} J_j \tag{11}$$

Proof. (sketch) For MCEDF the Property (11) holds by construction, as it requires that J_j be the root of a subtree that contains J_i and *MCEDF_PDAG* assigns the least SPT-priority job of a given criticality as the root of the subtree.

For MCPI, as P-DAG construction evolves, the property can only be potentially broken by the swap operations. However, for criticality level HI it is not broken because we never swap two HI jobs. For criticality LO it can be only invalidated if $CanSwap$ never returns 'false' and then 'true' in the same $PullUp$ subroutine call. This is so because the LO jobs are evaluated for swapping in an order compliant with reverse SPT and the stem of swapped jobs forms a chain in the same order as the swapping is done. The job for which $CanSwap$ would return 'false' would stay as predecessor of the current HI job and the job with 'true' would become successor, thus forming a pair of LO jobs connected inconsistently with SPT. However, this cannot happen if SPT is EDF-compliant, as the first 'false' result from $CanSwap$ will be followed by other 'false'. To show this, recall that by Lemma A.10 the HI job forms one busy interval $(\tau_1, \tau_2]$ with its subtree. When $CanSwap$ evaluates different LO jobs for the least priority it evaluates for the possibility that the swapped job can terminate at time τ_2 while meeting its deadline. The jobs are evaluated in reverse EDF order, so the job with the largest deadline will be evaluated first. However, if that job misses its deadline at time τ_2 then the other jobs will fail as well. □

By the above lemma, for MCEDF and MCPI(EDF), it is always possible to find a topological sort of graph G such that the resulting priority table satisfies the following property:

Definition 9 (HI-criticality EDF Compliance of Priority Table)**.** *Given an EDF-compliant SPT priority table, any LO-mode priority table PT is said to be HI-criticality EDF-compliant according to table SPT if the HI jobs appear in PT in the same order as in SPT, that is:*

$$\forall i,j \;.\; \chi_i = \chi_j = \mathit{HI} \;\wedge\; J_i \succ_{PT} J_j \;\;\Rightarrow\;\; J_i \succ_{SPT} J_j \;\wedge\; D_i \leq D_j$$

Consider a problem instance $\mathbf{J}$ where h jobs are HI-critical. We can partition an EDF-compliant priority table generated by MCEDF/MCPI(EDF) into the following sequence of job sets:

$$PT \ : \ \mathbf{J}_1^{\text{LO}} \succ_{PT} \{J_1^{\text{HI}}\} \succ_{PT} \mathbf{J}_2^{\text{LO}} \succ_{PT} \{J_2^{\text{HI}}\} \succ_{PT} \ldots \ \mathbf{J}_h^{\text{LO}} \succ_{PT} \{J_h^{\text{HI}}\} \succ_{PT} \mathbf{J}_{h+1}^{\text{LO}} \tag{12a}$$

$$\text{HI jobs} \ : \ J_1^{\text{HI}} \succ_{SPT} J_2^{\text{HI}} \succ_{SPT} \ldots J_{h-1}^{\text{HI}} \succ_{SPT} J_h^{\text{HI}} \tag{12b}$$

where subscript l and h denote LO and HI jobs and relation '$\succ$' between two job sets means that any job in the first set has a higher priority than any job in the second set.

Let us denote by $\rhd^{*\text{LO}}$ a relation between two jobs that are joined in the P-DAG by a path that may have only LO jobs as intermediate nodes. The following is trivial:

Lemma A.12. *There always exists a priority table PT obtained from a topological sort of P-DAG G of MCEDF or MCPI(EDF) that has the structure shown in Formulas (12) where, in addition, the LO job sets $\mathbf{J}_i^{LO}$ are defined as the sets of LO jobs related to J_i^{HI} by $\rhd^{*LO}$:*

$$\text{for } \ i = 1..h \ . \ \mathbf{J}_i^{LO} = \{J_j \mid \chi_j = LO \ \wedge \ J_j \rhd^{*LO} J_i^{HI}\} \tag{13a}$$

$$\mathbf{J}_{h+1}^{LO} = \{J_j \mid \chi_j = LO \ \wedge \ \nexists i : J_j \rhd^{*LO} J_i^{HI}\} \tag{13b}$$

Definition 10 (A Least LO-Priority Table)**.** *Given a P-DAG G that is per-criticality compliant to support priority table SPT. A priority table obtained from graph G that can be partitioned as shown in Formulas (12) and (13) is called a least LO-priority table.*

The reason to give a priority table this name is that such a table puts each LO job at the highest-i (and hence also the least-priority) set $\mathbf{J}_i^{\text{LO}}$. The following lemma states that one cannot give any LO job even less priority *w.r.t.* a HI job.

Lemma A.13. *Let $\mathbf{J}$ be a problem instance where MCEDF or MCPI(EDF) generates a P-DAG based on an EDF-compliant SPT, let $\mathbf{J}_i^{LO}$ characterize its least LO-priority table. Let PT' be a HI-criticality SPT-compliant priority table where some LO jobs in some job sets $\mathbf{J}_i^{LO}$ 'violate the least LO priority constraint' in the sense that they have a less priority than the corresponding HI job J_i^{HI}. Then at least one of such jobs will miss its deadline.*

Proof. Let i' be the smallest-index i of the job sets $\mathbf{J}_i^{\text{LO}}$ that contain 'violating' LO jobs in the sense defined the in lemma conditions. Let J_j be the least-priority violating job from that set. Let us show that it will miss its deadline. The part of the priority table PT' that contains jobs of priority higher or equal to J_j can be represented by (dropping the curly braces for singleton sets):

$$PT' \mid_{\succeq j} \ : \ \mathbf{J'}_1 \succ J_1^{\text{HI}} \succ \ldots \succ \mathbf{J'}_{i'-1} \succ J_{i'-1}^{\text{HI}} \succ \mathbf{J'}_{i'} \succ J_{i'}^{\text{HI}} \succ \mathbf{J''}_{i'} \succ J_j$$

Observing that in single processor scheduling the relative priority order of higher-priority jobs does not matter for the least priority job, let us reorder the priority of the last HI job and obtain table PT'' that results in *equal* termination time for job J_j:

$$PT'' \ : \ \mathbf{J'}_1 \succ J_1^{\text{HI}} \succ \ldots \succ \mathbf{J'}_{i'-1} \succ J_{i'-1}^{\text{HI}} \succ \mathbf{J'}_{i'} \succ \mathbf{J''}_{i'} \succ J_{i'}^{\text{HI}} \succ J_j$$

From the definition of violating jobs and from the assumption that the sets $\mathbf{J}_m^{\text{LO}}$ for $m \leq i' - 1$ contain no violating jobs (i' being the lowest 'violating' index) we have:

$$\text{for } 1 \leq m \leq i' - 1 \ \ : \ \ \mathbf{J}_m^{\text{LO}} \subseteq \mathbf{J'}_m$$

Also because, by our assumptions, J_j is the least priority violating job in set i' we have:

$$\mathbf{J}_{i'}^{\text{LO}} \subseteq (\mathbf{J'}_{i'} \cup \mathbf{J''}_{i'} \cup \{J_j\})$$

By the job-set inclusion relation above, the following priority table PT''' when compared to PT'' has at most the same but possibly *less* jobs of higher-priority than J_j:

$$PT''' \ : \ \mathbf{J}_1^{\text{LO}} \succ J_1^{\text{HI}} \succ \ldots \succ \mathbf{J}_{i'-1}^{\text{LO}} \succ J_{i'-1}^{\text{HI}} \succ (\mathbf{J}_{i'}^{\text{LO}} \setminus J_j) \succ J_{i'}^{\text{HI}} \succ J_j$$

By properties of MCEDF resp. MCPI(EDF) we have that job $J_{i'}^{\text{HI}}$ forms one busy interval BI with the higher subtrees connected to it and by observation that $J_j \rhd^{*\text{LO}} J_{i'}^{\text{HI}}$ also belongs to the same busy interval BI. Now observe that the reason why MCEDF resp. MCPI(EDF) assigned $J_{i'}^{\text{HI}}$ the least priority in the given BI is because the highest-deadline LO job belonging to the same interval would miss the deadline. J_j, by construction, cannot have a higher deadline, so it should also miss its deadline as the least-priority job in BI. Therefore it will also miss its deadline in PT''', and hence also in PT'' and PT'. □

We can now prove the following:

Theorem A.14. *For a given EDF-compliant SPT, MCEDF and MCPI(EDF) are optimal among the FPM algorithms that are HI-criticality EDF-compliant according to SPT.*

Proof. (sketch) First, note that if an instance $\mathbf{J}$ is MC-Schedulable, then the MCEDF and MCPI(EDF) algorithms will never fail in LO mode and produce a P-DAG that conforms to the lemma's and properties presented in this section. This is so because, firstly, both algorithms are based on iterative improvement of an EDF table, which is optimal in the LO mode. Secondly, at every improvement step the LO-schedulability of the problem instance is preserved as an invariant. This leads to two important conclusions:

1. The only possible schedulability failure that MCEDF or MCPI(EDF) can have is when a HI job that misses its deadline in a HI scenario.
2. For MC-schedulable instance, even if we see the worst case presented in point 1, both algorithms manage to construct a LO-schedulable P-DAG that satisfies all lemma's and properties presented in this section.

Consider an instance $\mathbf{J}$ with h HI jobs. Suppose by contradiction that MCEDF resp. MCPI(EDF) fail to produce a feasible schedule due to a failure in a HI scenario, whereas the optimal algorithm can. By lemma's above, we can present the solution of both algorithms as shown in Formulas (12) and (13).

The theorem conditions require that priority tables be HI-criticality EDF-compatible according to SPT, so the optimal priority table PT' can also be presented in a similar form:

$$PT' \ : \ \mathbf{J'}_1 \succ \{J_1^{\text{HI}}\} \succ \mathbf{J'}_2 \succ \{J_2^{\text{HI}}\} \succ \ldots \ \mathbf{J'}_h \succ \{J_h^{\text{HI}}\} \succ \mathbf{J'}_{h+1}$$

By Lemma A.13 we should have:

$$\text{for } 1 \leq m \leq h \ : \ \bigcup_{i=1}^{m} \mathbf{J}_i^{\text{LO}} \subseteq \bigcup_{i=1}^{m} \mathbf{J'}_i$$

where $\mathbf{J}_m^{\text{LO}}$ are the least LO-priority job sets of MCEDF resp. MCPI(EDF).

This means that, compared to MCEDF or MCPI(EDF), for every HI job J_m^{HI} the optimal algorithm puts at least the same but possibly a larger set of jobs as higher-priority *w.r.t.* to J_m^{HI}. On a single processor this can only decrease the progress made by each HI jobs up to any given point in time in the LO mode. Therefore, after a mode switch, all the HI jobs in the optimal algorithm will have at least the same or possibly more workload to complete than in MCEDF or MCPI(EDF). Therefore, if the latter would fail in some HI scenario all the more so the former would also fail in the same scenario, therefore the optimal algorithm would fail and thus we have a contradiction. □

The next theorem follows as a corollary of Theorem A.14:

Theorem A.15. *When using the same EDF-compatible SPT table MCEDF and MCPI(EDF) are equivalent.*

Note that, despite equivalence, MCEDF has a lower computational complexity, $O(k^2 \log k)$, than MCPI, which has $O(k^3 \log k)$ (where k is the number of jobs). Intuitively, this is so because MCEDF is 'specialized' for the single-processor scheduling problems, which is inherently 'simpler' than the multi-processor ones, handled by MCPI.