Priority-based scheduling of mixed-critical jobs

Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga

Noname manuscript No.
(will be inserted by the editor)

Priority-Based Scheduling of Mixed-Critical Jobs

Dario Socci[2] · **Peter Poplavko**[2] · **Saddek Bensalem**[1] · **Marius Bozga**[1]

the date of receipt and acceptance should be inserted later

Abstract Modern real-time systems tend to be *mixed-critical*, in the sense that they integrate on the same computational platform applications at different levels of criticality (*e.g.,* safety critical and mission critical). Scheduling of such systems is a popular topic in literature due to the complexity and importance of the problem. In this paper we propose two algorithms for job scheduling in mixed critical systems: Mixed Criticality Earliest Deadline First (MCEDF) and Mixed Critical Priority Improvement (MCPI). MCEDF is a single processor algorithm that theoretically dominates state-of-the-art fixed-priority algorithm Own Criticality Based Priority (OCBP), while having a better computational complexity. The dominance is achieved by profiting from a common extension of fixed-priority online policy to mixed criticality. MCPI is a multiprocessor algorithm that supports dependency constraints. Experiments show good schedulability results. Also we formally prove that both MCEDF and MCPI are optimal in a particular class of algorithms.

1 Introduction

Safety critical real-time systems are subject to certification, *i.e.,* they need to assure high dependability constraints. Modern standards [Joh92] define different levels of criticality that need to be certified at different level of assurance. When a system has applications at different criticality levels sharing the same computational platform, it is called mixed-critical. While designing such systems it is crucial to find a good balance between integration and isolation. On one hand, the integration of several functions using a common set of resources gives the advantage of reduced cost, weight and power consumption, which is important for many embedded systems. On the other hand, it leads to major complications in system design.

Among other aspects, the real-time scheduling of certifiable mixed critical systems has been recognized to be a challenging problem. Traditional techniques require complete isolation between criticality levels or global certification to the highest level of assurance, which leads to resource waste, thus loosing the advantage of integration. This led to a novel wave of research in the real-time community, and many solutions have been proposed. Among those, one of the most successful ideas is adapting Audsley approach to mixed criticality, which was for the first time done by S. Vestal in [Ves07] and later on by others, notably, [BLS10]. However this method has some limitations, which become more pronounced in the case of multiprocessor scheduling. For this reason the scheduling algorithms for multiprocessor mixed-critical systems are not as numerous in literature as those for single processor, and usually they are built on restrictive assumptions.

Research supported by the European ICT Project no. 288175 (CERTAINTY) and no. 332987 (ARROWHEAD).

1. Université Grenoble Alpes (UGA), VERIMAG, F-38000 Grenoble, France
2. Mentor® A Siemens Business, F-38334 Inovallée Montbonnot, France
(The presented research was done while still working at UGA/VERIMAG.)
E-mail: {Dario_Socci | Petro_Poplavko}@mentor.com {Saddek.Bensalem | Marius.Bozga}@univ-grenoble-alpes.fr

To tackle the complexity problem, we assume a fixed set of jobs as workload model. This model can represent a hyperperiod of synchronous periodic tasks or servers, removing some fundamental difficulties of non-synchronous systems (at risk to increase costs in some cases). Fixed job sets allow us to manipulate their priorities by using the novel concept of priority graph (P-DAG), which defines the minimal relation between the priorities in a schedule. Based on this formalism we propose two priority based algorithms. The first algorithm, Mixed Criticality Earliest Deadline First (MCEDF)[1], is a single processor algorithm that dominates state-of-the-art Audsley approach based algorithm *Own Criticality Based Priority* (OCBP). The second one is a multiprocessor algorithm, Mixed Criticality Priority Improvement (MCPI), that, given a global fixed-priority assignment for jobs, can modify it in order to iteratively improve its schedulability for mixed-criticality setting. Our experiments show an increase of schedulable instances up to a maximum of 30% if compared to well-established solutions for this category of scheduling problems. In addition we show that for single-processor problems MCEDF and MCPI are equivalent and optimal in an important class of algorithms, whereas MCPI is applicable also in the case of multi-processor problems with precedence constraints, and MCEDF has better computational complexity.

This paper is structured as follows. Section 2 introduces the scheduling model. In Section 3 we discuss related work. Then Section 4 introduces the Priority-DAG, a theoretical tool that is useful to understand how jobs interact with each other, and upon which all the algorithms presented in this paper are based. In Section 5 we present MCEDF and in Section 6 MCPI. Section 7 shows some common properties of the two algorithms. Finally in Section 8 we show experimental evaluation of the scheduling algorithms.

2 Scheduling Model

Firstly we introduce the scheduling problem for the case where the different jobs executed on the platform are independent. Afterwards, we extend the focus from this *independent-jobs case* to a possibly non-empty inter-job *precedence relation*.

2.1 Independent Jobs

The use of computing systems in life-critical applications such as avionics or automotive usually requires very high reliability and responsiveness. Most tasks in these applications have timing constraints, *i.e.,* deadlines, to be satisfied. Generally, real-time tasks are categorized as *periodic* and *sporadic*. In fact, a *job* is a single instance of a task execution. Periodic tasks are activated repeatedly in a fixed time interval, called period. Such tasks are usually poll sensors and execute control-loop subroutines. The activation of sporadic tasks can occur at any time (with certain limitations) and they are usually triggered by special conditions or operator command. Both type of tasks generate infinite number of jobs. However in this paper we will mainly focus on a finite set of jobs. This is motivated by the fact that mixed critical scheduling is a relatively new problem in research, and thus, even under this simplifying assumption, there is fertile ground for new research. Note that a set of periodic tasks can be easily modeled as a finite set of jobs, considering only the jobs appearing in one hyperperiod, *i.e.,* the least common multiple of the task periods. This modeling can, with limitations, be applied to sporadic tasks, using periodic servers.

In this section we introduce a formalization of the Vestal model [BBD+12b, Ves07] for Mixed-Critical System (*MCS*). Following a common trend [BBD+12b, BF11], in this paper we will focus on dual-criticality systems, *i.e.,* systems that have only two levels of criticality, the high level, being denoted as 'HI', and the low (normal) level, denoted as 'LO'. Every job gets a pair of WCET values: the LO WCET and the HI WCET. The former one is for normal safety assurance, used to assess the sharing of processor with the LO jobs, and the other one, a higher value, is used to ensure

[1] though we use this legacy name, we should admit that there are several other mixed criticality algorithms that may be considered more intimately related to EDF

certification [Ves07]. One important remark is that both HI and LO jobs are hard real-time, so both *must* terminate their executions before the deadlines. But only HI jobs undergo certification. This means that the designer is confident that the jobs will never exceed their LO WCET. However, it is required to prove to the certification authorities that the HI jobs will meet the deadlines even under the unlikely event that some jobs would execute at their HI WCET.

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

The index j is technically necessary to distinguish between jobs with the same parameters. The timing parameters A_j, D_j, C_j are integers that correspond to time resolution units (*e.g.,* clock cycles). We assume that [BBD+12b]:

$$C_j(\text{LO}) \leq C_j(\text{HI})$$

which makes sense, since $C_j(\text{HI})$ is a more pessimistic estimation of the WCET than $C_j(\text{LO})$. We also assume that the LO jobs are forced to terminate after $C_j(\text{LO})$ time units of execution, so:

$$(\chi_j = \text{LO}) \Rightarrow C_j(\text{LO}) = C_j(\text{HI})$$

An *instance* of the scheduling problem is a set of jobs $\mathbf{J}$. A *scenario* of an instance $\mathbf{J}$ is a vector of execution times of all jobs: $c = (c_1, c_2, \ldots, c_K)$, where K is the number of jobs. If at least one c_j exceeds $C_j(\text{HI})$, the scenario is called *erroneous*. The *criticality of scenario* $c = (c_1, c_2, \ldots, c_K)$ is LO if $c_j \leq C_j(\text{LO}), \quad \forall j \in [1, K]$, is HI otherwise. A scenario c is *basic* if:

$$\forall j = 1, \ldots, K \quad c_j = C_j(\text{LO}) \vee c_j = C_j(\text{HI})$$

A *schedule* $\mathcal{S}$ of a given scenario c is a mapping:

$$\mathcal{S} : T \mapsto \widehat{\mathbf{J}}_m$$

where T is the physical time and $\widehat{\mathbf{J}}_m$ is the family of subsets of $\mathbf{J}$ that contains all subsets $\mathbf{J}'$ of $\mathbf{J}$ such that $|\mathbf{J}'| \leq m$. Every job J_j should start at time A_j or later and run for no more than c_j time units. Note that in this definition we do not include the mapping of jobs to processors, but a valid mapping, if needed, can be easily obtained from a simulation which assumes that jobs may migrate from processor to processor if necessary.We assume that the schedule is *preemptive* and that job migration is possible, *i.e.,* that any job run can be interrupted and resumed later on the same or different processor.

A job J is said to be *ready* at time t iff:

1. it is already arrived at time t
2. it is not yet terminated at time t

The online state of a run-time scheduler at every time instance consists of the set of terminated 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 scheduling policy is *feasible* for the given problem instance if the following conditions are met:

Condition 1 *If all jobs run at most for their LO WCET, then both high-critical (HI) and low-critical (LO) jobs must terminate before their deadlines.*

Condition 2 *If at least one job runs for more than its LO WCET, then all high-critical (HI) jobs must terminate before their deadlines, whereas low-critical (LO) jobs may be even dropped.*

An instance **J** is *clairvoyantly schedulable* if for each non-erroneous scenario, when it is known in advance (hence *clairvoyantly*), one can specify a feasible schedule. This property is of purely theoretical interest, as in reality the execution time of every job J_j is only discovered when J_j signals its termination. Hence, whether the LO job timely termination is required is not known as long as no HI job has shown an execution time exceeding its $C_j(\text{LO})$.

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 instance **J** if for each non-erroneous scenario it generates a feasible schedule. A *mode-switched* scheduling policy uses χ_{mode} in the scheduling decisions, *e.g.,* to drop the LO jobs, otherwise it is *mode-agnostic*. A policy is said to be *work-conserving* if it never idles the processor if there is pending workload.

An instance **J** is *MC-schedulable* if there exists a correct scheduling policy for it.

2.1.1 Correctness and Sustainability

To verify the correctness of a scheduling policy one usually tests it for the scenario with maximal execution times for all jobs, which in our case corresponds to HI WCET's. However, to justify this test a scheduling policy must be *sustainable*, which means that reducing execution time of any job A (while keeping all other execution times the same) should not delay the termination of any other job 'B' [BB06]. In other words, sustainability means that the termination times must be *monotonically non-decreasing* functions of execution times.

For mixed-critical scheduling the sustainability requirement is too restrictive, as it does not take into account that an increase of an execution time of a HI job to a level that exceeds its LO WCET may lead to a mode switch and hence to dropping the LO jobs, which, in turn may lead to an earlier termination of another HI job, and hence non-monotonic dependency of termination times. Therefore, a weaker property is adopted in this case, which we call *sustainable per mode*. This property poses almost the same requirement of non-increasing termination time of a job B, but now it is not required anymore to hold under *arbitrary* execution time reduction of a job A. Now the non-delayed B is required only if the reduction of A does not lead neither to a change of the mode in which B terminates nor to preventing job A from switching. This implies that either A executed for less than LO WCET both before and after the reduction, or that the reduction keeps the execution time of A higher than its LO WCET.

The sustainability-per-mode is studied in [KPSB18a], where it is introduced as '*weak predictability*'. Predictability is the special case of sustainability when the correctness test is based on simulation of a given set of jobs for the worst-case scenario. In this paper we prefer the term sustainability.

The generalization of sustainable policies to sustainable-per-mode ones raises the problem of how to test the correctness of such policies, as we cannot anymore rely on the traditional method and just test the scheduling using one maximal scenario. It turns out that in this case it suffices to test the scheduling policies for $H + 1$ basic scenarios, where H is the total count of HI jobs in the problem instance.

Consider a LO basic scenario schedule $\mathcal{S}^{LO}$ and select an arbitrary HI job J_h. Let us modify this schedule by assuming that at time t_h when job J_h reaches its LO WCET ($C_h(\text{LO})$) it has not yet signalled its termination, thus provoking a mode switch. Then, by Condition 2, we should ensure that J_h and all the other HI jobs that did not terminate strictly before time t_h will meet their deadlines even when continuing to execute until their maximal execution time – the HI WCET. Note that in multiprocessor scheduling multiple jobs may also terminate *exactly* at time t_h in $\mathcal{S}^{LO}$, and they are conservatively assumed to also continue their execution after time t_h in the modified schedule. The behavior described above is formalized to a basic scenario where all HI jobs that execute after time t_h have HI WCET.

Definition 1 (Job-specific Basic Scenario) For a given problem instance, LO basic-scenario schedule S^{LO} and HI job J_h, the basic scenario defined above is called 'specific' for job J_h and is denoted $HI\text{-}J_h$, whereas its schedule is denoted $\mathcal{S}^{HI\text{-}J_h}$.

Note that $\mathcal{S}^{HI\text{-}J_h}$ coincides with $\mathcal{S}^{LO}$ up to the time when job J_h switches, and after the switching time it starts using HI execution times for the jobs that did not terminate before the switch.

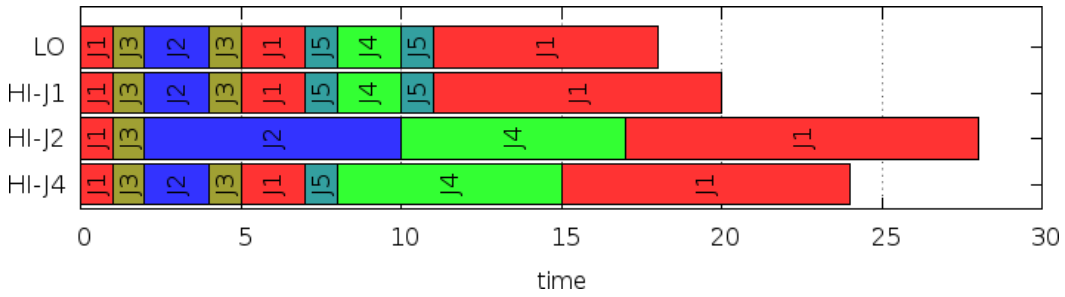


Fig. 1 The Gantt charts for Example 6 with $PT = (2, 4, 3, 5, 1)$

Example 2 Fig. 1 shows Gantt charts for the job-specific scenarios of the following single-processor problem instance:

Job	A	D	χ	C(LO)	C(HI)
1	0	30	HI	10	12
2	2	10	HI	2	8
3	1	8	LO	2	2
4	8	17	HI	2	7
5	7	11	LO	2	2

We see, for example that in the LO scenario job J_2 terminates at time 4, but in the HI-J_2 scenario job J_2 switches at time 4 and continues to execute, because, apparently, it has a HI WCET larger than the LO WCET. In fact, these schedules are obtained for a particular scheduling policy and demonstrate that this policy is correct for the given problem instance, as explained later in Example 6.

Theorem 3 (Correctness Verification by Job-specific Scenarios) *Suppose that the HI jobs satisfy the following (quite general) requirement:* $C_{HI} > C_{LO}$, i.e., *we exclude the possibility of* C_{HI}=C_{LO}.

Then to ensure correctness of a scheduling policy that is sustainable per mode it is enough to test it for the LO scenario and the scenarios HI-J_h *of all HI jobs* J_h.

Proof See [KPSB18a].

□

This theorem holds not only on single processor but also on multiple processors, and not only for independent but also for precedence-constrained jobs, under the requirements that the policy be sustainable per mode and that the HI jobs have strictly larger execution times in the HI mode. The latter requirement is introduced in order to avoid certain 'anomalies' that may appear if it is not satisfied, as shown in [KPSB18b].

Unfortunately, as we will see in the next section, priority-based policies that are sustainable in the ordinary scheduling may loose this property when generalised to mode-switched policies for mixed criticality. Still, priority-based policies are very attractive, as they are well studied and thus offer various useful heuristics for generating efficient schedules. Fortunately, there is an important observation that extends the utility of Theorem 3 to policies that are not sustainable per mode. Lemma 1 in [BBD+12b] proposes a useful transformation of any scheduling policy to a new policy which preserves correctness in the basic scenarios, while, by construction, ensuring sustainability. We call the new policy thus obtained the '*sustainability replacement*' of the original one. The sustainability replacement can be introduced as follows. The original policy is transformed into time-triggered policy with mode switching, which we call *Static Time-Triggered per Basic Scenario* (STTBS). This policy specifies a static time-triggered table for the LO and all job-specific HI scenarios. The policy switches the table upon a mixed-criticality mode switch. The Gantt charts in Figure 1 give, in fact, an example of a complete set of such static tables for the given problem instance. The tables are obtained by simulation of the original policy in the respective scenarios. The execution of STTBS policy starts in the LO static table. If a job finishes earlier than the allocated time, the processors are idled in the

remaining slots of that job. If while executing in the LO static table a HI job switches the criticality mode then the STTBS policy switches to the static table specific for that job.

In our algorithms, we use the sustainability replacement as online policy only in the case of multiple processors – *i.e.,* for the MCPI algorithm. For single-processor case, *i.e.,* for the MCEDF algorithm, this is not required, as we will see in the next section. It should be noted that the behavior of the replacement policy is identical to the behavior of the original policy whenever the simulated scenario is a basic LO scenario or a HI-job specific scenario. To define the proposed algorithms and analyse their properties we need to consider only those scenarios. Therefore, for simplicity and without loss of correctness, in our presentation we will reason exclusively in terms of the original policy, even in the cases where replacement has to be applied for sustainability. The original policy of both our algorithms is 'fixed priority per mode', introduced in the next section.

2.1.2 Fixed Priority and Fixed Priority per Mode

A *fixed-priority* (FP) scheduling policy is a mode-agnostic policy that can be defined by a priority table PT, which is a K-sized vector specifying all jobs (or, optionally, their indexes) 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 schedules the m highest-priority jobs in PT. Fixed priority is a *work-conserving* policy, which means that it never idles a processor when there is a ready job that is not running on another processor. 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 it is clear from the context to which priority table we are referring to.

We introduce *fixed priority per mode* (FPM), a natural extension of fixed-priority for MCS. FPM is mode-switched policy with two tables: PT_{LO} and PT_{HI}. The former includes all jobs. The latter needs to include only the HI jobs. As long as the current criticality mode χ_{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}. Suppose that after removing the LO jobs from PT_{LO} while keeping the same relative order of the HI jobs we obtain the PT_{HI} table. In this case one can just keep using the same priority table, PT_{LO}, after a switch to the HI mode with exactly the same result. Therefore in this particular case we say that we have *FPM-equivalent* tables: '$PT_{\text{LO}} \sim PT_{\text{HI}}$'.

The following [HL94] states a very useful property for which [KPSB18a] proves an important corollary:

Lemma 4 *Fixed-priority policy is sustainable for independent jobs or for single-processor case.*

Corollary 5 *For dual-critical independent-job instances FPM policy is sustainable per mode on single processor under (quite general) Requirement (i): for all HI jobs $C_{HI} > C_{LO}$. It is also sustainable both for single- and multi-processor case under the following (quite restrictive) Requirement (ii): $PT_{LO} \sim PT_{HI}$.*

Though we do not really exploit the general condition – Requirement (i) – in our algorithms, we mention it here to show that sustainability per mode is, in fact, inherent in single-processor FPM scheduling encountered in the literature, *e.g.,* EDF-VD policy [BBD+12a], even if it may modify the relative priorities of HI jobs upon a mode switch. Note that Requirement (i) is, in fact, also present in Theorem 3, and thus it ensures single-processor FPM policies can be tested for correctness using the job-specific basic scenarios. Though in our algorithms we apply the corollary only in independent-job case, we conjecture that this result can be extended to jobs with precedence constraints, because on a single processor the job dependencies can be modeled by priorities.

Unfortunately, in the multiprocessor case, as demonstrated in [KPSB18a], the per-mode sustainability of FPM cannot be asserted under a general condition such as Requirement (i). Moreover, if precedence constraints are present then even in the ordinary non mixed-critical case sustainability does not hold on multiprocessors and Lemma 4 does not apply.

As for Requirement (ii), it is, in fact, inherent property of our proposed algorithms in single-processor case, as we will see later. Though we could have imposed it also for the multiprocessor case, it would, however, impair the quality of the results of the algorithm, as 'good' multiprocessor priority tables should take into account the current job WCETs, and hence also the current criticality mode. Another reason for not having imposed this requirement is that in our multiprocessor algorithm we preferred to support precedence constraints, and then, as explained above, this requirement would not help to ensure sustainability anyway. Therefore, for the multiprocessor and precedence constraints case we assume that the solutions generated in FPM policy for the basic scenarios are executed using the sustainability replacement policy.

Example 6 Consider single-processor independent-job problem instance **J** defined in Example 2. For a given priority table $PT_{\mathrm{LO}} = PT_{\mathrm{HI}}$, the Gantt chart in Figure 1 shows the execution of FPM policy on single processor in all scenarios required by Theorem 3. In the presented scenarios, as the reader may verify, all jobs meet their deadlines. Since for this instance the corollary Requirement (i) holds (by the way, Requirement (ii) holds as well)), FPM is sustainable, and hence Theorem 3 is indeed applicable. Therefore the FPM policy with given priority table is correct for the given problem instance.

If a scheduling policy cannot be defined by a static priority table, it is referred to as a *dynamic-priority* policy. Such policies are, in general, more powerful than the fixed-priority policy, but they are generally more complex and they usually require heavier run-time computations. Also, they are not necessarily sustainable. Fixed-priority scheduling is popular thanks to the fact that it is sustainable and easier to implement. Also, it is natively supported by many operating systems and libraries for programming real-time systems.

Theorem 7 *The schedulability of FPM policy on a single processor (for independent as well as precedence-constrained jobs), under the assumption that it satisfies the requirements for sustainability per mode, can be checked with the same computational complexity as checking a single basic scenario under FP policy, which is $O(K \log K)$ in the independent-job case.*

The above theorem is proved in [KSPB18,SPBB15c,Soc16], where we show that in the specified case checking all job-specific scenarios as in Theorem 3 is not necessary. It is enough to check the basic LO scenario and a specially 'transformed' HI scenario.

2.1.3 Characterization of Problem Instance

To characterize the performance of scheduling algorithms one typically uses 'utilization' and related demand-capacity ratio metrics, *i.e.,* the maximal ratio between demand and capacity of the system. For a job set $\mathbf{J} = \{J_i\}$ and a scenario c the appropriate metric is load [Liu00]:

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

For a multiprocessor system there does not exist a necessary and sufficient schedulability bound on load, whereas it exists for *uniprocessor systems*:

$$\ell oad \le 1$$

For m-processor system the corresponding bound is only *necessary, but not sufficient* [BF05]:

$$\ell oad \le m$$

Baruah *et al.* [LB10] applied the load metric for mixed-critical scheduling with fixed-priority policy, wherefore they defined the LO- and the HI-mode load as shown below:

$$Load_{\mathrm{LO}}(\mathbf{J}) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i:\ t_1 \le A_i \wedge D_i \le t_2} C_i(\mathrm{LO})}{t_2 - t_1}$$

$$Load_{\text{HI}}(\mathbf{J}) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i:\ \chi_i=\text{HI}\ \wedge\ t_1 \le A_i \wedge D_i \le t_2} C_i(\text{HI})}{t_2 - t_1}$$

An instance can only be schedulable if the processors are not overloaded. Hence, a *necessary* condition for MC schedulability is [LB10]:

$$Load_{\text{LO}}(\mathbf{J}) \le m \wedge Load_{\text{HI}}(\mathbf{J}) \le m \tag{1}$$

This is also a sufficient condition for single processor clairvoyant scheduling, but not for practically realizable policies [LB10].

The characterization above proved useful for the fixed-priority policy. Note, however, that a shortcoming of $Load_{\text{LO}}$ and $Load_{\text{HI}}$ is that they ignore a phenomenon which we call the *WCET uncertainty*. This phenomenon makes a practically realizable policy inferior to a clairvoyant scheduler. The latter 'knows for certain' whether and when a mode switch will occur at runtime, whereas an ordinary policy is 'uncertain' about this and may 'learn' it online. By definition, this online information is taken into account only by mode-switched policies. The WCET uncertainty of a job can be measured as:

$$\Delta C_j = C_j(\text{HI}) - C_j(\text{LO}) \tag{2}$$

In [PK11] it is proposed to consider a new set of job deadlines for the LO scenario: $D'_j = D_j - \Delta C_j$. It was noticed in [PK11] that in the LO scenario the jobs should meet deadlines D'_j, otherwise deadlines D_j are missed in a HI scenario. A new metric, $Load_{\text{MIX}}$ is thus defined by modifying $Load_{\text{LO}}$ by substituting D'_j into D_j [PK11]:

$$Load_{\text{MIX}}(\mathbf{J}) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i: t_1 \le A_i \wedge D'_i \le t_2} C_i(\text{LO})}{t_2 - t_1}$$

The *necessary* condition (1) can be refined to the following proposition.

Lemma 8 (Necessary condition for schedulability) *A mixed-critical job set* $\mathbf{J}$ *is schedulable only if*

$$Load_{MIX}(\mathbf{J}) \le m \wedge Load_{HI}(\mathbf{J}) \le m \tag{3}$$

whereby, in addition, for all jobs in LO mode it must hold:

$$A_i + C_i(LO) \le D'_i \tag{4}$$

and HI jobs in HI mode must also respect the following condition:

$$A_i + C_i(HI) \le D_i \tag{5}$$

Note that by default (5) can be implied from (4) by applying the definition of D', but in Section 2.2 we refine this lemma to precedence constraints, whereby the arrival times and deadlines become mode-dependent, which makes it necessary to distinguish (5) from (4).

$Load_{\text{MIX}}$ is a better indicator of schedulability than $Load_{\text{LO}}$. This is shown in Section 5.4, where we introduce *splitting*, a transformation that modifies one instance into another with equal $Load_{\text{LO}}$, but lower $Load_{\text{MIX}}$. The instance thus generated is more likely to be schedulable by FPM policies.

The *Load* metric is a very powerful means to profile single processor problem instances. On multiprocessors, however, its effectiveness reduces. This is due to the fact that in a multiprocessor it is not always possible to use all the available resources for the available workload, due to the fact that apriori we cannot parallelize the execution of a single job. For example, if we consider an instance composed of only one job J_1 such that $C_1 = D_1 - A_1$, then we will have $Load = 1$. If we decrease the speed of the processor by a factor ϵ it will not be possible to schedule this instance. However, if we have, for example, 4 processors, the *Load* metric will tell us in this case that we have only a little bit

more than 1/4 of the resources busy. This shows the weakness of *Load* metric. To compensate this issue, we introduce the *Stress* metric:

$$stress_{\text{LO}}(\mathbf{J}) = \max_{0 \le t_1 < t_2} \left\{ \frac{m}{\min\{|\mathbf{J}'|, m\}} \cdot \frac{\sum_{J_i \in \mathbf{J}'} C_i(\text{LO})}{t_2 - t_1} \right\}$$

where $\mathbf{J}' = \{J_i \mid t_1 \le A_i \wedge D_i \le t_2\}$.

The $m/|\min\{|\mathbf{J}'|, m\}|$ 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. In the example given above, in fact, we will have that $m/|\mathbf{J}'| = 4$, thus giving $stress > 4$, coherently with the non-schedulability of the instance. Metrics $stress_{\text{HI}}$ and $stress_{\text{MIX}}$ are defined in a similar way.

In the context of Lemma 8, one can rewrite the necessary condition (3) using stress, but this would not make the lemma stronger, due to additional conditions (5) and (4). Nevertheless, in general we have $stress \ge \ell oad$, 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.

2.2 Precedence Constraints

2.2.1 Problem Definition

In this section we generalize the previously presented scheduling problem for the case of precedence-constrained jobs.

A *task graph* is $\mathbf{T} = (\mathbf{J}, \rightarrow)$, where is $\mathbf{J}$ is a set of jobs with unique indexes and $\rightarrow \subset \mathbf{J} \times \mathbf{J}$ is a precedence relation. We use the notation $J_a \nrightarrow J_b$ to indicate that there is no precedence relation $J_a \rightarrow J_b$. The precedence relation is well defined iff:

$$J_a \rightarrow^* J_b \Rightarrow J_b \nrightarrow J_a$$

where $\rightarrow^*$ is the transitive closure of $\rightarrow$. The above condition imposes absence of cycles in the precedence relation. An *independent jobs set* $\mathbf{J}$ can be seen as a special case of a task graph $\mathbf{T} = (\mathbf{J}, \emptyset)$.

The criticality of a precedence constraint $J_a \rightarrow J_b$ is HI if $\chi(a) = \chi(b) = \text{HI}$. It is LO otherwise. For each precedence constraint $J_a \rightarrow J_b$, job J_b may not run until J_a terminates. Thus when scheduling a task graph the additional condition for a job to be ready is when all its predecessors have signalled their termination.

The reader may have noticed that we allow precedences from LO to HI jobs. Such a precedence may be unusual and may be considered a bad practice, contrary to safety standards such as DO-178, as it could make safety-critical, and hence highly trusted, functions dangerously dependent on non-critical, and hence less trusted, functions. In [Bar12a, Bar13], it is proposed to 'preprocess' the task graph such that the criticality of LO predecessors of HI jobs is changed to HI. Nevertheless we still do not exclude the possibility of such precedences because in practice it is necessary to permit non-critical functions to provide some data that is of secondary importance (and hence posing no risk of impairing safety) but still useful. Let us consider a hypothetic practical situation illustrated in 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 airplane 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 system switches to HI mode we only wait for data coming from s4.

With the precedence constraints a schedule, to be feasible, needs to satisfy two additional conditions:

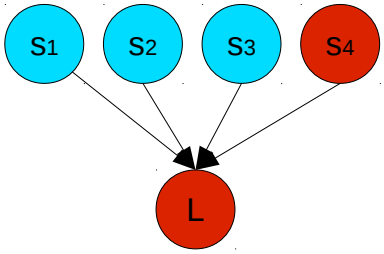


Fig. 2 The graph of an airplane localization system illustrating LO→HI dependencies.

Condition 3 *When the system is in LO mode,* all *precedence constraints must be respected.*

Condition 4 *When the system is in HI mode, HI precedence constraints must be respected whereas LO precedence constraints may be ignored.*

2.2.2 Extending Fixed Priority to Precedence Constraints

To generalize the fixed-priority and FPM policies for precedence constrained problem instances recall that in the previous section we redefined the concept of ready job, by adding the condition that all the predecessors must have terminated. The use of fixed-priority in combination with the adopted precedence-aware definition of ready job is called in literature *List Scheduling*. The generalization of fixed priority is then straightforward, but it is important to note that list scheduling is not sustainable on multiprocessor.

Sustainability is required to reason in terms of basic scenarios. We still use the list scheduling policy offline, but in the case of multiple processors as online policy we use a time-triggered 'sustainability replacement' policy, as explained in Sections 2.1.1 and 2.1.2.

Usually a priority table PT is required to be *precedence compliant*, *i.e.,* the following property must hold:

$$J \to J' \Rightarrow J \succ_{PT} J' \tag{6}$$

The above requirement is reasonable, since we may not schedule a job before its predecessors terminate. It is still possible to schedule using non precedence compliant tables, since the online policy will just ignore high-priority jobs until all their low-priority predecessors signal their termination, but as it will be clear in the next sections, precedence compliance can be convenient to analyze the schedule properties.

2.2.3 Characterization of Problem Instances

In this work we will show how to adapt the metrics introduced in Section 2.1.3 for the case of precedence constrained jobs.

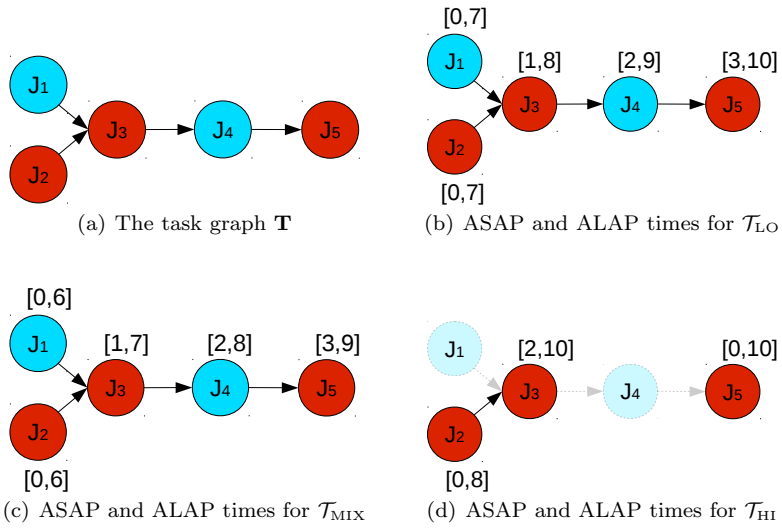
First, we give some preliminaries. A *mode graph* is a graph that considers only single mode of execution (either HI or LO) of $\mathbf{T}$. Formally, a mode graph $\mathcal{T} = (\chi, \mathbf{J}, \to)$ is defined by the criticality level χ of the considered mode, a set of jobs $\mathbf{J}$, and a precedence relation $\to$.

For a problem instance $\mathbf{T} = (\mathbf{J}, \to)$ it is convenient to consider the following mode graphs:

HI-mode graph $\mathcal{T}_{\text{HI}} = (\text{HI}, \mathbf{J}_{\text{HI}}, \to_{\text{HI}})$ considers mode HI and only HI-critical jobs $\mathbf{J}_{\text{HI}}$ with HI-critical precedences between them.

LO-mode graph $\mathcal{T}_{\text{LO}} = (\text{LO}, \mathbf{J}, \to)$ is a mode graph that considers LO mode and all jobs and precedences.

MIX-mode graph $\mathcal{T}_{\text{MIX}} = (\text{LO}, \mathbf{J}_{\text{MIX}}, \to)$ considers LO mode, all precedences, and the jobs $\mathbf{J}_{\text{MIX}}$ obtained from the original jobs $\mathbf{J}$ by modifying their deadlines to the new deadlines D' which are calculated from $D'_i = D_i - (C_i(\text{HI}) - C_i(\text{LO}))$. Note, the same deadlines are used in $Load_{\text{MIX}}$.



(a) The task graph **T**
(b) ASAP and ALAP times for $\mathcal{T}_{\text{LO}}$
(c) ASAP and ALAP times for $\mathcal{T}_{\text{MIX}}$
(d) ASAP and ALAP times for $\mathcal{T}_{\text{HI}}$

Fig. 3 Example of the various mode graphs

We define ASAP arrival times and ALAP deadlines, known in the task graph theory [KA99], but so far mainly used to derive priority tables rather than to compute the load[2].

Given a mode graph with execution times c that correspond to the considered mode, *the ASAP arrival times* A^* are the earliest times when jobs can possibly start:

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

Dually, ALAP deadlines D^* are the latest times when jobs may terminate:

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

By analogy to static timing analysis in digital circuits, ASAP and ALAP values are obtained by "*propagation*" of the arrival and deadline times through the mode graph by longest-path algorithm that calculates the formulas above by visiting the nodes in topological or reverse topological order.

ASAP arrival and ALAP deadlines for the same job are *mode-dependent*. We extend the notion of load and the necessary schedulability conditions in Lemma 8 for precedence-constrained jobs by substituting ASAPs and ALAPs from $\mathcal{T}_{\text{MIX}}$ into the formula for $Load_{\text{MIX}}$ and (4) and from $\mathcal{T}_{\text{HI}}$ into the formula for $Load_{\text{HI}}$ and (5). It is trivial that substituting mode-dependent ASAP arrival time and ALAP deadline to the job parameters does not change the schedulability of the task graph in the given mode, so the necessary conditions in Lemma 8 remain valid, whereas the lemma becomes, in general, stronger. Note that $\mathcal{T}_{\text{MIX}}$ and $\mathcal{T}_{\text{HI}}$ may possibly have different precedence constraints, and, by definition, we use the execution times c of two different modes for them: $C(LO)$ for the MIX-mode graph and $C(HI)$ for HI-mode graph.

An example of ASAP and ALAP times calculation is given in Fig. 3. Fig. 3(a) shows the topology of the task graph. For this graph consider that all jobs have $A = 0$, $D = 10$, $C(\text{LO}) = 1$. For the HI jobs (colored in red in the figures) we have $C(\text{HI}) = 2$, and, thus, for $\mathcal{T}_{\text{MIX}}$ we have $D' = D - (C(\text{HI}) - C(\text{LO})) = 9$. Fig. 3(b) shows ASAP and ALAP times for graph $\mathcal{T}_{\text{LO}}$. The numbers shown in the parenthesis give for each node, respectively, ASAP arrival time and ALAP deadline. In the same way, Figures 3(c) and 3(d) show, respectively, ASAP and ALAP times for $\mathcal{T}_{\text{MIX}}$ and $\mathcal{T}_{\text{HI}}$.

In sequel, unless mentioned otherwise, we assume in the algorithms and analysis that the load and stress values are computed using ASAP and ALAP values from the respective mode graph. Note that for independent-job sets this modification has no effect, because then the ASAPs and ALAPs are equal to the nominal arrivals and deadlines.

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

```
Algorithm:  AudsleyPriorityAssignment
Input:  job set J
Output:  priority table PT
 1: J' ← J
 2: PT ← ∅
 3: while J' ≠ ∅ do
 4:     i ← 1
 5:     JobFound = false
 6:     while i ≤ J'.size() ∧ ¬JobFound do
 7:         TT ← GetTerminationAtLeastPriority(J'[i], J')
 8:         if TT ≤ J'[i].D then
 9:             JobFound ← true
10:             PT ← J'[i] ⌢ PT
11:             J' ← J' \ {J'[i]}
12:         end if
13:         i ← i + 1
14:     end while
15:     if ¬JobFound then
16:         return (FAIL)
17:     end if
18: end while
```

Fig. 4 The Audsley algorithm

3 Related Work

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. (Most often, these offsets are assumed to be zero.) 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). Systems allowing pipelined tasks are known as arbitrary-deadline systems.

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 also consider general real-time policies that are not specifically designed for such systems, as they can be reused in our context as well. We are particularly interested in policies tailored for *global priorities* assigned to individual jobs. "Global" means that the priority controls the whole m-processor system.

3.1 Audsley Approach and its Limitations

A considerable part of the mixed-critical scheduling work in the real time literature uses the priority assignment technique known as "Audsley approach" [Aud93]. The pseudocode of a generic implementation of this technique is shown in Fig. 4. At each step of the external loop the algorithm selects a job to be the least-priority job from the working set of jobs $\mathbf{J'}$, which initially includes all jobs. The job is selected in the internal loop. For this, we pick a job and compute its termination time in the case the job is selected for the least priority. If this time is less than its deadline the job is selected and added in table PT. Then we remove the job from $\mathbf{J'}$ and run a new iteration. If no job can be selected the algorithm fails in finding a solution.

Audsley approach is based on the following assumptions:

1. The termination time of any job does not depend on the jobs having less priority
2. The termination time of any job does not depend on the relative priority of higher-priority jobs

If both assumptions are true, then Audsley approach is optimal. Unfortunately they are not always true in mixed-critical scheduling. Assumption 1 usually holds for preemptive scheduling, while cannot be guaranteed in the case of non-preemptive systems. Assumption 2 holds for single criticality scheduling on single processor. In [BLS10] it is shown that that this assumption also holds for mixed-critical scheduling in the case of the fixed-priority policy on a single processor system. Therefore,

they proposed Audsley approach based algorithm "OCBP" and shown that it is optimal in this case. However, as previously discussed, the FP policy is too restrictive for MCS problems, where FPM is preferred. In the latter case Assumption 2 does not hold since the order of higher-priority HI and LO jobs execution may determine whether the LO jobs get dropped or not in the case of a switch. Also, Assumption 2 does not hold in multiprocessor systems.

Nevertheless Audsley approach is used also in the cases where these assumptions do not hold. In that case the subroutine *AudsleyPriorityAssignment* may not compute the exact termination time, and an upper-bound must be computed instead. In this case, however, the estimation is often too pessimistic and/or computationally intractable.

In the rest of this subsection we illustrate this problem in the case of multiprocessor platforms. An example of an upper-bound calculation for this case is present in the work of [Pat12], where a schedulability analysis is given for sporadic tasks set. Using a similar technique we derive a formula to estimate the termination time in the case of finite job sets and use it to implement the *GetTerminationAtLeastPriority* subroutine of Fig. 4.

Consider a job J_k and assume that all the other jobs $\{J_i\}$ have higher priority: $J_i \succ J_k$. The time interval in which a job J_i may preempt the job J_k is given by:

$$\iota_{i,k} = [A_k, TT_k(\chi)] \cap [A_i, D_i]$$

where $TT_k(\chi)$ is a pessimistic estimation of the termination time of J_k in a scenario of criticality χ. Then the *interference* of J_i on J_k, *i.e.,* the cumulative length of the intervals in which J_i is executing and J_k is ready but not executing, is at most:

$$I_{i,k}(\chi) = \min\{|\iota_{i,k}|, C_i(\chi)\} \tag{7}$$

thus $TT_k(\chi)$ can be estimated by the following formula:

$$TT_k(\chi) = A_k + C_k(\chi) + \left\lfloor \frac{\sum_{i \neq k} I_{i,k}(\chi)}{m} \right\rfloor \tag{8}$$

The last term in the above formula gives a pessimistic bound on maximal time duration where the higher-priority jobs may keep all m processors busy, and it is based on assumption that the load of these jobs is perfectly balanced between the processors.

We realized the algorithm of Fig. 4 implementing *GetTerminationTime* subroutine using the minimal fixed point of Equation (8) to estimate the termination time and we performed some experiments. We randomly generated 181 450 instances of 30 jobs, at different values of $Stress_{\text{LO}}$ and $Stress_{\text{HI}}$, using a method similar to the one described in Section 8.2. The values of *Stress* ranged uniformly from 0 to 2. We tried to schedule each instance first using FPM policy with EDF priority and then with the priority computed by Audsley approach when applying Equation (8) for termination time estimation. Only 4.3% (7804) could be scheduled using Audsley approach, while 56.6% (102690) could be scheduled by EDF.

The weakness of this approach is shown in the following example:

Example 9 Consider the following instance **J**:

Job	A	D	χ	C(LO)	C(HI)
1	0	7	LO	3	3
2	0	8	LO	5	5
3	2	10	LO	5	5

It is easy to check that for $m = 2$ we have the following solutions for Equation (8):

$$TT_1(\text{LO}) = 8, \quad TT_2(\text{LO}) = 9, \quad TT_3(\text{LO}) = 11$$

hence no job can be selected for the least priority. Note that this instance has a low load for a 2-processor instance, in fact $Load_{LO}(\mathbf{J}) = 1.3$. Also note that *any* priority assignment will lead to a correct schedule for this instance.

To understand the weakness of this approach, let us compute the value $TT_1(\mathrm{LO})$. We choose a starting value of $TT_1^0(\mathrm{LO}) = D_1 = 7$. From this we compute $i_{2,1} = [0,7] \cap [0,8] = [0,7]$, thus $I_{2,1} = \min\{7,5\} = 5$. Similarly we have that $I_{3,1} = 5$. Thus, applying Equation (8), we have that $TT_1^1(\mathrm{LO}) = 8$, which is the fixed point for Equation (8). It is clear from the above computations that when the execution windows of jobs are wide, the second term $|\iota_{i,k}|$ of Equation (7) becomes too pessimistic and thus we assume that job J_i runs entirely in the execution window of job J_k.

Though our experiments in Section 8 suggest that even if Audsley approach disposed of practical perfect estimations, it would still be unlikely to significantly improve over our proposed multiprocessor algorithm. Nevertheless, we see that it could become a serious competitor to our algorithm. We consider improving the estimations of fixed-job-set interference bounds on multiprocessors an important subject of future work.

3.2 Multiprocessor Scheduling

Whereas for uniprocessor scheduling a fixed-job-priority algorithm (EDF) is optimal, for multiprocessor case, dynamic job priorities are essential for optimality[DB11]. Moreover, the EDF heuristic can be very inefficient for multiprocessors. In seminal work of Dhall and Liu [DL78] it was shown that the *best*, *i.e.,* maximal load, at which we can be sure to have a schedulable job set for EDF on multiprocessors is no better than the same characteristic for single-processor platforms. 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 [Bar04] it was shown that implicit-deadline sporadic task sets under *global fixed job priority* have the following best guaranteed utilization: $(m+1)/2$. Roughly speaking, this means that such policies can be guaranteed to find a multiprocessor schedule only if the system is loaded by no more than *one half* of its total capacity, and even this is only possible if job priorities are well calculated, *e.g.,* 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 implicit-deadline tasks [Bar04], and later on generalized to arbitrary-deadline tasks under name *EDF-DS*, where DS stands for *density separation* (see [DB11] 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 jobs below the threshold, 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, compared to EDF, schedulability can get even worse using a threshold $th = 1/2$. For finite job sets, our experiments suggest that a higher threshold is better for schedulability.

3.3 Precedence-constrained Scheduling

The *list scheduling* can be seen as generalization of fixed-priority scheduling by handling precedence constraints using *synchronization* between precedence-related jobs, *i.e.,* including the condition of waiting for predecessor termination 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 priority [F$^+$10, Bar12b]. For assigning priorities in precedence-constrained instances, it is generally recognized that the definition of EDF and related heuristics should be generalized to using *ALAP deadlines* D^* instead of the nominal deadlines. For example, the non-preemptive scheduling literature knows so-called 'ALAP' and b-level heuristics [KA99], based on a similar idea. Single-processor scheduling uses this approach for priority assignment with adjusted deadlines [F$^+$10]. Sometimes the ALAP-adjusted EDF is a key ingredient of an optimal strategy, see [KA99] for further references.

3.4 Mixed-critical Scheduling

3.4.1 Single Processor MC scheduling

One of the most notable results in MCS scheduling is OCBP algorithm, which is based on fixed priority. By contrast, the algorithms we propose in this paper, MCPI and MCEDF, are based on a more flexible policy: fixed-priority per mode (FPM). We show that this helps to make them dominant over fixed priority algorithms. To the best of our knowledge, in the previous work no other FPM algorithm [GESY11, BBD+12a, EY12] has been proven to theoretically dominate OCBP. The priority assignment of [GESY11] applies OCBP to compute PT_{LO}, thus having equivalent schedulability. [BBD+12a] proposes an efficient online algorithm with the optimal scaling factor and [EY12] presents a highly efficient priority computation method that dominates OCBP and several other algorithms empirically. Note, however, that [BBD+12a, EY12] are not directly applicable to the problem studied in this paper as they are designed for a sporadic job model with unknown arrival times. Nevertheless, certain very recent non-FPM algorithms theoretically dominate OCBP [Guo16] and for another non-FPM algorithm even a proof of dominance over MCEDF is presented in [BB17].

The FPM policy provides better results than the fixed priority one, but in general dynamic-priority policies are necessary for optimality.

3.4.2 OCBP Single-Processor Algorithm

OCBP algorithm is based on Audsley approach, presented in Section 3.1. It recursively selects the least-priority job J_i using the following criterion: even when having the least priority in the working set, job J_i still meets its deadline in the scenario $(c_k) = (C_k(\chi_i))\ |_{k=1\ldots K}$, *i.e.,* the basic scenario with the WCET at the criticality level χ_i, which is '*own*' for J_i, hence the name of the algorithm. Whether the job J_i fits for the least priority can be checked by fixed-priority scheduling simulation[3] with *any* priorities for the other jobs in the working set provided that they are higher than J_i. The correctness of this check is due to the following lemma [BBD+12b]:

Lemma 10 *On single processor, the termination time of a job J_i in a fixed priority scheduling algorithm depends on the arrival and execution times of jobs J_j with a priority higher than J_i, but not on their relative priority assignment.*

As it will become clear later, this is so because the least priority job terminates at the end of so-called "busy interval", *i.e.,* the interval where the processor is never idle, this interval is determined by jobs' total workload, but not by their priorities. Thus we can compute the exact worst case termination time for the last job in that interval and this is where the optimality of OCBP comes from. Note that Lemma 10 confirms Assumption 2 of Audsley algorithm. Note also that this assumption holds because OCBP does not drop LO jobs even if a HI job exceeds its LO WCET.

3.4.3 Multiprocessor MC Scheduling

Some of the first works made on multiprocessor MC scheduling are based on temporal isolation techniques (Mollison*et al.* [MEA+10], Herman *et al.* [HKM+12]). This approach provides good isolation between criticality levels, but it gives worse performances compared to solutions that allows jobs at different criticality to run concurrently. Li and Baruah [LB12] proposed a global multiprocessor algorithm, fpEDF, with a theoretical analysis of schedulability. This approach, however does not provide good utilization for high number of processors, and it was shown [BCLS14] that partitioned solutions can provide better schedulability guarantees for implicit-deadline tasks.

There are only a few works on precedence-constrained mixed-criticality scheduling. In [Bar13], multiprocessor list scheduling algorithm was proposed. However, it is restricted to jobs that all have the same arrival and deadline times. Finally, [YKRB14] consider pipelined scheduling for task graphs. However, they implicitly assume that the deadlines are large enough, such that they can be ignored

[3] [BLS10] uses a more efficient procedure - *makespan* (see Section 4.4)

Algorithm: *MC-ALGO*
Input: **job instance J**
Output: **priority table** *PT*
1: $SPT \leftarrow JobsOrderedByEDF(\mathbf{J})$;
2: **if** $LOscenarioFailure(SPT, \mathbf{J})$ **then**
3: **return** (FAILURE-NON-SCHEDULABLE)
4: **end if**
5: $PT \leftarrow ImproveHIJobs(SPT, \mathbf{J})$
6: **if** $anyHIscenarioFailure(PT, \mathbf{J})$ **then**
7: **return** (FAILURE-ALGO-CANNOT-SCHEDULE)
8: **end if**

Fig. 5 The algorithm for computing priorities

during the problem solving, as only period (throughput) constraints were considered and not deadline (latency) ones.

3.4.4 Own Work

The present article unifies our results on MCEDF [SPBB13] and MCPI [SPBB15a] and adds a new theoretical study of their common properties, equivalence, and optimality. The formulation of MCPI given here contains some important improvements *e.g.,* potential interference relation. The MCEDF algorithmic complexity presented here is also noticeably improved compared to the original paper.

4 Preliminaries

In this section we will introduce the concepts of Priority DAG (P-DAG), busy intervals and potential interference relation. We use these concepts to reason about how different jobs may interfere with each other when they are executed by a priority-based scheduling policy.

4.1 P-DAG Motivation

Informally a P-DAG is a graph that defines a partial order on the jobs that yields *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 that the jobs execute by default in the basic LO scenario. Recall that a priority table PT defines a total order on the set of jobs $\mathbf{J}$. 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: $PT \vDash_m \mathcal{S}$.

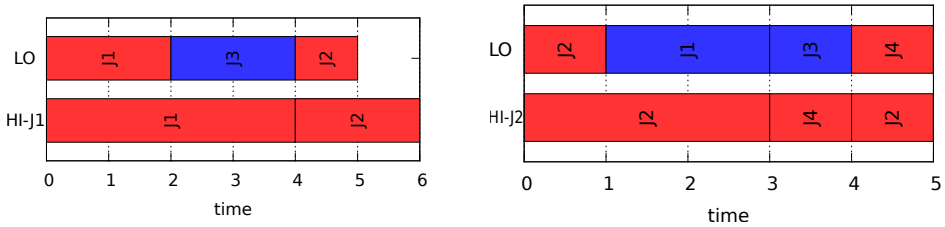
Before defining the concept of P-DAG, we will show in this section the reason why such a structure may be useful. Fig. 5 shows the pseudocode of an algorithm that computes priorities for the mixed critical scheduling problem.

The idea is to start with a good mixed criticality-unaware priority order (in this case EDF), referred to as "support priority table" SPT , and then to improve SPT by raising the priorities of HI-critical jobs. All the priority-based algorithms proposed in this work are based on this template. In terms of schedulability this procedure is constrained by meeting the LO scenario deadlines, postponing the HI scenario checks until the final solution is obtained.

A 'simplistic' implementation of the HI job priority improvement is shown in Fig. 6. This procedure increases the priorities of the HI jobs *w.r.t.* the LO jobs, while the relative priorities between the jobs of the same criticality level, LO or HI, remain in the EDF order. This is done in a manner similar to a bubble-sort (or an insertion sort) in the PT array. We visit the HI jobs in decreasing priority order, and try to raise each HI job ('raising a bubble') by repeatedly swapping priority with the adjacent priority LO job. Subroutine $CanSwap(j, j-1, \ldots)$ simulates the fixed priority schedule PT with entries j and $j-1$ swapped and returns whether all deadlines are met. Subroutine *Swap* performs the actual swapping.

Algorithm: *SimplisticImproveHIJobs*
Input: **priority vector** SPT
Output: **priority vector** PT
1: **for** $i \leftarrow 1$ **to** k **do**
2: $PT[i] \leftarrow SPT[i]$
3: $j \leftarrow i$
4: **if** $PT[j].\chi = \text{HI}$ **then**
5: **while** $j > 1 \wedge PT[j-1].\chi = \text{LO} \wedge CanSwap(j, j-1, \quad PT, \mathbf{J})$ **do**
6: $PT \leftarrow Swap(j, j-1, \quad PT)$
7: $j \leftarrow j - 1$
8: **end while**
9: **end if**
10: **end for**

Fig. 6 Simplistic improvement procedure, keeping the EDF order between same-criticality jobs



(a) The Gantt chart of Example 11 (b) The Gantt chart of Example 12

Fig. 7 Gantt charts for two different examples. The colors indicate the criticality level.

This procedure is illustrated in the following examples:

Example 11 Let $\mathbf{T}$ be the independent-job instance defined by the following table:

Job	A	D	χ	C(LO)	C(HI)
1	0	5	HI	2	4
2	3	6	HI	1	2
3	0	4	LO	2	2

The algorithm in Fig. 5 will first give EDF priorities to the jobs, thus generating the following priority table:

$$PT = (J_3, J_1, J_2)$$

Then the algorithm in Fig. 6 will be called to improve this priority table. First, since J_1 is HI job and J_3 is LO job, it will check if the swap between them is possible by checking if, after being moved to the second PT position, J_3 will still met its deadline in the LO scenario. In this case J_1 will execute for 2 time units, thus terminating at time 2, and then J_3 will execute for other 2 time units, thus terminating at 4. Since there is no deadline miss, we will accept the swap, thus obtaining:

$$PT = (J_1, J_3, J_2)$$

Then the algorithm will try to swap J_2 and J_3, but if it does so J_3 will terminate at time 5, missing its deadline. Since there are no other possible swaps, the algorithm terminates. Fig. 7(a) shows that using this priority order all deadlines are met in all possible scenarios of the instance.

However, this 'simplistic' procedure may easily fail, as shown in the next example:

Example 12 Let $\mathbf{T}$ be the independent-job instance defined by the following table:

Job	A	D	χ	C(LO)	C(HI)
1	0	3	LO	2	2
2	0	6	HI	1	4
3	3	4	LO	1	1
4	3	5	HI	1	1

The algorithm of Fig. 6 will first give EDF priorities to the jobs, thus generating the following priority table:

$$PT = (J_1, J_3, J_4, J_2) \tag{9}$$

The only possible swap here is between jobs J_4 and J_3, but it will lead to a deadline miss of job J_3. The algorithm will therefore leave this priority table unchanged, not having improved any HI-job priority. Thus the algorithm will fail, since, as the reader may check, the priority table of (9) will lead to a non-feasible schedule in scenario HI-J_2. In this case, a correct priority table is:

$$PT = (J_2, J_3, J_4, J_1) \tag{10}$$

and its performance is shown in Fig. 7(b).

The problem encountered in this example can be explained as follows. In (9) we put the jobs in a linear structure in the order of their deadlines and try to swap the neighbors in this structure. However, as we can see in the LO schedule in Fig. 7(b), *temporally* these jobs are located differently, as J_1 is next to J_2, so they are neighbors, even though *deadline-wise*, as in (9), they are not; this is because even although they at the opposite sides on the scale of deadlines they still arrive at the same time. This group of jobs – $\{J_1, J_2\}$ – executes in LO scenario strictly inside interval $(0, 3)$, no matter what their relative priorities are. The other group, $\{J_3, J_4\}$, arrives at time 3 and execute at the other interval: $(3, 5)$. The jobs of different groups do not interfere with each other, therefore we should swap the priorities in the two groups independently. Thus, by swapping J_1 and J_2 in (9) we obtain a correct priority table (10).

To avoid similar problems we introduce the concept of *Priority DAG* (P-DAG), that is, intuitively, a structure that represents how jobs interfere with each other and allows us to "swap" job priorities in a DAG structure instead of a linear chain structure.

4.2 P-DAG Definition and Properties

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

Definition 13 (P-DAG) We call $\mathbf{PT}(G)$ the set of all priority tables that can be obtained by a topological sort of $G(\mathbf{J}, \rhd)$. In other words, we have $J_1 \rhd J_2 \Rightarrow J_1 \succ_{PT} J_2$ for all $PT \in \mathbf{PT}(G)$. We also say that edges '$\rhd$' define relative priority constraints between jobs. G is a P-DAG on m processors for schedule $\mathcal{S}$ iff:

$$\forall PT,\ PT \in \mathbf{PT}(G) \Rightarrow PT \vDash_m \mathcal{S} \tag{11}$$

We indicate the schedule generated by a P-DAG G as $\mathcal{S}(G)$. Two P-DAGs giving the same schedule are called *equivalent*. Formally:

Definition 14 (Equivalent P-DAGs) Two P-DAGs G and G' are equivalent on m processors iff:

$$\mathcal{S}(G) = \mathcal{S}(G')$$

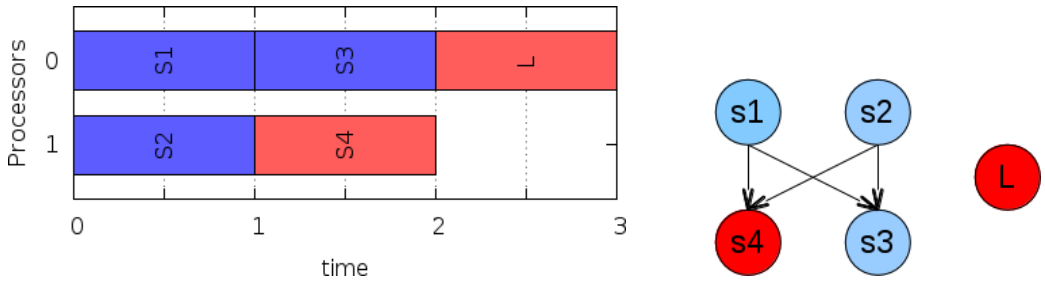
Lemma 15 *A necessary condition for* non *equivalence between two P-DAGs,* $G(\mathbf{J}, \rhd)$ *and* $G'(\mathbf{J}, \rhd')$*, is*

$$\exists J_1, J_2 \mid J_1 \rhd J_2 \wedge J_2 \rhd' J_1 \tag{12}$$

The above comes from the consideration that, to obtain a different schedule, at a certain time t the fixed-priority scheduler must schedule one job instead of another. This may only happen if a pair of jobs has opposite relative priority, so (12) must be true.

Also, the following is trivial:

Lemma 16 *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* $\mathbf{PT}(G') \subseteq \mathbf{PT}(G)$*.*



(a) Schedule (b) Canonical P-DAG

Fig. 8 The figures of Example 20.

Definition 17 (Canonical P-DAG) A *Canonical* P-DAG for a schedule $\mathcal{S}$ is a P-DAG G such that:

$$\forall PT,\ PT \in \mathbf{PT}(G) \Leftrightarrow PT \vDash_m \mathcal{S} \tag{13}$$

Definition 18 (Interference Relation '$\vdash$' between Jobs) Given two jobs J_1 and J_2 and priority table PT, we say that a higher-priority job J_1 *interferes with* a lower-priority job J_2 ($J_1 \vdash_{PT} J_2$) if there is a point in time t where the list scheduler has to select a job to execute on one of m processors from a list of ready jobs where both J_1 and J_2 are present and it selects J_1 whereas J_2 is not selected until a later time.

It's trivial that:

$$J_1 \vdash_{\mathcal{S}} J_2 \Rightarrow J_1 \succ_{PT} J_2 \tag{14}$$

Lemma 19 *Given a task graph* $\mathbf{T} = (\mathbf{J}, \rightarrow)$*, a table* PT *and a number of processors* m*. Consider the* interference *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 (13) holds. Let us first prove that G is actually a P-DAG (*i.e.,* (11) holds). This trivially comes from the observation that during the execution of the schedule $\mathcal{S}$, we only need to compare job priorities when one job interferes with another one. So the relative priority constraints defined by relation $\vdash_{\mathcal{S}}$ are *sufficient* to generate $\mathcal{S}$.

To prove that the priority constraints defined by $\vdash_{\mathcal{S}}$ are also *necessary*, let us suppose by contradiction that there exists a table PT' such that $PT' \vDash_m \mathcal{S}$ and $PT' \notin \mathbf{PT}(G)$. The latter means that $\exists\ J_1, J_2$ such that $J_1 \vdash_{\mathcal{S}} J_2$ and $J_1 \not\succ_{PT'} J_2$. By the first statement and by (14), we have $J_1 \succ_{PT'} J_2$ that contradicts the second statement. □

Example 20 Let us consider the task graph of Fig 2, where $\mathbf{J}$ is defined as follows:

Job	A	D	χ	C(LO)	C(HI)
s1	0	2	LO	1	1
s2	0	2	LO	1	1
s3	0	2	LO	1	1
s4	0	3	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 this PT produces the schedule $\mathcal{S}$ shown in Fig. 8(a). From this figure is easy to derive the interference relation $\vdash_{\mathcal{S}}$. We have: $s_1 \vdash s_3$, $s_2 \vdash s_3$, $s_1 \vdash s_4$, $s_2 \vdash s_4$. Note that L never gets interfered, because, due to precedence constraints, it is never ready until time 2, when all its predecessors terminate. From the interference relation $\vdash_{\mathcal{S}}$, we can derive the canonical P-DAG $G = (\mathbf{J}, \vdash_{\mathcal{S}})$, shown in Fig. 8(b).

The following trivially follows from Lemmas 19 and 16:

Lemma 21 *Consider a task graph* $\mathbf{T} = (\mathbf{J}, \rightarrow)$ *and a graph* $G = (\mathbf{J}, \rhd)$*. Let* $\rhd^*$ *be the transitive closure of* $\rhd$ *and* $\mathcal{S}$ *be a schedule generated by a priority table* $PT \in \mathbf{PT}(G)$*. Then* G *is a P-DAG iff:*

$$\forall J', J'' \in \mathbf{J},\ J' \vdash_{\mathcal{S}} J'' \Rightarrow J' \rhd^* J'' \tag{15}$$

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

Fig. 9 The forest P-DAG generation algorithm

4.3 Forest-shaped P-DAG generation

A P-DAG can, in general, be any kind of DAG. We are interested in generating P-DAGs that are shaped like forests (*i.e.,* a collection of unconnected trees) directed towards the roots. Please note that in this paper by "tree" we mean a directed tree, where all paths from the leafs to the root are directed paths. Also note that some authors use a different terminology by defining a directed tree simply as a DAG whose underlying undirected graph is a tree, and use the term "arborescence" to indicate the DAG that we defined as "tree". The reason why we want such a structure will be clear in the following sections, where we use the properties of forests 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. 9, it takes a task graph and a priority table PT as input. Note that PT must be precedence compliant so that the algorithm visits the task graph in a topological order and may always perform valid simulations, which for every simulated job also simulates all its task-graph predecessors. The algorithm 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 LO-mode basic scenario run of the jobs $\mathbf{J}'$ included so far in G and their precedences, using as priority table a topological sort of G concatenated (by '$\frown$') with J^{Curr} at the least priority position. We assume that the simulation is done by the list scheduling, whereby the precedence edges are defined by the *maximal* subgraph of the task graph, which means that we take not a subset but *all* precedence edges of graph $\mathbf{T}$ that join the jobs that have been included so far. Since the PT is precedence compliant and we included the jobs starting from the highest priority, we can guarantee that after a job is included all its precedence predecessors, the respective precedence edges, and the higher priority jobs are taken into consideration in the subsequent simulation. The simulation keeps track of jobs that interfere with J^{Curr} and returns this information as a relation '$\vdash$'. Then for each tree of forest G that contains a job that interferes with J^{Curr} we add a P-DAG edge going from the tree root to J^{Curr}. The algorithm terminates when all jobs have been handled.

Example 22 Consider the task graph and the priority table of Example 20. Now we construct a *forest* P-DAG for it, using *Forest_PDAG*. In the first step the algorithm picks up s_1, the highest priority job from PT, and adds it to the graph. In the second iteration, we pick up s_2. Since it does not get interfered by any job, we continue without adding any edge. Then we pick up s_3, which gets interfered by both s_1 and s_2, so we add the edges (s_1, s_3) and (s_2, s_3). At the next iteration we pick up job s_4, that also gets interfered by both s_1 and s_2, so we add an edge from the root of the tree that contains the interfering jobs (*i.e.,* s_3) to s_4. In the final iteration we pick up job L, that does not get interfered by any job, thus we add it to the graph without joining any edges to it. The resulting

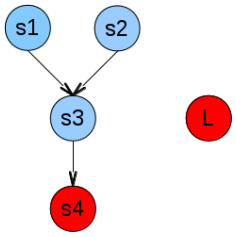


Fig. 10 Forest P-DAG

graph is shown in Fig. 10, which can be compared to the canonical P-DAG for the same example shown in Fig. 8(b).

Theorem 23 *The graph G generated by the* Forest_PDAG *algorithm is a P-DAG and a forest.*

Proof We prove the theorem by induction, by showing that it is true for the subgraph G_n, obtained after n-th iteration of the main loop while assuming the hypothesis that it is true for the subgraph G_{n-1}. By construction, the nodes of G_n are composed of the first n elements of table PT provided at the input. We denote by PT_n the table composed of the first n elements of PT. In fact, G_n is a P-DAG for the schedule obtained from priority table PT_n, *i.e.*, $PT_n \vDash \mathcal{S}(G_n)$ as shown below.

Basic step. The basic step for induction 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 $PT_{n-1} \vDash \mathcal{S}(G_{n-1})$. Also we know that G_{n-1} is a forest. Since we add edges to J_n only from the roots of unrelated subtrees, this operation may only generate another tree, thus G_n is a forest.

G_n is a P-DAG by construction, since the 'for' loop in Fig. 9 ensures property (15) of Lemma 21. Also, since J_n gets no successors in G_n, during a topological sort of G_n we can give J_n the n-th position in the topologically generated 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. 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. Thus $PT_n \vDash \mathcal{S}(G_n)$. □

4.4 Single-processor Job Interference and Busy Intervals

Whereas P-DAGs enable the reasoning about the interference between the jobs whose priorities are known, in our incremental priority calculation algorithms we also need to reason about the subsets of jobs whose priorities are not known or may be modified. In the single processor scheduling the concept of busy intervals is a useful tool for this purpose.

Definition 24 (Busy Interval) Consider work-conserving policy and a task graph $\mathbf{T}$. In the resulting single-processor schedule $\mathcal{S}$, we say that a job J is *active* at the given time if by this time it has arrived, has not yet terminated, and for which it holds that all of its transitive predecessors (*i.e.*, $\{J' \mid J' \rightarrow^* J\}$)} have arrived as well. A *busy interval* (BI) is a maximal open time interval (τ_1, τ_2) where the set of active jobs[4] is never empty.

The BI intervals *do not depend* on the selected work-conserving policy. They depend only on $\mathbf{T}$ and scenario c. In a fixed priority policy neither the start time τ_1 nor the length of a busy interval $\tau_2 - \tau_1$ depends on the exact priority assignment. In fact this is so because the former is given by:

$$\tau_1 = \min_{J_i \in \mathbf{J}_{BI}} \{A_i\}$$

[4] we do not say 'ready' jobs in order to also include jobs that are waiting for their predecessors to terminate

and for the latter we have:

$$\tau_2 - \tau_1 = \sum_{J_i \in \mathbf{J}_{BI}} c_i \tag{16}$$

where $\mathbf{J}_{BI} \subseteq \mathbf{J}$ is the subset of the jobs running in the busy interval.

By abuse of terminology, we apply the term 'busy interval' also to the subset $\mathbf{J}_{BI}$, and denote it BI. In general, a job set $\mathbf{J}$ can be partitioned into multiple busy intervals, because some jobs in $\mathbf{J}$ may arrive at or later than the end of a busy interval of some other jobs in $\mathbf{J}$.

To compute the busy intervals for a given task graph, it is convenient to avoid the case where $J_a \to J_b$ whereas $A_a > A_b$ as an 'irregular case'. For this we 'regularize' the job set.

Definition 25 (Regularized job set) Given a task graph $\mathbf{T} = (\mathbf{J}, \to)$, its *zero-mode graph* is $\mathcal{T}_{LO\text{-}\emptyset} = (LO, \mathbf{J}^{\emptyset}, \to)$ where $\mathbf{J}^{\emptyset}$ is the set of jobs $\{ (A_i, D_i, \chi_i, C_i^{\emptyset}) \}$ obtained from the original set by enforcing zero execution times: $C^{\emptyset} = (0, 0)$. Then the *regularized job set* $\mathbf{J}^*$ is the set of jobs whose arrival times are calculated as ASAP times A_j^* in zero-mode graph $\mathcal{T}_{LO\text{-}\emptyset}$. *i.e.,* using zero execution times and considering the complete set of jobs and precedences.

Definition 26 (Makespan algorithm to compute BIs in LO mode) [5] To compute the busy intervals one can use the *makespan* procedure, as suggested in [BLS10], which we adapt here for precedences. Consider the regularized set of jobs $\mathbf{J}^* = \{J_i^*\}$. Assume that these jobs are ordered by their arrival times (which are, in case of precedences, zero-mode ASAP times). Consider the sequence $f_1, f_2, \ldots, f_i$ of numbers defined by the following recurrence:

$$\begin{aligned} f_1 &= J_1^*.A + J_1^*.C(\mathrm{LO}) \\ f_i &= \max(f_{i-1}, J_i^*.A) + J_i^*.C(\mathrm{LO}) \qquad i > 1 \end{aligned} \tag{17}$$

Then the latest termination time (makespan) of a preemptive work-conserving schedule is given by $F = f_n$. To keep track of all busy intervals, we have to consider that whenever in Equation (17) we have that $f_{i-1} \leq J_i^*.A$ the previous busy interval ends at time f_{i-1} and the new one starts at time $J_i^*.A$.

The following holds trivially:

Lemma 27 (BI computation complexity) *For a task graph* $\mathbf{T}$*, makespan computation has linear complexity in the number of jobs – $O(K)$ – when the jobs are pre-sorted by (regularized) arrival times.*

Lemma 28 (Single-processor schedules are the same with precedence or regularization) *For regularized set* $\mathbf{J}^*$ *the schedule based on a precedence-compliant PT is the same when computed with precedence constraints* $(\mathbf{J}, \to)$ *and when computed for independent jobs* $\mathbf{J}^*$ *are identical.*

The latter lemma holds because in $\mathbf{J}^*$ the satisfaction of precedence constraints is ensured due to interference from higher-priority predecessors.

The following lemma is easy to prove:

Lemma 29 (Least priority in a busy interval) *Given a task graph* $\mathbf{T}$ *and any of its busy intervals BI (τ_1, τ_2). In a list scheduling or regularized fixed-priority scheduling with a precedence-compliant table PT the least-priority job running in the given BI terminates at time τ_2. It either gets interfered by a higher-priority job in BI or gets delayed due to precedence from at least one other job in BI, provided that BI contains some other jobs.*

[5] recall that by default we study LO-mode schedules in this section

4.5 Multiprocessor Case: Potential Interference Relation

In this section we extend the concept of busy intervals for more general multiprocessor case. In fact, in the single processor case two jobs are placed into the same busy interval if and only if they have sufficiently close arrival times such that in some priority-based schedules they may delay each other either by higher precedence or by higher priority. In the general case, we say that such two jobs are in *potential interference relation.*

As indicated in Lemma 28, on a single processor both reasons of one job delaying another can be modeled by interference relation $\vdash$. However, with $m = 2$ and larger, higher priority jobs that run in parallel do not necessarily interfere with a given job. Therefore, to reflect the delaying of a job due to their predecessors, we define the extended interference relation $\overrightarrow{\vdash}$ as follows:

Definition 30 (Extended interference relation) Given a task graph $T(G, \rightarrow)$ and a schedule for it $\mathcal{S}$ we say that $\forall J_1, J_2 \in G, \quad J_1 \overrightarrow{\vdash_{\mathcal{S}}} J_2$ iff $J_1 \vdash_{\mathcal{S}} J_2 \vee (J_1 \rightarrow J_2 \wedge TT_1^{\mathcal{S}} > A_2)$.

Thus in the extended sense also predecessors may interfere with (*i.e.,* delay) a given job if they terminate after the job's arrival.

Definition 31 (Potential Interference Relation) Given a task graph $\mathbf{T}(\mathbf{J}, \rightarrow)$, m processors and a subset $\mathbf{J'} \subseteq \mathbf{J}$ that for each contained job also includes its predecessors, 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:

$$\forall J_1, J_2 \in \mathbf{J'} : \left(J_1 = J_2 \ \vee \ \exists PT : J_1 \overrightarrow{\vdash_{PT}} J_2 \right) \quad \Rightarrow \quad J_1 \overset{\mathbf{J'}}{\sim} J_2$$

whereby we consider LO-mode m-processor list schedules with priority table PT applied to maximal task subgraph with nodes $\mathbf{J'}$.

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 of such relations is the total equivalence. The (unique) minimal potential interference relation can be obtained by union of extended interference relations under all possible PT's, followed by transitive, symmetric, 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 32 (Single-Processor Interference) *In list scheduling a potential interference relation for a single processor is also a potential interference relation for m processors.*

The intuitive meaning of this theorem is that when only one processor is available the 'competition' between the jobs for a processor is strictly larger than when $m > 1$ processors are available.

The following theorem can be derived by induction using Lemma 29:

Theorem 33 (Busy intervals define minimal potential interference relation on single processor) *Given a task graph* $\mathbf{T}$ *and a subset of jobs* $\mathbf{J'}$ *that for each contained job also includes its predecessors. Let* $\mathbf{T'}$ *be the maximal subgraph of* $\mathbf{T}$ *with nodes* $\mathbf{J'}$*. The busy intervals* BI *of sub-instance* $\mathbf{T'}$ *are equivalence classes of the minimal potential interference relation* $\overset{\mathbf{J'}}{\sim}$ *on single processor.*

From this we conclude that computing the minimal potential interference on a single processor can be done in linear time using the makespan procedure explained in Section 4.4. We also conclude that the result of this computation can be used to over-approximate the minimal interference relation on a multiprocessor. We apply these results later in our multiprocessor algorithm.

Algorithm: *MCEDF*
Input: **job instance J**
Output: **priority table** *PT*
1: **if** *LOscenarioFailure*(**J**) **then**
2: **return** (FAIL-NON-SCHEDULABLE)
3: **end if**
4: $G \leftarrow MCEDF\_PDAG(\mathbf{J}, \emptyset, \emptyset)$
5: $PT \leftarrow TopologicalSort(G)$
6: **if** *anyHIscenarioFailure*$(PT, \mathbf{J})$ **then**
7: **return** (FAIL-NON-SCHEDULABLE-BY-MCEDF)
8: **end if**

Fig. 11 The MCEDF algorithm for computing priorities

Algorithm: *MCEDF_PDAG*
Input: **job instance** $\mathbf{J}'$
Input: **node** J^{parent}
In/out: **P-DAG** *G*
1: **BI** $\leftarrow PartitionIntoBIs(\mathbf{J}')$;
2: **for** all $BI \in$ **BI do**
3: $J^{\text{least}} \leftarrow SelectLeastPriorityJob(BI)$
4: $G.\mathbf{J}' \leftarrow G.\mathbf{J}' \cup \{J^{least}\}$
5: **if** $J^{parent} \neq \emptyset$ **then**
6: $G. \triangleright \leftarrow G. \triangleright \cup \{(J^{least}, J^{parent})\}$
7: **end if**
8: $\mathbf{J}'' \leftarrow BI \setminus \{J^{\text{least}}\}$
9: $MCEDF\_PDAG(\mathbf{J}'', J^{least}, G)$
10: **end for**

Fig. 12 The MCEDF algorithm for computing P-DAG

5 Independent-Job Single Processor Scheduling – MCEDF

5.1 Mixed Critical Earliest Deadline First

Our proposed *Mixed-Critical Earliest Deadline First* (MCEDF) algorithm computes priority tables PT_{LO} and PT_{HI} for FPM policy for independent jobs on single-processor platforms. After a switch to the HI mode the scheduling problem becomes a standard non MC problem, for which EDF is optimal in single processor case. So PT_{HI} is assumed to be an EDF table. The problem is then reduced to compute PT_{LO}, which we will call just PT for the rest of this section. The algorithm is formulated here for *independent jobs*, but it can be easily extended to support task graphs.

The algorithm to compute PT is shown in Fig. 11. Initially, we verify the schedulability of the LO scenario in subroutine *LOscenarioFailure*, by running EDF. By optimality of EDF for single criticality level we mean that if a job misses the deadline, then the instance is not schedulable. Thus the algorithm establishes that MC schedulability Condition 1 (see Section 2.1) is satisfied, which remains invariant for the algorithm. The algorithm applies a best-effort heuristic to ensure Condition 2, *i.e.,* that the deadlines of all HI jobs are met also in HI scenarios, by trying to ensure that the priorities of the HI jobs are as high as possible, under the constraint that all jobs meet their deadlines in the LO scenario (*i.e.,* Condition 1 is still satisfied).

To compute the final priority table, MCEDF first calls subroutine *MCEDF_PDAG*, which generates a P-DAG with HI job priorities improved *w.r.t.* the original EDF table. Subroutine *TopologicalSort* employs the well-known topological sort algorithm to obtain the priority table PT from the P-DAG. Finally, the subroutine *anyHIscenarioFailure* evaluates whether Condition 2 is met. In this case the algorithm succeeds. The check could be done by a simulation over the set of job specific HI scenarios $HI\text{-}J_n$ in line with Theorem 3. However, we use a more efficient schedulability check, which does not do exhaustive enumeration of scenarios but instead applies Theorem 7.

The core of the algorithm, *i.e.,* procedure *MCEDF_PDAG* that constructs the P-DAG G, operates only in the LO mode. Therefore in our description of this procedure in the remainder of this section we keep assuming that all jobs execute in the LO basic scenario.

The subroutine *MCEDF_PDAG* is defined in Fig. 12. The algorithm is based on the concept of busy interval, which was defined in Section 4.4. The P-DAG construction algorithm splits the given subinstance $\mathbf{J}'$, $\mathbf{J}' \subseteq \mathbf{J}$ into BI's and selects the least priority job in each BI (see Fig. 12, line 3). Observe that by Lemma 29 in a busy interval (τ_1, τ_2), the selected job will terminate at time τ_2, which can be computed by Equality (16). Let $J^{\text{late}}_{\text{LO}}$ and $J^{\text{late}}_{\text{HI}}$ be the latest deadline [6] job among the LO and the HI jobs of BI respectively. Subroutine *SelectLeastPriorityJob* selects the least priority job according to the following rule.

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

This rule prefers to assign the least priority to $J^{\text{late}}_{\text{LO}}$ if BI has LO jobs and if the 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 if otherwise it would break Condition 1. Let us now show that in a feasible problem instance this rule makes a choice that is feasible for the LO scenario. In uniprocessor scheduling the choice of the least-priority job can adversely affect the schedulability of that job only. Thus, we only need to ensure that this job meets the deadline. The job selected by the described rule can only miss its deadline if the latest-deadline job among all jobs in BI would also miss its deadline, which is only possible in an infeasible instance.

The P-DAG G, as constructed by MCEDF, has multiple subtrees that correspond to the BI's of the complete problem instance $\mathbf{J}$. Subroutine *PartitionIntoBIs* in Fig. 12 splits the currently examined instance into BI's. Then the subroutine *MCEDF_PDAG* examines every busy interval BI to select the least-priority job there. Afterwards 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 tree-children of J^{least} 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.

Example 34 Let the problem instance $\mathbf{J}$ be the same as defined in Example 6:

Job	A	D	χ	C(LO)	C(HI)
1	0	30	HI	10	12
2	2	10	HI	2	8
3	1	8	LO	2	2
4	8	17	HI	2	7
5	7	11	LO	2	2

The priority table illustrated for the same example in Fig. 1 is, in fact, the one computed by MCEDF algorithm: $PT_{MCEDF} = (2, 4, 3, 5, 1)$.

Let us demonstrate how MCEDF computes this table step-by-step. MCEDF starts by checking whether the instance is schedulable in the 'LO' scenario by a simulation with $PT_{EDF} = (3, 2, 5, 4, 1)$. Instead, Fig. 1 row 'LO' shows a simulation for the PT_{MCEDF}; no deadline is missed there, and hence the same should hold for PT_{EDF} as well.

Then MCEDF generates the P-DAG, see the end result in Fig. 13. When constructing the tree roots we observe that there will be only one root, as instance $\mathbf{J}$ has only one busy interval BI, This is visible from the LO-scenario simulation in Fig. 1, where processor remains continuously busy until all jobs terminate, during the interval $(0, 18)$. Thus for $BI = \mathbf{J}$ we should select the overall least-priority job as the tree root. For the considered BI, the latest-deadline jobs are $J^{\text{late}}_{\text{LO}} = J_5$ and $J^{\text{late}}_{\text{HI}} = J_1$. Since $D_5 = 11 < 18$, we cannot select J_5, so we select J_1 as J^{least} for the P-DAG root. Now we split the subinstance $\mathbf{J} \setminus \{J_1\}$ into BI's, obtaining $BI' = \{J_3, J_2\}$, running in $(1, 5)$ (again, see Fig. 1) and $BI'' = \{J_5, J_4\}$, running in $(7, 11)$. For these intervals we select, respectively, J_3 (since $D_3 \geq 5$) and J_5 (since $D_5 \geq 11$). The remaining subinstances have only one job, so the final P-DAG generation steps are trivial (see Fig. 13). Priority table PT_{MCEDF} satisfies the partial order of the resulting P-DAG.

[6] for equal-deadline jobs we break the ties by selecting the job with minimal $C_j(\text{HI}) - C_j(\text{LO})$. This choice is explained in Section 5.2.

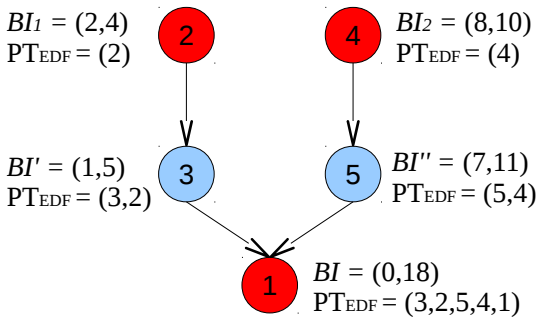


Fig. 13 The P-DAG for Example 34; each node is annotated by the selected job index (colors represent criticality).

Finally the algorithm checks the HI mode schedulability. We can do it by simulating all HI-job specific scenarios (by Theorem teo:correctnessScenarios), as illustrated in rows 'HI-J_j' in Fig. 1. Because, as the reader can verify, the deadlines are met, the algorithm succeeds.

Lemma 35 *The graph G generated by MCEDF_PDAG is a forest and a P-DAG.*

Proof Graph $G = (\mathbf{J}, \triangleright)$ is a forest by construction. To prove that it is a P-DAG, consider two jobs J_1, J_2 such that $J_1 \vdash J_2$ in the final PT obtained from G. This implies that $J_1 \overset{\mathbf{J}}{\sim} J_2$ and hence by Theorem 33 J_1 and J_2 are in the same BI, and thus MCEDF should include them into the same tree in G. Let J^a be their closest common tree ancestor, *i.e.,* the root of a subtree that contains both of them such that: $(J_1 \triangleright^* J^a \vee J_1 = J^a) \quad \wedge \quad (J_2 \triangleright^* J^a \vee J_2 = J^a)$. It may not be that $J^a = J_1$, because then we would have that $J_2 \triangleright^* J_1$, which contradicts $J_1 \vdash J_2$. Hence, we have $J^a \neq J_1$. Suppose that also $J^a \neq J_2$. Then let ST^a be the subtree of G rooted in J^a and $\mathbf{J}^a = ST^a \setminus \{J^a\}$. Since J^a is the closest ancestor of J_1 and J_2, they will not have any ancestor in $\mathbf{J}^a$, thus $J_1 \overset{\mathbf{J}^a}{\nsim} J_2$, which also contradicts $J_1 \vdash J_2$. Thus it may only be that $J^a = J_2$, we thus we have shown that $J_1 \vdash J_2 \Rightarrow J_1 \triangleright^* J_2$ and hence the theorem statement is true by Lemma 21. □

Lemma 36 (MCEDF Complexity) *MCEDF has an implementation with complexity $O(K^2)$ where $K = |\mathbf{J}|$.*

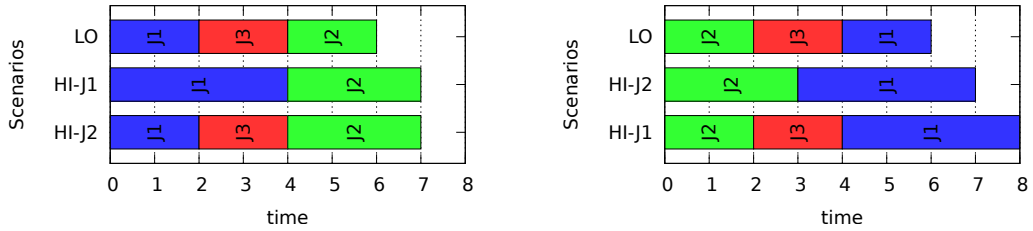
Proof First of all, let us agree that we represent each subinstance $\mathbf{J}'$ by a list that is initially pre-sorted (in time $O(K \log K)$) by arrival times. Note that when splitting subinstances into busy intervals to obtain new subinstances they can stay sorted by arrival times without any additional sorting as they are obtained by simple decomposition of a pre-sorted list into more lists. In *LOscenarioFailure* we perform one fixed-priority schedule simulation. The total cost of one simulation is $O(K \log K)$ (see [SPBB15b]).

Graph G, being a collection of trees, has K nodes and at most $K - 1$ edges. The complexity of *TopologicalSort* for such graphs is $O(K)$ [CLRS01].

We now analyze the complexity of *MCEDF_PDAG*. Let ST^i denote subtree of G rooted at node J_i. The most time-costly procedure at each node J_i of graph G is the partitioning of the current subinstance $\mathbf{J}'_i = ST^i - \{J_i\}$ into busy intervals. Because the subinstance is previously sorted by the arrival times, this can be done in a time linear in $|\mathbf{J}'_i|$ by the makespan procedure (see Lemma 27). Next to splitting into BI's, the other basic procedure at each P-DAG node is the selection of the least-priority job J_i in $\mathbf{J}'_i$, which is also linear in $|\mathbf{J}'_i|$, as a selection of the maximal-deadline job in a list of jobs.

Let us now assign every node in forest G to a tree level based on its distance to the roots. Now observe that all ST^i rooted at the same tree level together contain at most K jobs, with exactly K jobs at the first level and removing some of them when going from the roots to the leafs. So, the tree generation cost is $O(K)$ per level. Because there are at most K tree levels, the total P-DAG generation complexity is $O(K^2)$.

Finally, *anyHIscenarioFailure* can be done in $O(K \log K)$ time according to Theorem 7. □



(a) $PT = J_1 \succ J_3 \succ J_2$ (b) $PT' = J_2 \succ J_3 \succ J_1$

Fig. 14 The Gantt charts of Example 37

5.2 The Support Priority Table for MCEDF

As the candidates for getting the least priority, J_{LO}^{late} and J_{HI}^{late}, the MCEDF assigns the latest-deadline jobs at the given criticality level. MCEDF, however, does not prescribe anything specific to break the tie in case multiple jobs have the same deadline. Certain properties of MCEDF do not depend on how the ties are broken. However, for better schedulability in HI mode and for certain other properties this has to be specified. In this case, we assume that an additional input is provided to the MCEDF algorithm, the so-called *support priority table*, denoted SPT, and that MCEDF selects the least SPT-priority LO and HI jobs as the least-priority candidates. This table must be EDF-compliant:

$$J_1.D < J_2.D \Rightarrow J_1 \succ_{SPT} J_2$$

To construct the SPT table the user may follow certain heuristics, to which we refer as *support algorithm*. In our MCEDF implementation the following algorithm is used. In the case of equal-deadline jobs we break the ties by selecting for the least priority the job with the least WCET uncertainty ΔC_j (see Equality (2)). Our rationale for this stands in the observation that jobs with a high WCET uncertainty are 'more critical', because the quantity of additional computation they add in the case of a switch is higher. We will show the advantage of this heuristics in the following example.

Example 37 Consider the instance **J** defined by the following table:

Job	A	D	χ	C(LO)	C(HI)
1	0	7	HI	2	4
2	0	7	HI	2	3
3	0	4	LO	1	1

In the case of the uncertainty-based SPT MCEDF will generate the following priority table:

$$PT = J_1 \succ J_3 \succ J_2$$

Using this priority table the jobs will meet the deadlines in all scenarios, as shown in Fig. 14(a). In the case we would disambiguate the jobs differently MCEDF will generate the following solution:

$$PT' = J_2 \succ J_3 \succ J_1$$

This table is not schedulable, since J_1 misses the deadline in scenario HI-J_1, as shown in Fig. 14(b).

5.3 Dominance over OCBP

In this subsection we provide a theoretical evidence that MCEDF dominates OCBP. Example 34 has shown an MCEDF-schedulable instance which is, in fact, not OCBP-schedulable. The latter can be shown as follows. Suppose one can select the least OCBP-priority job in this instance. It cannot be a LO job, because, as shown earlier (see Fig. 1), instance **J** consists of a single BI that terminates at time 18, when any LO job would miss its deadline. If we could select a HI job, then OCBP would

evaluate its termination time by effectively extending the aforementioned LO-scenario BI into a longer HI-scenario BI where all HI jobs take $C_j(\text{HI}) - C_j(\text{LO})$ extra time. Summing up these differences, this adds 13 time units to the termination time 18. But the termination time 31 is beyond the latest HI job deadline, $D_1 = 30$.

Thus, the dominance is given by the following result:

Theorem 38 *If an instance is OCBP schedulable, then it is schedulable by the MCEDF algorithm as well, whereas the opposite (as already shown) is not true.*

Proof Recall that, by P-DAG definition, the preference for one particular topological order to derive the PT from the P-DAG generated by MCEDF does not impact its schedulability. Similarly, when OCBP has multiple choices for the selection of the least priority job then preferring a particular choice does not matter for the OCBP schedulability [BLS10]. So, we will show that if one follows certain rules in making a choice in the MCEDF and OCBP, then both algorithms will construct the same priority table PT for any OCBP-schedulable instance $\mathbf{J}$.

Let us first examine in detail how MCEDF constructs PT going in reverse topological direction from less-priority roots of graph G to higher-priority leafs. At each step of the reverse topological sort we select a job and add it to the head of PT. The job is selected from a 'ready set' (RS), *i.e.,* the set of jobs $\{J_i^{\text{RS}}\}$ that are that are roots of subtrees $\{ST_i^{\text{RS}}\}$ which consist of jobs that have not yet been selected and which correspond to busy intervals $\{BI_i^{\text{RS}}\}$. Implicitly, there is a sub-instance $\mathbf{J'}$ of all such jobs, of which BI_i^{RS} are the busy intervals and J_i^{RS} are their respective J^{least} jobs. MCEDF select the J^{least} job of any of those BIs as the least-priority job in sub-instance $\mathbf{J'}$. What we have to show is that if $\mathbf{J'}$ is OCBP-schedulable then at least one BI will provide a job J^{least} that can be selected for the least OCBP priority as well.

- **Case 1: There is a BI_i^{RS} whose J^{least} is a LO job.**
  In this case, OCBP can select the J^{least} of any such busy interval. This is because when evaluating whether a LO job can be assigned the least priority OCBP simulates the basic LO scenario, effectively doing the same check as MCEDF.
- **Case 2: The J^{least} in every busy interval is a HI job**
  In this case, the MCEDF rule to select the J^{least} in a BI implies that the end time of every BI_i^{RS} is later than the deadline of any LO job contained in it. Consequently, no LO job can be selected by OCBP, because in an OCBP simulation a least-priority LO job will terminate at a time equal to the end time of its BI_i^{RS}, thus missing its deadline.

  Therefore, because instance $\mathbf{J'}$ is OCBP-schedulable, OCBP should be able to select a HI job. Let us denote this job J'^{ocbp}. Let J'^{least} be the job selected by MCEDF for the BI that contains J'^{ocbp}. Like J'^{ocbp}, J'^{least} must be a HI job. Because MCEDF selects the latest-deadline HI job in the BI, we have: $J'^{\text{least}}.D \geq J'^{\text{ocbp}}.D$.

  The HI jobs are evaluated by OCBP using the HI scenario where no LO jobs are dropped and the jobs have $C_j(\text{HI})$ execution times. Because these execution times are larger or equal to the execution times in the basic LO scenario and no LO jobs are dropped we conclude that J'^{ocbp} and J'^{least} must be located in the same busy interval not only in the LO scenario, but also in the HI scenario evaluated by OCBP. The fact that J'^{ocbp} can be selected by OCBP means that if it terminates at the end of this HI busy interval then it still meets its deadline. But because the deadline of J'^{least} is not less than that of J'^{ocbp}, it is eligible to let J'^{least} terminate at the end of that HI busy interval as well, and hence it can also be selected by OCBP.

Thus, for an OCBP-schedulable instance, both algorithms can construct the same PT. MCEDF uses this priority table before the mode switch, thus having exactly the same behavior as OCBP under these conditions. After the mode switch OCBP meets the HI job deadlines without dropping the LO jobs, and hence MCEDF will surely be able to do the same because it drops the LO jobs and employs EDF, an optimal strategy. □

To the best of our knowledge, so far MCEDF is the only FPM scheduler that exploits the freedom to drop the LO jobs (or to reduce their priority) to perform, *in theory*, strictly better than OCBP, the

optimal fixed-priority scheduler. Nevertheless, recently some non-FPM algorithms were proposed that dominate OCBP, [Guo16], and for another non-FPM algorithm a proof of dominance over MCEDF is presented in [BB17].

5.4 MCEDF and Splitting

In this section we will introduce *splitting*, a theoretical transformation[7] of a job instance into a new instance where a HI job is equally divided into a certain number (called *split factor*) of equal smaller 'sub-jobs' with the same arrival time and deadlines as the original job and with execution times $C_j(LO)$ and $C_j(HI)$ that add up to that of the original job. Obviously, the splitting does not impact $Load_{\text{LO}}$ and $Load_{\text{HI}}$, but it reduces the WCET uncertainty ΔC_j and $Load_{\text{MIX}}$. Recall from Section 2.1.3 that ΔC_j reflects the amount of information that mode-switched policies get online about the mode switch and the future workload based on the current mode. The less the uncertainty the less information has to be learned by such policies from the jobs that have not yet signalled their termination in order to make a proper scheduling decision.

Therefore, for mode-switched policies, such as MCEDF, the uncertainty reduction due to splitting can translate a non-schedulable instance into a schedulable one. An infinitely extended splitting of all HI jobs can bring the optimality of a mode-switched policy asymptotically close to that of (practically impossible to realize) clairvoyant scheduling, which 'knows' all information on the future workload in advance. For some instances, a finite splitting is enough to equate the clairvoyant scheduling.

On the other hand, mode-agnostic policies, such as OCBP, can never show any improvement due to splitting, as they account only for splitting-agnostic maximal workload *apriori* and do not profit from any online information *aposteriori*. This observation, for what concerns OCBP, are confirmed in our experiments reported in Section 8.

The following example demonstrates the effect of splitting. It has $Load_{\text{MIX}} = 1.166\ldots$:

Job	A	D	χ	C(LO)	C(HI)
1	0	6	LO	5	5
2	0	12	HI	2	12

This instance is not schedulable because the necessary condition in Formula (3) is broken and due to uncertainty of the execution time. If J_1 executes first then J_2 starts at time 5. In the LO scenario there would be no problem, but J_2 misses its deadline should it 'decide' to execute in the HI scenario, for 12 time units. Otherwise, if J_2 starts first then even in the HI scenario it meets its deadline (whereby the LO job J_1 can be dropped), but there is a problem in the LO scenario, as J_2 would delay J_1 by two time units, leading to a missed deadline. The clairvoyant scheduler would know the scenario in advance and make the proper choice accordingly.

It is easy to check that after splitting J_2 into two jobs, the instance becomes MCEDF-schedulable.

Job	A	D	χ	C(LO)	C(HI)
1	0	6	LO	5	5
21	0	12	HI	1	6
22	0	12	HI	1	6

MCEDF produces the following priority table: $PT = (J_{22}, J_1, J_{21})$. The schedule first executes J_{22} until termination, effectively getting from it the online knowledge of the execution scenario that was missing in the previous case. If job J_{22} has executed in the LO scenario, J_1 can follow, starting at time 1, and then J_{21} can run from time 6 even until time 12 in the HI scenario. If job J_{22} has executed in the HI scenario, J_1 will be skipped, and J_{22} together with J_{21} meet the deadline. Compared to the instance before the split, $Load_{\text{MIX}}$ reduces from $1.166\ldots$ to 1, whereas $Load_{\text{LO}} = 0.833\ldots$ and $Load_{\text{HI}} = 1$ stay constant, thus not showing any advantage of the splitting.

Note that splitting, even being a theoretical transformation, may have some practical significance. Its practicability depends on the WCET tools, in particular, by what extent the sum of WCETs may

[7] it ignores the runtime overhead that would be incurred by fragmentation of tasks in practice

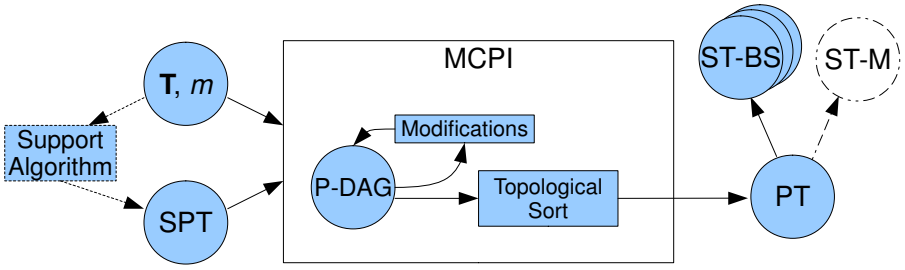


Fig. 15 Proposed algorithm MCPI. **T** stands for task graph and *SPT* for support priority table. ST-BS stands for static table per (job-specific) basic scenario. ST-M stands for static table per mode.

change by the splitting of jobs into sub-jobs. Note that despite the fact that the arrival times of all sub-jobs are equal, they are not restricted to be data-independent of one another. This is due to the fixed-priority per job scheduling policy, which has the property that the jobs with equal arrival times never preempt each other but instead execute (on a single processor) in a sequential priority-driven order whereas the sequential blocks of the job code can be assigned to the sub-jobs in the same order.

6 Multi Processor Scheduling – MCPI

We define here the *Mixed Criticality Priority Improvement* (MCPI) algorithm. It is, basically, an algorithm to compute job priorities under list scheduling offline, while online for sustainable termination times we use the sustainability replacement policy.

6.1 The Main Idea of MCPI

As previously discussed, our aim is to overcome the limitation of Audsley approach in multiprocessor systems. Recall (see Sec. 3.1) that Audsley approach assigns the priorities starting from the least one. This makes it problematic to compute the job termination times. In fact, MCEDF proceeds in a similar way and hence extending it to multiprocessors would encounter the same obstacle.

MCPI proceeds differently and assigns priorities from the highest one. Thus, we can trivially calculate exact termination times. However, a drawback of our approach is that, unlike Audsley approach, just selecting a job that meets the deadline is not enough for optimality, as, unlike *e.g.,* OCBP, the selection made out of different alternatives does, in fact, have effect on the final outcome. Thus we lose the property of Audsley approach that ensures optimality of the priority table while just making an arbitrary valid job selection at each step.

Therefore, in MCPI we compensate for this by greedy priority-table improvement modifications, which turn out to perform quite well. As we show in Section 7, on single processor MCPI is equivalent to MCEDF, thus dominating Audsley algorithm (OCBP) in this case. On the other hand, experiments (see Section 8) show that if an 'ideal' Audsley approach could find exact upper-bounds on termination times then it would constitute a serious competitor to MCPI on multiple processors. Thus, we have also encountered an interesting direction for future research.

Fig. 15 shows an overview of MCPI. The algorithm takes as input the task graph **T**, the number of processors m and a priority table SPT. The latter may be generated by any known multiprocessor algorithm. We call this algorithm *support algorithm* and the input priority table *Support Priority Table*, by analogy to MCEDF. Our "priority improvement" algorithm MCPI tries to improve the priority table generated by the support algorithm so that the termination times of HI jobs can be improved and thus the mixed-critical schedulability criteria can be met for a larger set of problem instances. Similarly to MCEDF, the algorithm is based on the concept of *Priority Direct Acyclic Graph*, (P-DAG), but unlike MCEDF, we construct the P-DAG by adding at each step a job with

Algorithm: *MCPI*
Input: **processor count** m
Input: **task graph T**
Input: **priority table** SPT, PT_{HI}
Output: **priority table** PT
1: $SPT \leftarrow PrecedenceComplianceTransform(SPT, \mathbf{T})$
2: $CheckLOscenarioSchedulability(\mathbf{T}, SPT, m)$
3: $G \leftarrow MCPI\_PDAG(\mathbf{T}, SPT, \emptyset, m)$
4: $PT \leftarrow TopologicalSort(G)$
5: **if** $anyScenarioFailure(PT, PT_{\text{HI}}, \mathbf{T}, m)$ **then**
6: **return** (FAIL)
7: **end if**

Fig. 16 The MCPI algorithm

the highest priority (according to SPT) and not the least one. The idea of MCPI resembles the $SimplisticImproveHIJobs$ algorithm we have seen in Section 4.1. Each time we insert a HI job, we apply a modification to the priority order given by table SPT, to improve the schedulability of HI scenarios. The modification is done in a '*insertion-sort*' way, *i.e.,* we fist put the job at the least priority position and then try to raise its priority by repetitive swapping with the job at the previous position. We only swap the HI job with a LO job (never with another HI job) and we accept the swap only if the concerned LO job and the other jobs with less priority do not start missing their deadlines. Note that we do not do a usual 'insertion-sort' on a *linear array* (*i.e.,* the priority table), as such a naïve approach may encounter some artificial hazards, as shown earlier in Section 4.1. Instead, we move the HI job along the *stem of a subtree* in the P-DAG. When all jobs have been added to the P-DAG (with an improvement attempt for each HI job), a priority table PT_{LO} is obtained by topological sort of the P-DAG.

The algorithm, in fact, takes two tables, SPT_{LO} and SPT_{HI}, from the support algorithm, which is thus applied twice, for LO and HI modes, as explained later in Section 6.3. However, MCPI improves only the SPT_{LO} table to obtain PT_{LO}, whereas it assumes that PT_{HI} is just copied from SPT_{HI}. Therefore, we denote PT_{LO} and SPT_{LO} just as PT and SPT for convenience. When the PT table construction is finished we know that the system is schedulable using PT in LO mode. Then, we also test schedulability of all possible switches to HI mode, accompanied by switches from PT to PT_{HI}. For this, in line with Theorem 3, we test all job-specific scenarios 'HI-J_j', whereas, unlike MCEDF, we may not use a more efficient check from Theorem 7, reserved for the single-processor case.

As explained in Section 2.1.1, to ensure sustainability the final priority tables should be translated into static tables for the STTBS policy (ST-BS in Fig. 15). In [SPBB15b] we describe alternative approach, which for multiprocessors works only in a high percentage of cases but not always, which consists in generating 'static tables per mode' (ST-M), which is more efficient because only two tables need to be generated: a LO-mode table and a HI-mode table. Recall that the replacement of FPM by STTBS policy is needed only in the case of multiple processors.

In the next subsections we describe the MCPI algorithm itself and our support algorithm for it.

6.2 MCPI Algorithm Specification

The pseudocode of MCPI is given in Fig. 16. The algorithm takes as inputs the support priority table to improve, SPT, the table to use in HI mode, the task graph $\mathbf{T}$ and the number of processors m. We update SPT to make it precedence compliant, and we ensure precedence compliance in the intermediate tables obtained by improvements of SPT. Precedence compliance ensures that we handle the jobs in a topological order of the job precedences, *i.e.,* from task-graph sources to sinks. The SPT is updated by *PrecedenceComplianceTransform* algorithm, which sorts SPT in the lexicographic order, where the first criterion is whether the jobs are related by $\rightarrow^*$ (*i.e.,* the existence of a task-graph path) and the second criterion is preserving the same relative order as the original SPT.

We then check LO scenario schedulability, by running the list scheduler with priorities SPT in the LO mode. If the LO schedulability holds, it will be maintained invariant during the priority

Algorithm: *MCPI_PDAG*
Input: **processor count** m
Input: **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
Input: **priority table** SPT
In/out: **forest P-DAG** $G(\mathbf{J}', \triangleright)$
1: **while** $G.\mathbf{J}' \neq \mathbf{T}.\mathbf{J}$ **do**
2: $J^{\text{curr}} \leftarrow SelectHighestPriorityJob(\mathbf{T}.\mathbf{J} \setminus G.\mathbf{J}', SPT)$
3: $\mathbf{J}'' \leftarrow G.\mathbf{J}' \cup \{J^{\text{curr}}\}$
4: $PT'' \leftarrow (TopologicalSort(G) \frown J^{\text{curr}})$
5: $\mathbf{T}'' \leftarrow MaximalSubgraph(\mathbf{T}, \mathbf{J}'')$
6: $\vdash \leftarrow SimulateListSchedule(\text{LO},\ \mathbf{T}'',\ PT'', m)$
7: $\overset{\mathbf{J}''}{\sim} \leftarrow EstimatePotentialInterference(\text{LO},\ \mathbf{T}'', m)$
8: $G.\mathbf{J}' \leftarrow \mathbf{J}''$
9: **for** *all trees ST of G* **do**
10: **if** $J^{\text{curr}}.\chi = \text{LO}$ **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: **if** $\exists\ J' \in ST\colon\ J' \overset{\mathbf{J}'}{\sim} J^{\text{curr}}\ \vee\ J' \rightarrow J^{\text{curr}}$ **then**
16: $ConnectAsRoot(ST, J^{\text{curr}})$
17: **end if**
18: **end if**
19: **end for**
20: **if** $J^{\text{curr}}.\chi = \text{HI}$ **then** $PullUp(J^{\text{curr}}, G, \mathbf{T}, SPT, m)$
21: **end while**

Fig. 17 The algorithm for computing priority tree in MCPI

improvement. Otherwise the algorithm terminates with a failure (not shown in the pseudocode). The schedulability is maintained by letting the algorithm accept priority swappings only if they do not lead to a LO-scenario deadline miss when simulating the complete set of jobs. Subroutine *MCPI_PDAG* generates a (directed-forest shaped) P-DAG, based on the support priority table SPT and the HI-job priority improvements. Then, similarly to MCEDF, we obtain a priority table from G by using *TopologicalSort* procedure which traverses the trees in G from the leafs to the roots while adding the visited nodes to the tail of PT. Finally, the subroutine *anyScenarioFailure* verifies schedulability in the case of any possible switch to HI mode.

In Fig. 17 subroutine *MCPI_PDAG* is shown. It combines elements from the 'simplistic improvement' and 'forest P-DAG' subroutines, Fig. 6 and Fig. 9. It takes as inputs the task graph $\mathbf{T}$, the support priority table SPT, and the graph G generated so far (which is empty at the beginning). In every iteration, the algorithm handles J^{curr}, the highest-priority job of table SPT which is not yet in G and eventually adds that job to G. The algorithm terminates when all jobs have been added to G.

First, the current job is added to a priority table at a position inferior to all previously handled jobs, using priority-table concatenation operator '$\frown$'. List-schedule simulation is carried out to discover which of the previous jobs would interfere with the current job when that job would have the least priority. We say that the interference relation '$\vdash$' is thus calculated. Next to the *actual* interference relation with job J^{curr}, we also estimate the *potential* interference relation '$\overset{\mathbf{J}'}{\sim}$' between all jobs in $\mathbf{J}'$. For this we currently use the makespan algorithm to derive the single-processor busy intervals as explained earlier (Theorem 32), though better approximations of potential interference are to be investigated in future work, to take into account the number of available processors m.

After that:

if the current job criticality is LO we add an edge to J^{curr} from all the roots of the trees ST present in G where $\exists\ J'\colon J' \vdash J^{\text{curr}}$. This makes J^{curr} the new root of ST. This is needed to ensure that the priorities derived from G are compliant to Equation (15). In addition we do the same for the subtrees containing task-graph predecessors of J^{curr}, to ensure precedence compliance, as defined by Equation (6).

if the current job criticality is HI we do similar actions as in the case of a LO job, but instead of the (actual) interference relation we use the *potential* interference relation. The reason for this

```
Algorithm:  PullUp
Input:  job J
In/out:  forest P-DAG G
Input:  task graph T(J, →)
Input:  priority table SPT
 1: DONE = ∅
 2: while LOpredecessors(J, G) ≠ DONE do
 3:     J' ← SelectLeastPriorityJob( ( LOpredecessors(J, G) \ DONE), SPT)
 4:     DONE ← DONE ∪ {J'}
 5:     if CanSwap(J, J', G) then
 6:         TreeSwap(J, J', G)
 7:         DONE ← DONE ∩ LOpredecessors(J, G)
 8:     end if
 9: end while
```

Fig. 18 The pull-up subroutine

```
Algorithm:  CanSwap
Input:  HI job J
Input:  LO job J'
Input:  forest P-DAG G
Input:  task graph T(J, →)
Input:  priority table SPT
 1: if J' →* J then
 2:     return False
 3: end if
 4: TreeSwap(J, J', G)
 5: PT ← (TopologicalSort(G) ⌢ (SPT |≺ J))
 6: allDeadlinesMet ← SimulateListSchedule(LO, T, PT)
 7: return allDeadlinesMet
```

Fig. 19 The subroutine for checking the feasibility of a priority swap

difference is that for HI jobs the final priority of J^{curr} is not known *apriori* as for such jobs 'insertion-sort' priority improvements are applied.

Note that the difference between the two cases given above is that for a HI job we use the '$\overset{\mathbf{J'}}{\sim}$' relation instead of a smaller relation '$\vdash$'. This is needed to ensure correctness of further modifications of G. The modifications themselves are done by subroutine *PullUp*, which is the core of the algorithm.

Subroutine *PullUp* is described in pseudocode in Fig. 18. It modifies the P-DAG generated so far, trying to raise the priority of the given HI job. This is done by 'swapping' its position in the graph with LO jobs while preserving the schedulability of the LO scenario. This improves the schedulability in the HI scenarios. Note that if this subroutine were not called then the algorithm would just generate a P-DAG of the initial priority table SPT.

Procedure $LOpredecessors(J, G)$ returns for node J the set of its direct P-DAG predecessors[8] of LO criticality: $\{J_s \mid J_s \rhd J,\ \chi_s = \text{LO}\}$. At each step in Fig. 18 we select the least SPT-priority P-DAG predecessor from the working set $LOpredecessors(J, G) \setminus DONE$, where $DONE$ is the set of the LO jobs which we already tried to swap with the current job. Then, subroutine *CanSwap* checks whether the current job and the selected job can swap priorities. If so, we apply the actual swapping transformation to graph G, otherwise the selected job is added to $DONE$, to avoid trying to perform the given swapping again. The subroutine proceeds until we have tried to swap with all LO predecessors of J in the P-DAG.

As shown in Fig. 19, subroutine *CanSwap* uses a private copy of graph G to perform a tentative swap modification and then evaluates its impact. To do so, it constructs a complete priority table by concatenating the priorities obtained from the modified graph G with the trailer of SPT table that contains the jobs that were not yet handled. This is the part of SPT table that follows after the current job J, hence the notation $(SPT \mid\prec J)$; more precisely, this is the priority table that consists of the jobs whose SPT priority is less than that of J and which puts these jobs in the same relative order as SPT. Thus, we check all jobs and not only those whose priorities have been modified. This

[8] they are also tree-children of node J, as in a P-DAG forest the edges are directed from children to parents

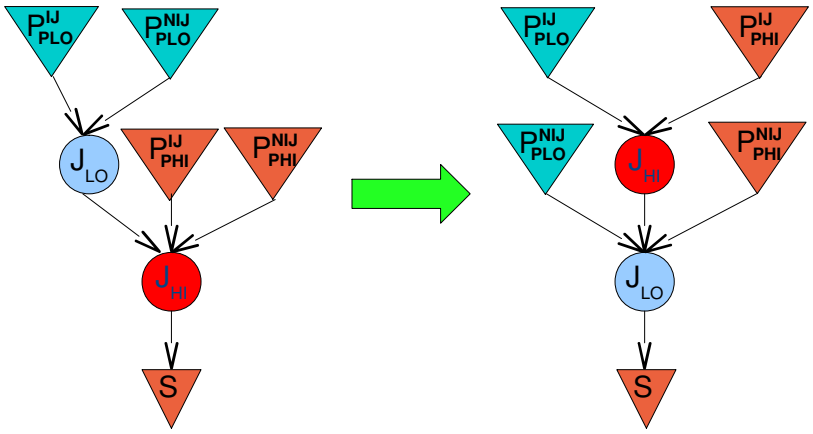


Fig. 20 The effect of a Swap.

is required because, unlike the single-processor case, modifying adjacent priorities of a pair of jobs on a multiprocessor may impact not only their schedules but also the schedules of all jobs that have less priority. We accept the swapping only if it does not directly lead to a deadline miss for any job. This is how we maintain the schedulability in LO mode as invariant of the algorithm. Note that *CanSwap* immediately rejects to swap J and J' if $J' \rightarrow^* J$, to maintain precedence compliance of priorities.

Subroutine $TreeSwap(J_{\text{HI}}, J_{\text{LO}}, G)$ performs the 'swap' modification of graph G, defined as follows:

Definition 39 (Swap Modification) Let $G = (\mathbf{J}', \rhd)$ be a forest P-DAG, let $J_{\text{LO}} \rhd J_{\text{HI}}$ be the edge between the two jobs to be swapped and let $\mathbf{J}''$ represent the subset of jobs whose priorities can be possibly higher than or equal to J_{HI} after the swap is performed:

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

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

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

The swap is illustrated in Fig. 20. In the original P-DAG the orange triangle marked with S represents the P-DAG successors of J_{HI}, while the triangles marked with $P^{\text{IJ}}_{\text{PHI}}, P^{\text{NIJ}}_{\text{PHI}}$ and $P^{\text{IJ}}_{\text{PLO}}, P^{\text{NIJ}}_{\text{PLO}}$ are, respectively the P-DAG predecessors of J_{HI} and J_{LO}. More specifically, we assume in the figure for $P^{\text{IJ}}_{\text{PHI}}$ and $P^{\text{IJ}}_{\text{PLO}}$ are subtrees where the condition '*contains a job that either potentially interferes with* J_{HI} *or is its task-graph predecessor*' is true, while it is false for $P^{\text{NIJ}}_{\text{PHI}}$ and $P^{\text{NIJ}}_{\text{PLO}}$.

After the swap modification, the *PullUp* subroutine updates the set *DONE* and reiterates.

Example 40 Consider again the instance and the priority table of Example 20. Let us apply MCPI on them. The table PT is already precedence compliant, so *PrecedenceComplianceTransform* will not modify it. Then we check LO schedulability, by simulation. The result of this simulation is, in fact, the Gantt chart in Fig. 8(a), where it is easy to check that no job misses its deadline.

Then we apply subroutine *MCPI_PDAG*. The graph G obtained in the first few iterations before the first *PullUp* is illustrated in the left side of Fig. 21. In the first iteration we add $s1$ to G. No job interferes with it, so we proceed with the second iteration. $s2$ is added to G, again we do not have any interference. Next we add job $s3$, and we have the following interference 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 and $s3 \overset{\mathbf{J}'}{\sim} s4$, we add the edge $s3 \rhd s4$, since $s3$ is the root of the only tree of G.

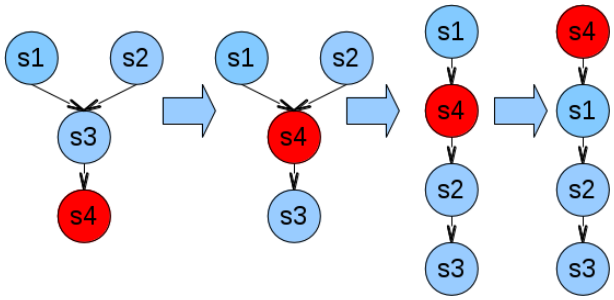


Fig. 21 The effect of subroutine *PullUp* on job $s4$ in Example 40.

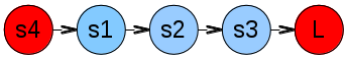


Fig. 22 The final P-DAG generated by MCPI in Example 40.

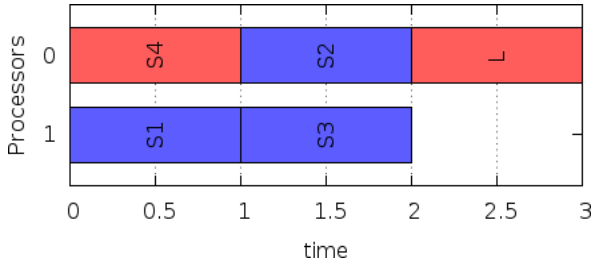


Fig. 23 The schedule obtained by MCPI in Example 40.

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. 21. 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 P-DAG shown in Figure 22.

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 LO schedule of Fig. 23. The reader may easily verify that using the initial priority assignment, whose LO-scenario schedule is shown in Fig. 8(a), will fail if instead of following the LO scenario job $s4$ will continue execution until $C(HI) = 3$ time units (which, in fact, happens in scenario HI-$s4$). At the same time, using the table generated by MCPI , which results in the LO-mode behavior shown in Fig. 23, $s4$, having the highest priority, starts earlier and would meet its deadline even in this scenario.

Below we give two theoretical results for MCPI.

Lemma 41 *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 23. There are only two differences of the former subroutine *w.r.t.* the latter:

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

Since, by Lemma 16, with extra edges added G still remains a P-DAG, we observe, by Theorem 23, 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 a *swap* operation G remains to be a P-DAG. Let J^s be the job that is pulled up and let $TreeSwap(J^s, J^p_k, G)$ be the k-th swap. Note that J^s is a HI job, and just before the first swap it has been root of some tree ST. By construction, all the other trees, which have not been initially connected to J^s, contain only jobs that are not in potential interference relation with J^s and any other job in ST, so their execution will not be influenced by

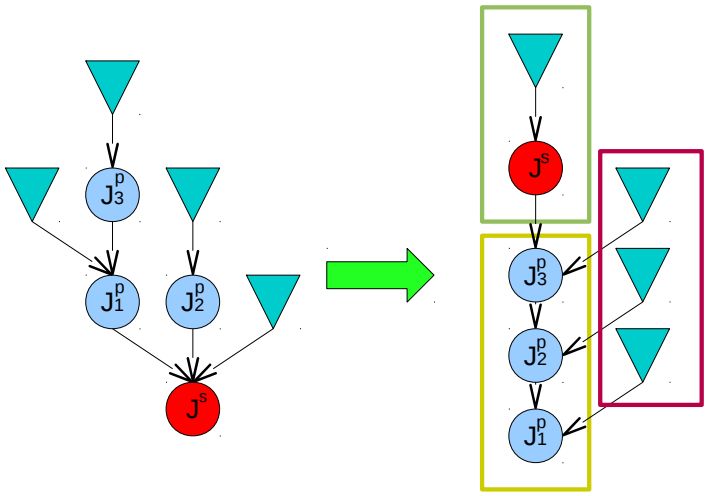


Fig. 24 The effect of multiple Swaps on ST, $k = 3$.

any change in ST. Thus, without loss of generality, we can assume that G is composed of only one tree (*i.e.,* $G = ST$).

After the first swap, G is still a tree, such that J_1^p is the new root After multiple swaps, the situation will be as illustrated in Fig. 24. On the left side of the figure we have the initial ST with HI job J^s as the root. After swapping J^s with J_1^p,J_2^p and J_3^p (in this order), we obtain the tree on the right side. We can distinguish three areas in the tree: a chain of LO jobs in the lower part (inside the yellow box), connected to a subtree that has J^s as root (the green box) and some subtrees connected to the LO jobs in the chain that are not in potential interference relation with J^s (the red box).

Let us assume by contradiction that after the k-th swap – $TreeSwap(J^s, J_k^p, G)$ – the resulting graph $G' = (\mathbf{J}', \triangleright)$ is no longer a P-DAG. By Lemma 21 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 $TreeSwap(J^s, J_k^p, G)$ all the possible $J' \vdash J''$ relations that were not present before the swap are such that either $J'' = J_k^p$ or $J_k^p \rightarrow^* J''$. This is because, by lowering J_k^p priority (*i.e.,* shifting forward its execution), it might enter in the execution window of another job and get interfered by it. The same holds for its successors in $\mathbf{T}$.

For $J'' = J_k^p$, we can rewrite our contradicting hypothesis as follows:

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

After the swap, J_k^p is the root of a subtree ST_k. So $\forall J \in ST_k, \ J \triangleright^* J_k^p$. All jobs J' that are not in ST_k are either the chain below J_k^p (yellow box) or in the side subtrees branched into them, included in the red box. For the jobs J' in these subtrees we can show that: $J' \nvdash_{\mathcal{S}} J_k^p$. This is so because by properties of swap they are not in potential interference relation with J^s and hence they also are not in potential interference relation with J_k^p, as J^s is swapped only with jobs in the same potential interference equivalence class. For jobs $J' = J_1^p, J_2^p, \ldots J_{k-1}^p$ in the yellow box we have that they have lower priority than J_k^p and hence: $J' \nvdash_{\mathcal{S}} J_k^p$.

Let us now consider jobs J'' such that $J_k^p \rightarrow^* J''$. An invariant of our algorithm is precedence compliance, *i.e.,* $J_k^p \rightarrow^* J'' \Rightarrow J_k^p \triangleright^* J''$. This means that all such J'' are among $J_1^p, J_2^p, \ldots J_{k-1}^p$ in the chain below J_k^p. The same reasoning as in the previous case holds. □

Theorem 42 *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*

$$O(EK^2 + K^3(\log K + m)) \tag{18}$$

Proof One of the main contributions to the computational complexity of the algorithm is given by the high number of list schedule simulations. The complexity of one simulation is, according to [SPBB15b]:

$$O(E + K(\log K + m)) \tag{19}$$

Let us now analyze the algorithm of Fig. 16 line by line. Routine *PrecedenceComplianceTransform* has a complexity of $O(K^2)$. This is because to prepare for sorting the jobs for precedence compliance we have to compute the transitive closure of the precedence relation, which takes $O(K^2)$ time, because the maximum number of predecessors for each job is $O(K)$. *CheckLOscenarioSchedulability* does one simulation, thus it has a complexity of (19). *MCPI_PDAG* gives the highest contribution, and its complexity will be discussed later. *TopologicalSort* of a forest has complexity $O(K)$ [CLRS01]. Finally, *anyScenarioFailure*, according to Theorem 3, does $O(K)$ simulations, and thus its complexity is $O(K(E + K(\log K + m)))$.

Let us now analyze subroutine *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 (19). 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* (Fig. 18). In this subroutine there is a while loop that is executed once for each LO predecessor of the current job in the P-DAG, thus a $O(K)$ number of times inside *PullUp* and $O(K^2)$ number of times per 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 (19). Thus the *CanSwap* subroutine gives the main total contribution to the complexity of the algorithm, executing in total $O(K^2)$ simulations of complexity (19), which gives the result given in (18). □

Note 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 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, the complexity can be assumed as follows:

$$O(K^3 \log K)$$

Compared to $O(K^2)$ complexity of MCEDF, the higher complexity here can be explained by non-applicability of some computation economizing properties, such as Theorem 7 and Lemma 29, to the multiprocessor case.

6.3 The Support Algorithm for MCPI

Based on related-work analysis, by default for MCPI we assume that the support algorithm is *EDF with modified deadlines and density threshold* (EDF-DS). To adapt this algorithm to precedences and mixed criticality we compute the ALAP deadlines $D_j^*(\chi)$ in mode graphs $\mathcal{T}_{\text{MIX}}$ and $\mathcal{T}_{\text{HI}}$ for modes $\chi = \text{LO}$ and $\chi = \text{HI}$ respectively. The ALAP deadlines are used to compute SPT_{LO} and SPT_{HI}.

The support priority table in mode χ is generated by sorting jobs in the lexicographic order according to two criteria. The first criterion employs the notion of *job density*. Adapting the formula from related work, the job density in mode χ is $\delta_j(\chi) = C(\chi)/(D_j^*(\chi) - A_j)$. The jobs are split into two groups: high-density jobs, with $\delta_j(\chi) > thr$, and low-density jobs, the rest. We determined the best density threshold thr experimentally, $thr = 0.8$. According to the first criterion of EDF-DS, the high-density jobs have higher priority. Thus, this criterion corresponds to 'density separation' (DS).

The second criterion, applied to the jobs in the same group, is EDF-based. A job with a smaller deadline $D^*(\chi)$ has a higher priority. Note that the described support algorithm for MCPI has in common with that of MCEDF (see Section 5.2) that it also takes the WCET uncertainty ΔC_j into account, though in a different way. Recall that in mode graph $\mathcal{T}_{\text{MIX}}$ the job uncertainties are subtracted

from the deadlines. Since we compute the LO-mode table using this graph, among two jobs with the same deadline this algorithm will also prioritize the one with a higher uncertainty.

To indicate explicitly that we are using a support algorithm ALG we use the notation MCPI(ALG). Thus we use notation MCPI(EDF-DS) to indicate the use of MCPI with the above described support algorithm, while we will use MCPI(EDF) if we use the EDF policy without density separation.

Note also that in our MCPI experiments in Section 8 a support algorithm serves not only as part of MCPI but also as a reference point to evaluate the advantage of MCPI, being considered as a 'competitor' heuristics to calculate the FPM priority tables. As such, it serves as a 'state-of-the art' representative for our problem formulation. Indeed, the described algorithm has mixed-criticality 'awareness', as it gives HI jobs higher priority by reducing their deadlines (as we do in $\mathcal{T}_{\text{MIX}}$).

7 The Properties Common for MCEDF and MCPI

In this section we give some theoretical properties of MCEDF and MCPI for independent jobs on single processor. The main result is the optimality of MCEDF and MCPI in the class of scheduling algorithms where the HI jobs are in relative EDF order. Also, we show that MCEDF and MCPI are equivalent. Note that because these algorithms use LO-mode schedules to construct the priority tables, under the 'scheduling' we always mean the LO-mode scheduling unless mentioned otherwise.

In this section we assume that both algorithms use the same EDF-compliant support priority table SPT, that the jobs are independent and $m = 1$.

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

Lemma 43 (Subtrees and BIs) *In MCPI, as in MCEDF, for any subtree ST at any level of the P-DAG holds that the sub-instance composed of the jobs of ST consists of only one busy interval.*

Proof (sketch) For MCPI, we argue that this property is true by demonstrating that it is maintained at each basic step of the algorithm, *i.e.,* 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 interfere with the given job when it has the least priority. Since they interfere with the given job then they must be in the same busy interval. When a HI job is initially connected, the property holds by construction since in this case MCPI evaluates the $\overset{\mathbf{J}'}{\sim}$ relation which on single processor corresponds to busy intervals.

Now consider the swap modification according to Definition 39. After the swapping, the current HI job forms the same busy interval with the subtrees connected to it as connecting these subtrees is also based on relation $\overset{\mathbf{J}''}{\sim}$. The LO job which was swapped forms one busy interval with the current HI job and the whole job set $\mathbf{J}''$ by the observation that this was already the case before the swapping and the busy intervals do not change when priority assignment changes. □

Lemma 44 (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:*

$$\forall i, j \;.\; \chi_i = \chi_j \quad \wedge \quad J_i \rhd^* J_j \quad \Rightarrow \quad J_i \succ_{SPT} J_j \tag{20}$$

Proof (sketch) For MCEDF the Property (20) 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 a call to *CanSwap* returns 'false' and then a subsequent call returns '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 sequence of swapped LO jobs forms a P-DAG chain in the same order as the swapping is done. The job for which *CanSwap* would return

'false' would stay as P-DAG 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 necessarily followed by other 'false' results. To show this, recall that by Lemma 43 the HI job forms one busy interval (τ_1, τ_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 jobs with the larger deadline will be evaluated first. Therefore, if a 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 45 (HI-criticality EDF Compliance of Priority Table) Given an EDF-compliant SPT priority table, a 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:

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

Consider a problem instance $\mathbf{J}$ with h HI jobs. We can partition an EDF-compliant priority table into the following sequence of job sets:

$$PT \;:\; \mathbf{J}_1^{\text{LO}} \succ_{PT} \{J_1^{\text{HI}}\} \succ_{PT} \mathbf{J}_2^{\text{LO}} \succ_{PT} \{J_2^{\text{HI}}\} \succ_{PT} \ldots\; \mathbf{J}_h^{\text{LO}} \succ_{PT} \{J_h^{\text{HI}}\} \succ_{PT} \mathbf{J}_{h+1}^{\text{LO}} \tag{21a}$$

$$\text{HI jobs} \;:\; J_1^{\text{HI}} \succ_{SPT} J_2^{\text{HI}} \succ_{SPT} \ldots J_{h-1}^{\text{HI}} \succ_{SPT} J_h^{\text{HI}} \tag{21b}$$

where subscripts LO and HI indicate criticality level of jobs in the sets 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 46 (Compliant Tables from MCEDF/MCPI) *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 (21) where, in addition, the LO job sets $\mathbf{J}_i^{LO}$ are related to J_i^{HI} by $\rhd^{*LO}$:*

$$\textit{for} \;\; i = 1..h \;.\; \mathbf{J}_i^{LO} = \{J_j \mid \chi_j = LO \;\wedge\; J_j \rhd^{*LO} J_i^{HI}\} \tag{22a}$$

$$\mathbf{J}_{h+1}^{LO} = \{J_j \mid \chi_j = LO \;\wedge\; \nexists i : J_j \rhd^{*LO} J_i^{HI}\} \tag{22b}$$

Corollary 47 (Compliance and Sustainability) *The above lemma implies that MCEDF(EDF) and MCPI(EDF) can generate a priority table where HI jobs are put in the same relative priority order in PT_{LO} and PT_{HI}. Hence, by Corollary 5, for those algorithms, the FPM policy is sustainable per mode for any independent-job single-processor problem instance.*

Definition 48 (Least LO-Job Priority Table) Given a P-DAG G that is generated my MCEDF or MCPI(EDF) with support priority table SPT. A priority table obtained from graph G that can be partitioned as shown in Formulas (21) and (22) is called a *least LO-job 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 while keeping HI-criticality EDF compliance.

Lemma 49 (Inviolability of Least LO-Job Priority) *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 some HI-criticality SPT-compliant priority table where some LO jobs in some job sets $\mathbf{J}_i^{LO}$ 'violate the least LO-job priority constraint' in the sense that they have 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 LO jobs that in table PT' 'violate' the least priority constraint. Let J_j be the least-priority violating job from the respective set $\mathbf{J}_{i'}^{\text{LO}}$. 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):

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

where $\mathbf{J}'_1, \mathbf{J}'_2, \ldots, \mathbf{J}'_{i'}$ are sets of LO jobs, whereas $\mathbf{J''}_{i'}$ may also contain HI jobs. 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 a new table PT'', which should result in the same termination time for job J_j:

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

Let us now relate PT'' to the least LO-job priority tables obtained from MCEDF or MCPI(EDF). 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:

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

Also because, by our assumptions, J_j is the least priority violating job in set i' we have that $\mathbf{J''}_{i'}$ contains all other violating jobs from $\mathbf{J}_{i'}^{\text{LO}}$, and hence:

$$\bigcup_{i=1}^{i'} \mathbf{J}_i^{\text{LO}} \subseteq \left(\bigcup_{i=1}^{i'} \mathbf{J'}_i \cup \mathbf{J''}_{i'} \cup \{J_j\} \right)$$

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:

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

This table differs from the one obtained from MCEDF or MCPI(EDF) only in that it puts J_j and not $J_{i'}^{\text{HI}}$ at the last position.

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 $J_j \rhd^{*\text{LO}} J_{i'}^{\text{HI}}$ we have that J_j 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 claim.

Theorem 50 (Optimality Property) *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 Consider an instance $\mathbf{J}$ that is MC-Schedulable. The MCEDF and MCPI(EDF) algorithms will never fail in LO mode . 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 maintained as 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 misses its deadline in a HI scenario.
2. For MC-schedulable instance, even if we see the failure 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 to the theorem statement that MCEDF (resp. MCPI(EDF)) fail to produce a feasible schedule due to a failure in a HI scenario, whereas the optimal EDF-compliant algorithm can. By lemma's above, we can structure the failing solution in the form shown in Formulas (21) and (22).

By our assumptions the optimal priority table PT' is also HI-criticality EDF-compliant according to SPT and hence it 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 49 we should have:

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

where $\mathbf{J}_m^{\text{LO}}$ are the least LO-job 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 reduce 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 terminate 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 50:

Theorem 51 (Algorithm Equivalence) *When using the same EDF-compatible SPT table MCEDF and MCPI(EDF) are equivalent.*

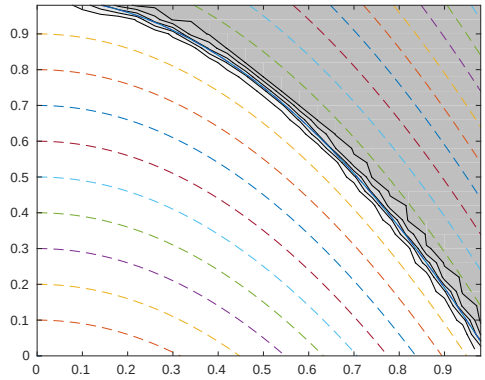
It can be easily shown that OCBP can also be restricted to be HI-criticality EDF-compliant, thus Theorem 38 can be seen as corollary of Theorem 50.

8 Implementation and Experiments

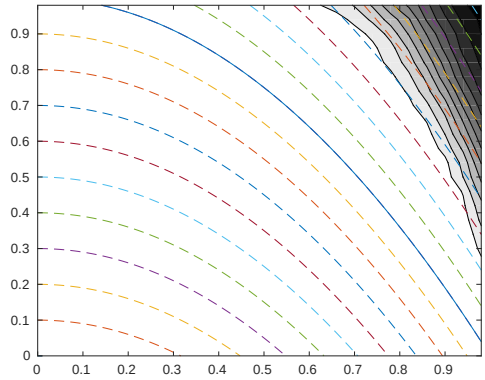
We evaluated the schedulability performance of MCEDF and MCPI in experiments with randomly generated job instances. The instance size was restricted due to the computation delays of job generation algorithm and our intention to evaluate a large number of points. The generated jobs had integer timing parameters, simulating CPU clock cycle count of some imaginary machine. Every job instance was generated for a target pair of values – LO and HI – of load or stress.

The method to generate a job instance worked as follows. First we randomly generated a tentative instance, not paying attention to the target loads. This was done by repeatedly generating a new sporadic task, i.e. sequence of jobs arriving one after another at random arrival intervals. For every job, both the job deadline and the arrival interval were uniformly distributed in a range 5K-25K (kilocycles), and the job's criticality level was set to HI (*i.e.,* $\chi = \text{HI}$) with a probability 50%. Every sporadic task produced just enough jobs to fill a random interval from 0 to a bound in range 15K-100K. The WCET $C_j(\text{LO})$ of each job was uniformly distributed between 0 and the relative deadline, each HI job had a $C_j(\text{HI})$ obtained by scaling the value $C_j(\text{LO})$ by a random factor [1..1000]. For MCEDF new sporadic tasks were invoked until all tasks together have produced more than 20 jobs, and then jobs were randomly removed until only 20 remained. For MCPI, instead of 20, we produced 30, 60 or 120 jobs in a similar way for processor counts $m = 2, 4, 8$. To finalize the job instance generation, the algorithm calculated the loads of the tentative instance and scaled the execution times to obtain the target load in the final instance.

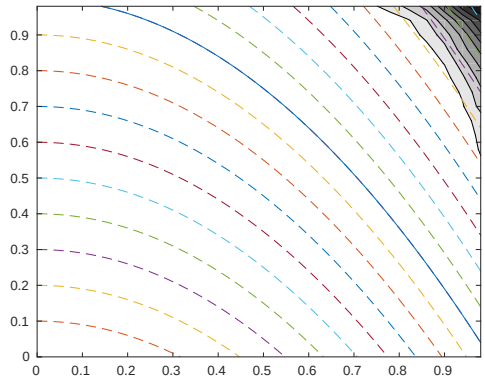
When scaling the loads, we took care that when $C_j(HI)$ would have to be scaled below $C_j(LO)$, it is instead set to $C_j(LO)$. This could result in imprecise final $Load_{\text{HI}}$. As a result, there was a load scaling problem, as the scaling sometimes failed to approximate the target load with the specified precision. In this case we cancelled the generated instance and made another attempt to generate it



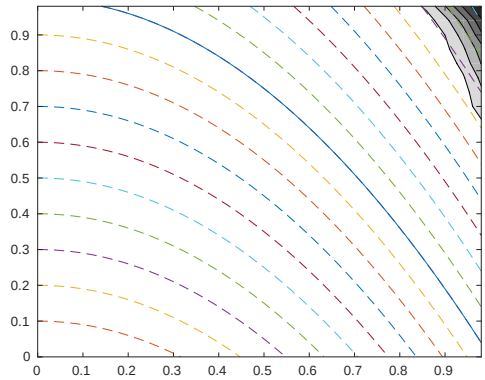
(a) Region where problem instances were generated, 537460 instances in total, uniformly distributed in the region.



(b) Distribution density of non OCBP-schedulable instances, 14% of total (Black color is 100% density)



(c) Density of non MCEDF-schedulable instances, 2.1% of total



(d) Density of non MCEDF-schedulable instances after split, 1.25% of total

Fig. 25 The contour graphs of random instances; the horizontal axis is $Load_{\mathrm{LO}}$, the vertical is $Load_{\mathrm{HI}}$.

until multiple attempts produced no satisfactory load scaling result within a timeout. Due to this, and due to high complexity of load calculations the job generation process itself took a considerable time in the experiments.

8.1 MCEDF Experiments

We ran multiple job generation experiments, ranging each target of $Load_{\mathrm{LO}}$ and $Load_{\mathrm{HI}}$ from 0.0025 to 1 with step 0.0025. Per each target, ten experiments were run, generating the points lying near the target with tolerance 1%. We only selected the 'overloaded' targets *i.e.,* those lying at or above the parabola $Load^2_{\mathrm{LO}}(\mathbf{T}) + Load_{\mathrm{HI}}(\mathbf{T}) = 1$, yielding instances where OCBP could potentially fail [LB10]. By looking at the loads below 1 we compare both OCBP and MCEDF to the clairvoyant scheduler, which can schedule all such points and which gives an upper bound on the best scheduling performance. Fig 25(a) gives the contour graph of the distribution of the generated points in grayscale. The grid follows the parabolic lines of equal $Load^2_{\mathrm{LO}}(\mathbf{T}) + Load_{\mathrm{HI}}(\mathbf{T})$. The total number of trials was 537460.

Around 14% (75203) of points showed failure for OCBP. In those 14%, roughly 2.1% (11316) were not schedulable by MCEDF as well, whereas 11.9% (63887) were schedulable by MCEDF. Thus, MCEDF proved to reduce the set of non-schedulable instances by a factor 6-7. The distributions in Fig. 25 suggest that MCEDF is less sensitive to high loads.

For the 2.1% (11316) non-MCEDF schedulable jobs we ran additional experiments. We considered *splitting* (see Section 5.4), a transformation of a job instance into a new instance where a HI job is

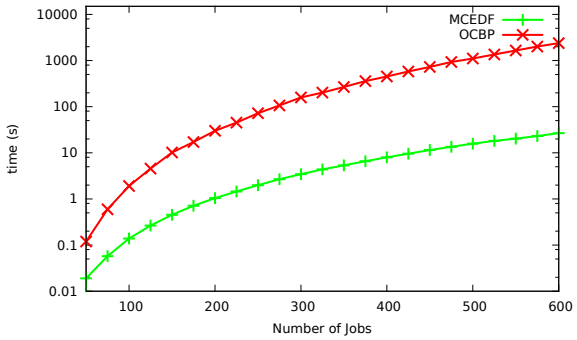


Fig. 26 The measured computation times of OCBP and MCEDF

m	jobs	arcs	step	δ	σ_s	instances
2	30	20	0.005	0.01	3.2	128800
4	60	40	0.02	0.05	6	50500
8	120	80	0.05	0.125	12	31575

m	EDF	EDF-DS	MCPI(EDF)	MCPI(EDF-DS)	diff(%)	diff-DS(%)
2	20924	21023	27375	27467	**30.83%**	**30.65 %**
4	6839	6887	8263	8310	**20.82%**	**20.66 %**
8	3065	3082	3521	3538	**14.88%**	**14.80%**

Table 1 Experimental results for MCPI. The upper part gives characteristics of the problem instance generation process, including the total number of instances generated. The bottom part specifies how many of those instances were schedulable by the support algorithms (on the left) and by the MCPI applied to the support algorithms (in the middle). It also gives (on the right) the percentage of the difference between the MCPI and the respective support algorithm.

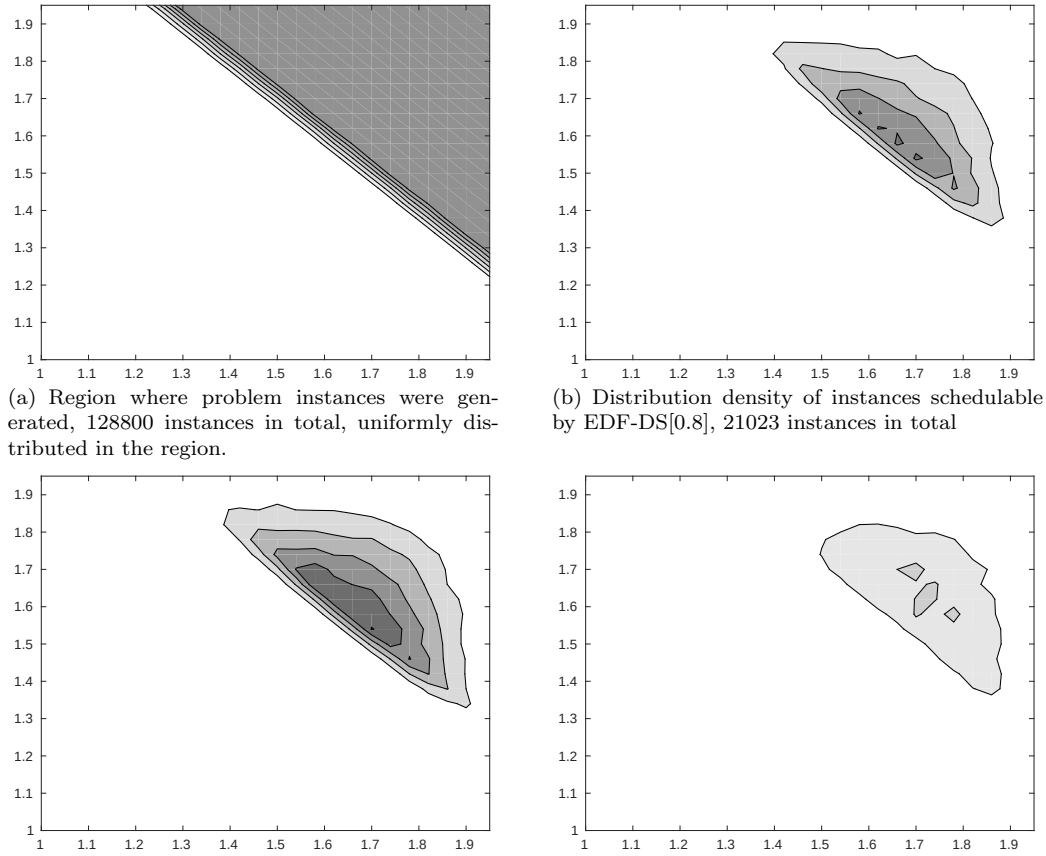
equally divided into a certain number (called *split factor*) of equal smaller jobs, whose total execution times $C_j(LO)$ and $C_j(HI)$ add up to that of the original job. Recall that splitting reduces the WCET uncertainty of jobs and that for mode-switched policies, such as MCEDF, this can lead to improved schedulability, whereas mode-agnostic policies, such as OCBP, cannot take any advantage of splitting.

We split all HI jobs by factors $2, 3$, and 4. This kept the load the same but reduced the WCET uncertainty. As expected, after splitting the instances remained to be non-OCBP schedulable but the number of non-MCEDF schedulable instances has reduced, coming to 1.25% (6735). So if we could accept this load-preserving transformation, we would go from 14% non-schedulability of OCBP to the 1.25% non-schedulability of MCEDF. Note that 0.85% (4581) were gained due to the splitting, whereby in the most of cases, 0.55% (2961), split factor 2 was sufficient. So assuming that in practice we could split the HI jobs into a few sub-jobs such that both WCET values scale, then we could in many cases obtain a schedulable instance. That the fragmentation of jobs would preserve the same total WCET is likely to be an overly optimistic assumption for the WCET tools, but still doing this is worth a try.

We also performed some experiments to evaluate the computation times of both algorithms, implemented using the same software library. Every point was obtained as the average computation time for 20 different randomly generated instances with $Load_{\text{LO}} = Load_{\text{HI}} = 0.8$. The results are shown in Fig. 26. They confirm our expectation of almost one order of magnitude of difference, as we estimate the best direct implementation of OCBP to be $O(K^3)$ and the best MCEDF to be $O(K^2)$ for K jobs, according to Lemma 36.

8.2 MCPI Experiments

We evaluated the schedulability performance of MCPI comparing it with the performance of its support algorithm. We restricted our experiments to "hard" task graphs, *i.e.,* those whose points in



(a) Region where problem instances were generated, 128800 instances in total, uniformly distributed in the region.

(b) Distribution density of instances schedulable by EDF-DS[0.8], 21023 instances in total

(c) Distribution density of instances schedulable by MCPI , 27467 instances in total

(d) Density of instances schedulable by MCPI and not by EDF-DS[0.8], 6444 instances in total

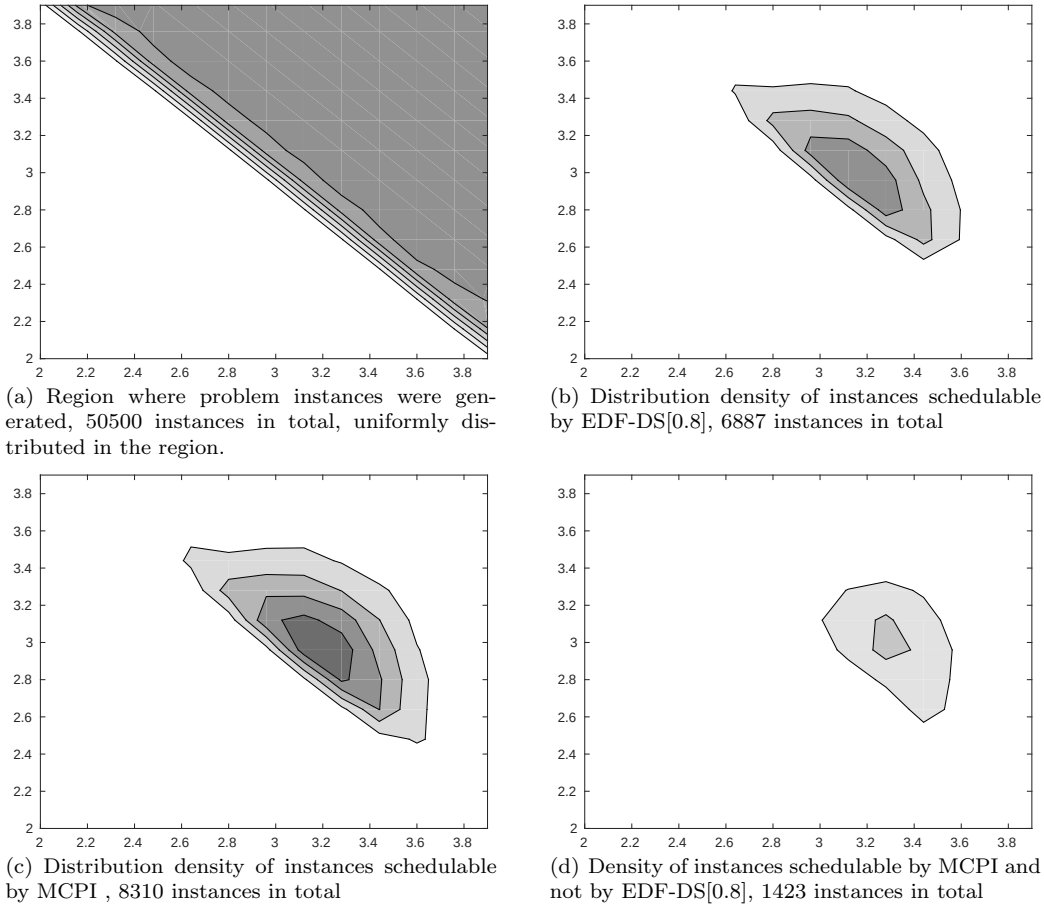
Fig. 27 The contour graphs of random task graphs for 2 processors. The horizontal axis is $Stress_{\text{LO}}$, the vertical is $Stress_{\text{HI}}$.

the (LO,HI) stress space lie above a certain line:

$$Stress_{\text{LO}}(\mathbf{T}) + Stress_{\text{HI}}(\mathbf{T}) \geq \sigma_s \tag{23}$$

The values of σ_s were adjusted such that task graphs under that line would be 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 (23) with a fixed step s. Per each target, ten experiments were run, generating the points lying near the target with a certain tolerance δ. All points satisfied $Stress_{\text{MIX}} \leq m$. 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). For EDF-DS we used a threshold of 0.8. All algorithms were 'mixed-criticality aware' in the sense that they used FPM for evaluating schedulability and the modified WCET-uncertainty aware deadlines; see Section 6.3 for the details. From the result we can see that MCPI gives a significant improvement in schedulability compared to the support algorithm, reaching a maximum of 30.83%.

Fig. 27 and Fig. 28 give the contour graph of the distribution of the generated points in grayscale, where black is the maximum value and white is 0. The horizontal axis is $Stress_{\text{LO}}$, the vertical is $Stress_{\text{HI}}$. Figures from Fig. 27(a) to Fig. 27(d) refer to the experiments made for 2 processors. In particular Fig. 27(a) shows the distribution of the generated task graphs, Fig. 27(b) shows the distribution of instances schedulable by EDF-DS among the generated ones. Likewise Fig. 27(c) shows the distribution of task graphs schedulable by MCPI (EDF-DS) and Fig. 27(d) shows the distribution 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. 27(d) is particularly



(a) Region where problem instances were generated, 50500 instances in total, uniformly distributed in the region.

(b) Distribution density of instances schedulable by EDF-DS[0.8], 6887 instances in total

(c) Distribution density of instances schedulable by MCPI , 8310 instances in total

(d) Density of instances schedulable by MCPI and not by EDF-DS[0.8], 1423 instances in total

Fig. 28 The contour graphs of random task graphs for 4 processors. The horizontal axis is $Stress_{\mathrm{LO}}$, the vertical is $Stress_{\mathrm{HI}}$.

interesting, because it shows how MCPI increases its effectiveness over the support algorithm when the load increases. Note that approximately around point $(1.7, 1.7)$ the distribution density is the highest, suggesting that around this point MCPI is the most effective.

Figures from Fig. 28(a) to Fig. 28(d) show respectively the same information of figures from Fig. 27(a) to Fig. 27(d), but referred to experiments on 4 processors. From those graph we have confirmation of the conclusions made above. Also in Fig. 28(d) we have an area where MCPI is particularly effective, approximately around point $(3.3, 3.1)$.

8.2.1 Comparison with Audsley approach on multiple processors

One of the main motivations for this work was to overcome the limitations of Audsley approach, as discussed in Section 3.1. In this section we show through experiments that we successfully reached our goal, by comparing MCPI with Audsley approach.

First we compared with two 'ideal' implementations of Audsley approach, named AUD_FP and AUD_FPM. In both of them we implemented an exact version of function $GetTerminationTime$ of Fig. 4. To achieve this, to test if we can assign the least priority to a job J, we compute its worst case termination time by simulating all possible combinations of relative priority of jobs with higher priority. In AUD_FP we assumed to use a fixed priority scheduling, *i.e.,* we did not consider the possibility of dropping LO jobs. Thus when computing the worst case termination time we simulate a scenario where all jobs J_i run for $C_i(\mathrm{HI})$. Note that this algorithm is equivalent to OCBP on single processor. Conversely in AUD_FPM we assumed to use a fixed priority per mode scheduling, *i.e.,* we

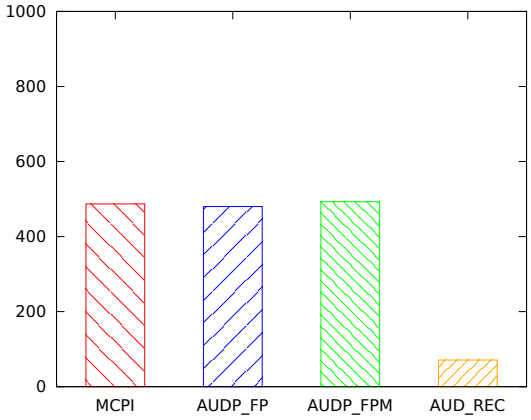


Fig. 29 Comparison of MCPI with Audsley approach

drop LO jobs when a mode switch happens. In this case when computing the worst case termination time we simulate all basic scenarios for all possible combination of job priorities. Note that both algorithms have an exponential complexity and easily become computationally intractable.

Finally we also compared the results using a computationally tractable termination-time estimations in Audsley approach, AUD_RC, which is based on Formula (8) given in Section 3.1.

We tested the algorithms on 1000 randomly generated instances of 8 jobs on 2 processors. The instances had a target $Stress_{\mathrm{LO}} = Stress_{\mathrm{HI}} = 1.8$. MCPI Solved 487 instances, AUD_FP 480, AUD_FPM 493 and AUD_RC only 71. The experimental results are shown in Fig. 29. Note that if we could, similarly to MCEDF, perform splitting[9] we would probably see the advantage the mode-switched MCPI and AUD_FPM versus the mode-agnostic AUD_FP, be analogy to MCEDF versus OCBP. For the small-size instances which we could practically evaluate this advantage is not visible, hypothetically due to non-uniform WCET uncertainty of a few random jobs. Also, the experiments show that MCPI does not strictly dominate AUD_FP as in some cases the exhaustive exploration of priority tables can help the latter to better deal with the Dhall effect.

Even if the experiments were not extensive, due to the computational complexity of AUD_FP and AUD_FPM, we can state that MCPI behaves "competitively" to an "ideal" implementation of Audsley approach that is computationally intractable, and clearly outperforms its "reasonable" implementation. This demonstrates superiority of our P-DAG insertion-sorting based improvement compared to what Audsley approach can currently offer.

9 Conclusions

In this paper we studied the problem of fixed job priority scheduling of dual criticality systems. Our study was focused on finite sets of job instead of more general task models. We motivate this by the observation that reasoning on tasks is more complex, and thus the finite job set model, when applicable, potentially allows for better processor utilization. It also enables support of precedences between jobs with different arrival times and deadlines.

We have considered one of the most common approaches to schedule mixed critical systems: Audsley approach. This approach has, however, some serious limitations for mode switched scheduling and multiprocessors. To overcome these limitations we introduced new priority assignment heuristics. To equip these heuristics with means of formal reasoning of how jobs influence each other we introduced the concepts of *Priority-DAGs* (P-DAGs) and *potential interference relation*. We oriented our heuristics towards *Fixed Priority per Mode* (FPM) scheduling policy, as an improvement over the usual *Fixed Priority* (FP) policy, commonly assumed in Audsley approach. Unlike FP, FPM policy aborts

[9] in this case we would need to put precedence edges between sub-jobs

non-critical jobs in emergency mode (mode HI) and performs better when the WCET uncertainty gets more uniformly distributed between the jobs *e.g.,* due to job splitting.

In Section 5, we presented the *Mixed Critical EDF* (MCEDF), an FPM algorithm for single processor. This algorithm was compared with the *Own Criticality Based Priority* (OCBP) algorithm, which is the optimal FP algorithm, based on Audsley approach. We formally proved dominance of MCEDF over OCBP, and hence over all FP algorithms, and showed its better algorithmic complexity than that of a direct implementation of OCBP.

The *Mixed Criticality Priority Improvement* (MCPI) algorithm was then discussed in Section 6. It is an FPM scheduling algorithm that supports multiprocessors and job precedences. To the best of our knowledge, in the literature there are no other multiprocessor mixed-criticality algorithms that support precedences, if not under very restrictive assumptions.

In Section 7 we show some theoretical properties that are common for MCEDF and MCPI. We formally proved that when applied on single processor with no precedences they both are optimal among the FPM policies that keep the priorities safety-critical jobs in relative EDF order. From the above property, we also deduce their equivalence.

We concluded the paper with experimental results (Section 8) where we show noticeable improvement over OCBP and other heuristics using randomly generated benchmarks. We also showed an empirical comparison between MCPI and Audsley approach on multiprocessors, from where we deduce that our approach gives comparable results to the "ideal case" of Audsley approach, *i.e.,* the case where it would dispose of an exact bound on job termination times. We also showed that a much more precise bound is required that the one that could be directly deduced from the literature.

In future work we are planning to extend MCEDF to precedence constraints and to prove equivalence with MCPI also in this extended case. MCPI currently uses a weak method to estimate the potential interference relation. Finding a more precise technique should increase its performance. A related issue is finding good upper bounds of termination times for Audsley approach. Also, an alternative way of handling Dhall effect beyond density separation would be desirable.

Finally, we would like to extend both algorithms to non-preemptive case and to multiple levels of criticality. The latter is important for most standards, like DO-178B, but addressing it is not trivial.

Acknowledgements We would like to thank Prof. Sanjoy Baruah for valuable discussions, especially for a hypothesis that lead to establishing improved algorithmic complexity result for MCEDF. We would also like to thank the reviewers for providing very constructive remarks that helped us to correct some important errors and improve the quality.

References

[Aud93] N.C. Audsley. *Flexible Scheduling in Hard-Real-Time Systems.* PhD thesis, Dept. of Computer Science, Univ. of York, 1993.

[Bar04] Sanjoy K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Trans. Comput.*, 53(6):781–784, June 2004.

[Bar12a] Sanjoy Baruah. Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In *RTNS '12*, pages 11–19. ACM, 2012.

[Bar12b] Sanjoy Baruah. Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In *RTNS'12*, pages 11–19. ACM, 2012.

[Bar13] Sanjoy K Baruah. Implementing mixed criticality synchronous reactive systems upon multiprocessor platforms. *The University of North Carolina at Chapel Hill, Tech. Rep*, 2013.

[BB06] Sanjoy K. Baruah and Alan Burns. Sustainable scheduling analysis. In *Real-Time Systems Symposium (RTSS 2006)*, pages 159–168, 2006.

[BB17] Lalatendu Behera and Purandar Bhaduri. Time-triggered scheduling of mixed-criticality systems. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4):74:1–74:25, 2017.

[BBD+12a] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Euromicro Conf. on Real-Time Systems*, ECRTS'12, pages 145–154. IEEE, 2012.

[BBD+12b] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Trans. Comput.*, 61(8):1140 –1152, aug. 2012.

[BCLS14] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[BF05] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 9 pp.–329, Dec 2005.

[BF11] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium*, RTSS '11, pages 3–12. IEEE, 2011.

[BLS10] Sanjoy K. Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium*, RTAS'10, pages 13–22. IEEE, 2010.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), October 2011.

[DL78] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[EY12] Pontus Ekberg and Wang Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Euromicro Conf. on Real-Time Systems*, ECRTS'12, pages 145–154. IEEE, 2012.

[F+10] Julien Forget et al. Scheduling dependent periodic tasks without synchronization mechanisms. In *RTAS'10*, pages 301–310, 2010.

[GESY11] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium*, RTSS'11, pages 13–23. IEEE, 2011.

[Guo16] Zhishan Guo. *Real-time Scheduling of Mixed-critical Workloads upon Platforms with Uncertainties*. PhD thesis, University of North Carolina, 2016.

[HKM+12] Jonathan L Herman, Christopher J Kenna, Malcolm S Mollison, James H Anderson, and Daniel M Johnson. Rtos support for multicore mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 197–208. IEEE, 2012.

[HL94] Rhan Ha and J. W S Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. Int. Conf. Distributed Computing Systems*, pages 162–171, Jun 1994.

[Joh92] Leslie A. Johnson. DO-178B: Software considerations in airborne systems and equipment certification. In *Radio Technical Commission for Aeronautics.*, RTCA, 1992.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, dec 1999.

[KPSB18a] Rany Kahil, Peter Poplavko, Dario Socci, and Saddek Bensalem. Predictability in mixed-criticality systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2018 IEEE 24th International Conference on*. IEEE, 2018.

[KPSB18b] Rany Kahil, Peter Poplavko, Dario Socci, and Saddek Bensalem. Predictability in mixed-criticality systems. Technical Report TR-2018-8, Verimag, July, 2018. (Extended version of the RTCSA-18 paper).

[KSPB18] Rany Kahil, Dario Socci, Peter Poplavko, and Saddek Bensalem. Algorithmic complexity of correctness testing in MC-scheduling. In *RTNS '18*, pages 180–190, New York, NY, USA, 2018. ACM.

[LB10] Haohan Li and Sanjoy Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Intern. Conf. on Embedded Software*, EMSOFT '10, pages 99–108. ACM, 2010.

[LB12] Haohan Li and Sanjoy K. Baruah. Outstanding paper award: Global mixed-criticality scheduling on multiprocessors. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012*, 2012.

[Liu00] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, Inc., 2000.

[MEA+10] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Int. Conf. Computer and Information Technology*, CIT '10, pages 1864–1871. IEEE, 2010.

[Pat12] Risat Mahmud Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 309–320. IEEE, 2012.

[PK11] Taeju Park and Soontae Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Intern. Conf. on Embedded software*, EMSOFT '11, pages 253–262. ACM, 2011.

[Soc16] Dario Socci. *Scheduling of Certifiable Mixed-Criticality Systems*. PhD thesis, VERIMAG Research Center, Université Grenoble Alpes, 2016.

[SPBB13] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 93–102. IEEE, 2013.

[SPBB15a] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Multiprocessor scheduling of precedence-constrained mixed-critical jobs. In *IEEE ISORC 2015*, 2015.

[SPBB15b] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Time-triggered mixed-critical scheduler on single- and multi-processor platforms (revised version). Technical Report TR-2015-8, Verimag Research Report, 2015.

[SPBB15c] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Time-triggered mixed-critical scheduler on single-and multi-processor platforms. In *17th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2015.

[Ves07] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium*, RTSS'07, pages 239–243. IEEE, 2007.

[YKRB14] Eugene Yip, Matthew Kuo, Partha S Roop, and David Broman. Relaxing the Synchronous Approach for Mixed-Criticality Systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.