2013 25th Euromicro Conference on Real-Time Systems

Mixed Critical Earliest Deadline First

Dario Socci, Peter Poplavko, Saddek Bensalem and Marius Bozga
UJF-Grenoble 1
CNRS VERIMAG UMR 5104,
Grenoble, F-38041, France
{*Dario.Socci* | *Petro.Poplavko* | *Saddek.Bensalem* | *Marius.Bozga*}*@imag.fr*

***Abstract*—Using the advances of the modern microelectronics technology, the safety-critical systems, such as avionics, can reduce their costs by integrating multiple tasks on one device. This makes such systems essentially *mixed-critical*, as this brings together different tasks whose safety assurance requirements may differ significantly. In the context of mixed-critical scheduling theory, we studied the dual criticality problem of scheduling a finite set of hard real-time jobs. In this work we propose an algorithm which is proved to dominate OCBP, a state-of-the-art algorithm for this problem that is optimal over fixed job priority algorithms. We show through empirical studies that our algorithm can reduce the set of non-schedulable instances by a factor of two or, under certain assumptions, by a factor of four, when compared to OCBP.**

I. Introduction

Mixed-critical systems (MCS) integrate tasks with significantly *asymmetric* safety requirements on a single assemble of processing resources. For example, in avionics systems, different maximal tolerated error counts range from 10^{-9} per hour for the autopilot to 10^{-3} for the communication during the flight. Mixing the asymmetric safety requirements is of a significant importance for the scheduling of mixed-critical tasks on modern microelectronic devices, because the hardware technology improvements enable low cost and low weight by integrating exponentially growing (in transistor count) amount of processing power on a single device – a system-on-chip.

These technological developments and ever growing importance of embedded computers in avionics and other safety-critical areas have called into existence a special mixed-critical (MC) real-time (RT) scheduling theory, that has been developed at least since 2007 [1]. This theory is distinguished by treating the asymmetric safety requirements by adequate scheduling methods, which lead to much more efficient resource usage compared to classical scheduling approaches [2]. In particular, MCS-aware scheduling methodologies were demonstrated in [3] to significantly outperform traditional pragmatic approaches such as reservation-based techniques. The latter is a widely adopted approach in safety-critical systems that has an important disadvantage in that it provides a *symmetric isolation* in timing or space, being redundant in that it *equally* isolates not only high-critical from low-critical tasks but also *vice versa*. Differently from this, the new theory performs a paradigm shift towards *asymmetric isolation*, as pointed out in [2]. Some previous literature achieves this goal by on-demand best-effort priority switching in favor of highly critical tasks [2], others assume that lower-critical tasks are soft real-time [4].

In this paper, we follow an important major branch of the MCS scheduling theory, the *certification-cognizant* mixed-critical scheduling, assuming that all tasks are hard real-time and assuming the certification prescribed by safety-critical standards such as DO-178B[5]. Although this approach tries to follow these prescriptions in a rather simple pragmatic way, it faces NP-complete problems even under basic assumptions [3]. In particular, simple polynomial *fixed job priority* scheduling policies, such as EDF [6] (earliest-deadline-first), do not guarantee optimal schedulability. This is unfortunate, because a significant part of theory and practice (RTOS) is dedicated to supporting such policies, and therefore they remain of big importance for MCS. The Own-Criticality Based Priority (OCBP) [7] is theoretically the best among all fixed job priority scheduling algorithms for MCS. Recent extensions of the fixed job priority policy [8], [9] perform a switch between different priority tables for different modes, showing *empirically* a significant improvement of schedulability over fixed job priority, while still reusing a lot in terms of implementation and theory from the classical priority-based scheduling. However, to the best of our knowledge none of such extended fixed job priority policies has ever been *theoretically* proven to dominate OCBP, despite their extra degree of freedom – to switch the priority tables.

Our main contribution is to fill this gap by proposing a new mode-switched priority assignment algorithm – *mixed critical EDF (MCEDF)* – that we theoretically prove to dominate OCBP. Another contribution is a thorough empirical evaluation of the two algorithms through extensive simulations. Previous works, in fact, evaluate OCBP only by specifying sufficient analytical conditions for schedulability [3] or by running experiments only for periodic jobs [9]. Our empirical comparison to OCBP suggests that the probability of failing to find a correct schedule reduces by roughly one half when applying MCEDF. Finally our last contribution is to show how one can improve the schedulability of the mode-switched policies even further by splitting jobs into smaller subjobs. The empirical results optimistically predict that the probability of schedulability failure can further be halved after such a load-preserving transformation, while mode-unaware algorithms like OCBP cannot take any advantage of it by construction.

Following a significant volume of previous MC scheduling work (*e.g.,* [3], [7], [10], [11]) in this paper we do not work directly with periodic/sporadic-task models and consider the basic problem of single-core scheduling for a finite set of jobs whose exact arrival times are known *a priori*. As argued

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 288175 [Certainty].

1068-3070/13 $26.00 © 2013 IEEE
DOI 10.1109/ECRTS.2013.20

in [10], this assumption applies without restrictions when generating schedules for *time-triggered architecture*, in this case one can just apply a finite job algorithm, like the one presented in this paper, to a hyperperiod. Moreover, when applied at run-time, it can be readily imported into computing the dynamic priorities for the jobs of sporadic tasks [12]. Therefore, this approach is worth considering in many practically relevant RT scheduling contexts.

This paper is organized as follows. In Section II we introduce the MC scheduling problems, giving the problem definition and a basic taxonomy. In Section III we present a new algorithm, show its dominance over OCBP, discuss the characterization of schedulability. The experimental results presented in Section IV evaluate how often OCBP may fail to schedule an instance while MCEDF can. Section V compares the MCEDF to related work. Section VI summarizes the paper and indicates the directions for future work.

II. Scheduling in Mixed Critical Systems

A. Background

Consider a set of hard real-time jobs where different jobs have different *levels of criticality*, *i.e.,* different levels of tolerated error rate. A common approach is to model different criticality requirements by giving different worst-case execution times (WCETs) for the same job. We consider *dual-criticality* systems, having two levels of criticality, the high level, denoted as 'HI', and the low (normal) level, denoted as 'LO'. Every highly critical 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 certification [1]. One important remark is that both HI and LO jobs are hard real-time, so both *must* complete 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, calculated by very pessimistic certification tools. This necessity thus comes from certification needs (*i.e.,* legal constraints) and not from engineering considerations. For this reason, upon the hypothetical event in which some jobs violate their LO WCET, the scheduling policy tolerates that the LO jobs may miss their deadlines or even drops them altogether, in order to certify the HI jobs at the least processor capacity requirements. This approach for mixed-critical RT systems is called *certification-cognizant* [3], [10].

Even two criticality levels are enough to elevate the complexity of the classical uniprocessor scheduling problem from polynomial to NP-complete [3]. Dual-criticality system are of practical interest, since certain safety-critical application domains can be classified as such. For instance in the *unmanned aerials vehicle* (UAV's) domain, functionalities are divided into *mission-critical* functionalities and *flight-critical* functionalities, and only the latter undergo certification [7]. Generalization of the proposed algorithm to more criticality levels is future work.

B. MC Scheduling Formalism

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

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

The index j is technically necessary to distinguish between the jobs with the same parameters. We assume that $C_j(\text{LO}) \leq C_j(\text{HI})$. We also assume that the LO-criticality 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* $\mathbf{J}$ of the MC-scheduling problem is a set of K jobs with indexes $1 \ldots K$. A *scenario* of an instance $\mathbf{J}$ is a vector of execution times of all jobs: $(c_1, c_2, \ldots, c_K)$. If at least one c_j exceeds $C_j(\text{HI})$, the scenario is called *erroneous*. The *criticality of scenario* $(c_1, c_2, \ldots, c_K)$ is the least critical χ such that $c_j \leq C_j(\chi), \quad \forall j \in [1, K]$. A scenario is *basic* if for each $j = 1, \ldots, K$ either $c_j = C_j(LO)$ or $c_j = C_j(HI)$.

A (preemptive) schedule of a given scenario is a mapping from physical time to $\mathbf{J} \cup \{\epsilon\}$, where ϵ denotes no job. Every job should start at time A_j or later and run for no more than c_j time units. The online state of a run-time scheduler at every time instance consists of the set of completed jobs, the set of *ready jobs*, *i.e.,* jobs that have arrived in the past and did not complete yet, the progress of ready jobs, *i.e.,* how much each of them has executed so far, and the current criticality mode, χ_{mode}, initialized as $\chi_{mode} = \text{LO}$ and switched to 'HI' as soon as a HI job exceeds $C_j(\text{LO})$. A schedule is *feasible* if the following conditions are met:

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

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

An instance $\mathbf{J}$ is *clairvoyantly schedulable* if for each non-erroneous scenario, when it is known in advance (hence *clairvoyantly*), one can specify a feasible schedule.

Based on the online state, a *scheduling policy* deterministically decides which ready job is scheduled at every time instant. A scheduling policy is *optimal* (or *correct*) for the given instance $\mathbf{J}$ if for each non-erroneous scenario it generates a feasible schedule. We assume without loss of generality that the scheduling policies are *monotonic*, *i.e.,* never postponing any jobs when getting less workload. One can check optimality of such policies by simulating them for all basic scenarios. A *mode-switched* scheduling policy uses χ_{mode} in the scheduling decisions, *e.g.,* to drop the LO jobs, otherwise it is *mode-ignorant*. An instance $\mathbf{J}$ is *MC-schedulable* if there exists an optimal scheduling policy for it. A *fixed-priority* scheduling policy is a mode-ignorant monotonic 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 selects the highest-priority job in PT. If a scheduling policy cannot be defined by a static priority table, it is called *dynamic-priority*.

C. Optimal Priority Assignment

For the 'ordinary' non-MC scheduling, the fixed priority policy EDF (earliest-deadline first) is optimal for any schedulable instance [6]. In the MC scheduling, for some schedulable instances no fixed priority tables PT are optimal (we will see examples later). Nevertheless, when an optimal PT exists, it will be computed by the *own criticality-based priority* (OCBP) algorithm [7]. It is based on the so-called "Audsley approach" [13], where the priorities are assigned by repeatedly assigning the least priority (*i.e.,* the last position in PT) to a job that will meet its deadline even when executing at the least priority. The job that has got the least priority assigned is removed from the working set of jobs, as it has no impact on the behavior of the higher-priority jobs. This procedure is repeated in multiple steps until no jobs remain in the working set. If at some step, no job can be selected for the least priority in the set then the instance is considered non-schedulable.

OCBP selects the least-priority job J_i using the following criterion: when having the least priority in the working set, job J_i still meets its deadline in the scenario $c_j = C_j(\chi_i), \forall j$, *i.e.,* the basic scenario with the WCET at the level that is '*own*' for J_i. This can be checked by simulating[1] the scheduling 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 [3]:

Lemma 2.1: The execution time available for 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.

Example 2.1: Let $\mathbf{J}$ be described by the following table:

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

At the first step OCBP tries to find a job to assign the least priority. We check job J_1 first. We simulate the execution of $\mathbf{J}$ assuming that J_1 has the least priority, under the hypothesis that every job executes for its C(LO) execution time (since $\chi_1 =$ LO). At time 0, only J_3 is ready, so it executes for 1 time unit. Then the CPU is idle for 2 time units, until at time 3 J_1 and J_2 arrive. J_2 has higher priority, so it executes for 1 time unit, completing its execution at time 4. We can now schedule J_1, but it already missed its deadline. So now we check whether job J_2 can have the least priority instead. Since $\chi_2 =$ HI, J_3 now has a WCET of 4. At time 0, J_3 is scheduled, switching to HI mode at time 1. At the mode switch the fixed priority policy, assumed by OCBP, keeps the same priority table and does not drop any LO jobs. After the switch, J_3 executes for 2 time units more until at time 3 J_1 and J_2 arrive. Since J_2 has the least priority, at time 3 only J_1 and J_3 compete for the CPU. J_3 and J_1 will then execute for a total of 2 time units, completing at time 5. In this case J_2 will miss its deadline. We then check J_3 for the least priority. At time 0, J_3 will be scheduled and it will execute until time 3. Then we have to execute J_1 and J_2 for 2 time units. At time 5 we can schedule J_3 again, which will execute for another time unit, completing at time 6, thus meeting its deadline.

[1] [7] uses a more efficient procedure - *makespan*

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

Fig. 1. The MCEDF algorithm for computing priorities

At the second step we repeat a similar procedure for the working set $\mathbf{J}'$, $\mathbf{J}' = \mathbf{J} \setminus J_3$. If J_1 has less priority than J_2, the first possibility for it to start would be at its deadline time 4, so J_1 cannot have the least priority. But J_2 can be delayed by 1 time unit due to J_1. J_1 meets its deadline when it has the highest priority. Thus, we obtain the following priority table for $\mathbf{J}$: $PT = (J_1, J_2, J_3)$.

III. MCEDF Algorithm

A. Fixed Priority per Mode (FPM)

A fundamental limitation of the default fixed-priority scheduling is that it is by definition mode-ignorant, so it cannot change the priority of jobs or drop them when switching to the HI criticality mode. The algorithm proposed in this paper computes the priority table for the scheduling policy which we call *fixed priority per mode* (FPM). This (mode-switched) scheduling policy has two priority tables: PT_{LO} and PT_{HI}. The former includes all jobs. The latter includes only the HI jobs. As long as the current mode is LO, this policy performs the fixed priority scheduling according to PT_{LO}. After the switch to the HI mode, this policy drops all pending LO jobs and applies priority table PT_{HI}. (Optionally, the LO jobs can be still executed at the least priority if their tardy completion is still desired.)

One can always use the EDF priority assignment for PT_{HI} because scheduling after the mode switch is a single-criticality problem, for which the EDF is optimal. Therefore, the priority assignment reduces to computing PT_{LO}, in sequel denoted simply as PT.

B. MCEDF

Our proposed mixed-criticality earliest deadline first (MCEDF) algorithm computes a PT for FPM policy, see Figure 1.

Initially, we compute the schedulability of LO scenario in subroutine $LOscenarioFailure$, by running EDF. By optimality of EDF for single criticality level, if a job misses

1: **Algorithm:** $GeneratePriorityTree$
2: **Input:** **job instance J**
3: **Input:** **node** v_{parent}
4: **In/out:** **priority tree** G
5: $\mathbf{B} \leftarrow PartitionIntoBIs(\mathbf{J})$;
6: **for** all $BI \in \mathbf{B}$ **do**
7: $J^{\text{least}} \leftarrow SelectLeastPriorityJob(BI)$
8: $v_{child} \leftarrow (BI, J^{\text{least}})$
9: $G.V \leftarrow G.V \cup \{v_{child}\}$
10: **if** $v_{parent} \neq \emptyset$ **then**
11: $G.E \leftarrow G.E \cup \{(v_{parent}, v_{child})\}$
12: **end if**
13: $\mathbf{J}' \leftarrow BI \setminus \{J^{\text{least}}\}$
14: $GeneratePriorityTree(\mathbf{J}', v_{child}, G)$
15: **end for**

Fig. 2. The MCEDF algorithm for computing priority tree

the deadline, then the instance is not schedulable. Thus the algorithm establishes that MC schedulability Condition 1 (see Section II-B) is satisfiable, and it considers this condition as a constraint in the construction of the final solution. While satisfying Condition 1, the algorithm applies a best-effort heuristic to ensure Condition 2, *i.e.,* that the deadlines of all HI jobs are met in any basic HI scenario, trying to ensure that the priorities of the HI jobs are as high as possible.

To compute the final priority table, the MCEDF algorithm first calls subroutine $GeneratePriorityTree$, which generates constraints on the priorities. These constraints are represented by a directed graph G, which consists of one or more trees; for simplicity, we refer to G just as a *priority tree*. Every node in G associates to a unique job, and every edge to a 'strictly greater than' constraint between the job priorities, thus G defines a partial order on job priorities. In fact, any possible total order PT respecting this partial order would give equally good result, as shown later. It is easy to generate such an arbitrary total order, and we chose to employ the well-known $TopologicalSort$ procedure (see *e.g.,* [14]), which traverses the trees in G from the roots to the leafs while adding the visited nodes to PT. Finally, the subroutine $anyHIscenarioFailure$ evaluates whether Condition 2 is met. In this case the algorithm succeeds. The check is done by a simulation over the set of all basic HI scenarios where a particular HI job is the first in the schedule to switch to the HI mode and all the HI jobs that execute after the switch conservatively use the $C(\text{HI})$ execution times. Because the FPM policies, like MCEDF, can be shown monotonic and because the HI scenarios we check are the most conservative HI scenarios, $anyHIscenarioFailure$ thus indirectly covers a complete schedulability check over all possible HI scenarios. Combined with the algorithm's property that the computed PT is schedulable in the basic LO scenario, this implies that a successful MCEDF run ensures a correct solution to the MC scheduling problem.

The core of the algorithm, *i.e.,* generating priority tree G, performs the schedulability checks only in the basic LO scenario. In the remainder of this subsection we explain this procedure. Hereby, by default, we conservatively assume that all jobs execute using the $C(\text{LO})$ execution times exactly.

The subroutine $GeneratePriorityTree$ is defined in Figure 2. The algorithm is based on the concept of a *busy interval*. A busy interval for an (sub-)instance $\mathbf{J}'$ is a time interval in a priority-based schedule for $\mathbf{J}'$ that is a maximal time interval $(\tau_1, \tau_2]$ where the set of ready jobs is never empty. Note that the interval is half-open because the jobs arriving at time t count ready only for the time instances strictly later than t. It is obvious that neither the start time nor the length of a busy interval depends on the exact priority assignment, because the former corresponds to the earliest job arrival and the latter is equal to the sum of $C(\text{LO})$ of all jobs in the interval. By abuse of terminology, we apply the term 'busy interval' also to the subset of jobs running in that interval, and denote it BI. In general, a job set $\mathbf{J}'$ can be partitioned into multiple BI's, because some jobs in $\mathbf{J}'$ may arrive at or later than the end of a busy interval of some other jobs in $\mathbf{J}'$.

The priority tree construction algorithm splits the instance into BI's and selects the least priority job in each BI (see Figure 2, line 5). Observe that in a busy interval $(\tau_1, \tau_2]$, the selected job will complete at time τ_2. Let $J^{\text{late}}_{\text{LO}}$ and $J^{\text{late}}_{\text{HI}}$ be the latest deadline[2] 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 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. Let us now show that in a feasible problem instance this rule makes a choice that is feasible for the LO scenario. The choice of the least-priority job affects the schedulability of that job only. 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 unfeasible instance.

The priority tree G is formally defined as a directed graph $G = (V, E)$, where V is the set of nodes $v = (BI_v, J^{\text{least}}_v)$, where BI_v is a busy interval associated with node v and $J^{\text{least}}_v \in BI_v$ is the job selected in BI_v as defined above. This definition of a node v conforms with line 8 in Figure 2, where new priority tree nodes are constructed. As for the edges, any edge (v, v') by construction satisfies the property: $BI_{v'} \subset BI_v$ (see line 13). This implies that J^{least}_v has less priority than the $J^{\text{least}}_{v'}$ at all the children of v and, by induction, at the whole subtree below. Let $PP(J_j)$ denote the position of job J_j in PT *i.e.,* the larger numerically the less the priority. Let $v \prec_G v'$ denote that (v, v') is an edge in G. The priority constraints implied by priority tree G can be defined as:

$$v \prec_G v' \ \Rightarrow \ PP(J^{\text{least}}_v) > PP(J^{\text{least}}_{v'}) \qquad (1)$$

In general, the priority tree G has multiple subtrees whose root nodes correspond to the BI's of the complete problem instance $\mathbf{J}$. Subroutine $PartitionIntoBIs$ in Figure 2 splits

[2] for equal-deadline jobs we break the ties by selecting the job with minimal $C_j(\text{HI}) - C_j(\text{LO})$. This choice minimizes the *uncertainty* (see Section III-D). Other heuristics are also possible.

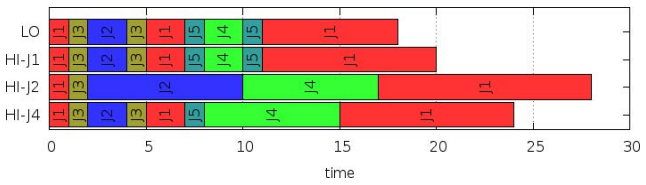


Fig. 3. The Gantt charts for Example 3.1 with $PT = (2, 4, 3, 5, 1)$

the currently examined instance into BI's. Then the subroutine $GenerateProrityTree$ examines every busy interval BI to select the least-priority job in that interval. 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 children of BI in the priority tree. 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 3.1: Let the instance $\mathbf{J}$ be defined by:

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

MCEDF computes the following solution $PT_{MC} = (2, 4, 3, 5, 1)$. Let us demonstrate MCEDF computations 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, Figure 3 row 'LO' shows a simulation for the PT_{MC}; no deadline is missed there, and hence the same should hold for PT_{EDF} as well.

Then MCEDF generates the priority tree, see Figure 4. Instance $\mathbf{J}$ has one busy interval BI. We can see this by the LO-scenario simulation in Figure 3, where job J_1 remains ready in interval $(0, 18]$, which implies that the processor is continuously busy until all jobs finish. Thus $BI = \mathbf{J}$ comes into the root of the tree. For this BI, $J^{\text{late}}_{\text{LO}} = J_5$ 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 tree root. Now we split the subinstance $\mathbf{J} \setminus \{J_1\}$ into BI's, obtaining $BI' = \{J_3, J_2\}$, running in $(1, 5]$ and $BI'' = \{J_5, J_4\}$, running in $(7, 11]$, selecting, 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 tree generation steps are trivial (see Figure 4). Priority table PT_{MC} satisfies the partial order of the resulting tree. Finally the algorithm runs the simulations for the HI scenarios, which deviate from the basic LO scenario by switching to the HI mode at some HI job J_j, as illustrated in rows 'HI-J_j' in Figure 3. Because, as the reader can verify, the deadlines are met, the algorithm succeeds.

Lemma 3.1 (Partial Priority Order): Given a partial priority order defined by the priority tree G as given in Equation (1). If G was generated by a successful run of MCEDF then any priority table PT compliant to this partial order will provide a correct solution for the FPM policy.

Proof: In any LO-scenario execution any two jobs that are not comparable by $\prec_G$ are never ready at the same time, being

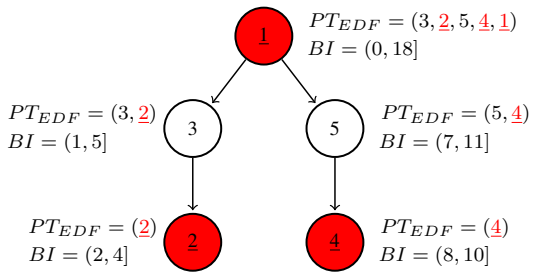


Fig. 4. The priority tree for Example 3.1; each node is annotated by the selected job index; underlined job indices indicate HI jobs.

located in different busy subintervals of the common parent in tree G. So the basic LO scenario schedule is exactly the same for any PT compliant to the partial order. At any possible switch to the HI mode the FPM policy switches from PT to the EDF priority for the HI jobs. Therefore, the selection of a PT compliant to $\prec_G$ has no impact on the schedule whatsoever in neither the LO mode nor the HI mode. Since it is given that for one compliant PT the procedure $anyHIscenarioFailure$ returns success, and hence the result would be the same for any other compliant PT. ■

Lemma 3.2 (MCEDF Complexity): MCEDF has an implementation with complexity $O(K^2 \log K)$ where $K = |\mathbf{J}|$.

Proof: Each fixed-priority schedule simulation, which is done once in $LOscenarioFailure$ and several times in $anyHIscenarioFailure$, sorts the jobs by the arrival times, in an $O(K \log K)$ time. Every simulation examines the jobs in the order of arrivals, adding each arriving job to the priority queue data structure [14]. At most once per job arrival or completion the greatest-priority job needs to be selected in the queue. Once per job completion, a job is removed from the queue. Because the addition, removal and selection operations are done $O(K)$ times and each priority-queue operation costs $O(\log K)$, the total cost of one simulation is $O(K \log K)$. Because $anyHIscenarioFailure$ performs $O(K)$ simulations, its complexity is $O(K^2 \log K)$.

Graph G, being a collection of trees, has K nodes and at most $(K - 1)$ edges. The complexity of $TopologicalSort$ for such graphs is $O(K)$ [14]. What remains to be shown is that the complexity of $GenerateProrityTree$ does not exceed $O(K^2 \log K)$. The most time-costly procedure at each node v is the partitioning of a subinstance $\mathbf{J}'_v = BI_v - \{J^{\text{least}}_v\}$ into busy intervals. The partitioning can be computed simultaneously for all nodes v at the same *tree level*, by the simulation of subinstance $\mathbf{J}'' = \bigcup_{v \in \text{Level}} \mathbf{J}'_v$ with arbitrary PT. In the simulation, whenever the set of ready jobs becomes empty the next jobs are assigned to a new BI. Now observe that all BI's together at each tree level contain at most K jobs, with exactly K jobs at the root level and removing some of them when going from the roots to the leafs. So, the tree generation cost is $O(K \log K)$ per level. Because there are at most K tree levels, the total tree generation complexity is $O(K^2 \log K)$. ■

C. Dominance over OCBP

In this subsection we provide a theoretical evidence that MCEDF dominates OCBP. Example 3.1 gives an MCEDF-schedulable instance which is 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, instance **J** consists of a single BI that finishes at time 18, when any LO job would miss its deadline. If we could select a HI job, then OCBP would evaluate its completion 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 completion time 18. But the completion time 31 is beyond the latest HI job deadline, $D_1 = 30$.

Thus, the dominance is given by the following:

Theorem 3.3: If an instance is OCBP schedulable, then it is schedulable by the MCEDF algorithm as well.

Proof: Recall that, by Lemma 3.1, the preference for one particular topological order of the priority tree does not impact the MCEDF 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 [7]. 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 **J**.

Let us first examine in detail how MCEDF constructs PT. At any step of the topological sort, there is a 'ready set' (RS), *i.e.,* the set of busy intervals $\{BI_i^{\text{RS}}\}$ of the priority tree nodes v_i that are not yet selected but whose parent node has already been selected. Implicitly, there is a sub-instance $\mathbf{J}'$ of which BI_i^{RS} are the busy intervals. MCEDF can choose to pick any BI to provide its J^{least} 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 complete 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 select a HI job. Let us denote this job J' and let J'^{least} be the HI job selected by MCEDF for the busy interval BI_i^{RS} where J' is located. Because MCEDF selects the latest-deadline HI job, we have: $J'^{\text{least}}.D \geq J'.D$.

  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 than the execution times in the basic LO scenario and no LO jobs are dropped we conclude that J' and J'^{least} must be located in the same busy interval also in the HI scenario. The fact that J' can be selected by OCBP means that if it completes 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', it is eligible to let J'^{least} complete 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 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.

D. Characterization

The characterization of a scheduling algorithm means defining certain metrics that estimate the schedulability of problem instances under different scheduling algorithms. These estimations are not always necessary to evaluate whether an algorithm can schedule an instance, as for a finite-job problem it can be more efficient to run the algorithm itself together with its built-in verification of the correctness of the solution, for example MCEDF does this verification in $LOscenarioFailure$ and $anyHIscenarioFailure$. In this context, the characterization metrics are only used as convenient indicators of algorithm performance, but not necessarily for a schedulability test.

One metric, proved useful for mixed-critical scheduling, is *speedup factor* [7]. A scheduling algorithm has a speedup factor s if any instance that is clairvoyantly schedulable on a unit-speed processor is also schedulable by the algorithm on a processor of speed s. We know from [7], [3] that OCBP has a speedup factor of $(\sqrt{5}+1)/2 \approx 1.62$, and that this value is optimal, *i.e.,* no non-clairvoyant scheduling algorithm can have a smaller speedup factor. From this observation and due to dominance of MCEDF over OCBP we derive the following:

Corollary 3.4: MCEDF has the optimal speedup factor $s = (\sqrt{5}+1)/2$

The most common and well-known characterization metric is the *utilization – i.e.,* the percentage of CPU cycles utilized by all tasks, but it is usually defined only for the infinite sets of jobs produced by periodic tasks, where the intervals between the releases of the jobs of the same task are equal. When the intervals between the jobs are arbitrary, the utilization generalizes to *load*, *i.e.,* the maximal ratio between the processing demand and the processing capacity. Baruah *et al.* [15] defined the load metrics for mixed-critical scheduling problems and applied them to analyze the OCBP algorithm. The authors determine the load a CPU can experience in a LO and in a

HI scenario as shown below:

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

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

An instance can only be schedulable if the processor is not overloaded. Hence, a *necessary* condition for MC schedulability is [15]:

$$Load_{\text{LO}}(\mathbf{J}) \le 1 \wedge Load_{\text{HI}}(\mathbf{J}) \le 1 \tag{2}$$

This is also a sufficient condition for clairvoyant scheduling, but not for the online policies, [15].

The following *sufficient* condition for an instance to be schedulable by the OCBP algorithm is proven in [15]:

$$Load^2_{\text{LO}}(\mathbf{J}) + Load_{\text{HI}}(\mathbf{J}) \le 1 \tag{3}$$

Due to the dominance over OCBP we have:

Corollary 3.5: Equation (3) is a sufficient condition for MCEDF schedulability.

The characterization above proved useful for the fixed-priority policy. However, we would like to stress 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. By definition, this knowledge can be exploited online only by mode-switched policies. The job WCET uncertainty can be measured as $\Delta C_j = C_j(\text{HI}) - C_j(\text{LO})$ (strictly positive only for the HI jobs). In [11] 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 [11] 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 as the one equal to $Load_{\text{LO}}$ after substituting D'_j into D_j [11]:

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

By the reasoning above, the *necessary* condition (2) can be refined to:

$$Load_{\text{MIX}}(\mathbf{J}) \le 1 \wedge Load_{\text{HI}}(\mathbf{J}) \le 1 \tag{4}$$

$Load_{\text{MIX}}$ is a better indicator of schedulability than $Load_{\text{LO}}$, especially for mode-switched policies. To demonstrate this, consider *splitting*, a theoretical transformation[3] 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 jobs, whose total execution times $C_j(LO)$ and $C_j(HI)$ 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 uncertainty and $Load_{\text{MIX}}$. Therefore, for mode-switched policies, such as MCEDF, the splitting can translate an unschedulable instance into a schedulable one. An infinitely large splitting of all HI jobs can bring the optimality of a mode-switched policy infinitely closer to that of the clairvoyant scheduling. For some instances, a finite splitting is enough to equate the clairvoyant scheduling. Mode-ignorant policies, such as OCBP, cannot take any advantage of splitting by construction. These observations are confirmed in our experiments in Section IV.

[3]we ignore the overhead incurred by such a transformation.

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 (4) 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

The scheduler can execute J_{22} until completion, 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, not showing any advantage of the split instance *w.r.t.* the original one.

Note that the splitting, even being a theoretical transformation, may have some practical significance. This depends on the WCET tools, in particular, by what extent the sum of WCETs may change by the splitting of code into blocks. Note that despite the fact that the arrival times of all subjobs 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 in a sequential priority-driven order and the sequential blocks of the job code can be assigned to the subjobs in the same order.

IV. Implementation and Experiments

We evaluated the schedulability performance of MCEDF relative to OCBP in experiments with randomly generated job instances of 20 jobs[4] with integer timing parameters, simulating CPU clock cycle count of some imaginary machine. Every job instance was generated for a target LO and HI load pair. The method to generate a job instance is described in [16].

We ran multiple job generation experiments, ranging each target of $Load_{\text{LO}}$ and $Load_{\text{HI}}$ 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 $Load^2_{\text{LO}}(\mathbf{J}) + Load_{\text{HI}}(\mathbf{J}) = 1$, yielding instances where OCBP could potentially fail. 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 5(a) gives the contour graph of the density of the generated points in grayscale. The grid follows the parabolic lines of equal $Load^2_{\text{LO}}(\mathbf{J}) + Load_{\text{HI}}(\mathbf{J})$. The total number of trials was 537460, whereas we failed to generate 2.7% (14427) of points due to the limitations of the job instance generation algorithm. All these missing points are at the right bottom corner in Fig 5(a).

Around 14% (77005) of points showed failure for OCBP. In those 14%, roughly 5% (28806) were not schedulable by MCEDF as well, whereas 9% (48199) were schedulable by MCEDF. Thus, MCEDF proved to reduce the set of non-schedulable instances by more than one half. The density distributions in Figure 5 suggest that MCEDF is less sensitive to high loads.

For the 5% (28806) non-MCEDF schedulable jobs we ran additional experiments. We split all HI jobs by factors $2, 3$, and 4. This kept the load the same but reduced the WCET uncertainty. After splitting the instances remained to be non-OCBP schedulable (as OCBP cannot take advantage of less uncertainty) but the number of non-MCEDF schedulable instances has reduced, coming to 3% (16991). So if we can accept this load-preserving transformation, we go from 14% non-schedulability of OCBP to the 3% non-schedulability of MCEDF. Note that 2% (11812) were gained due to the splitting, whereby in the most of cases, 1.5% (7877), split factor 2 was sufficient. So assuming that in practice we can split the HI jobs into a few sub-jobs such that both WCET values scale, then we can 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.

Figure 6 demonstrates an example of generated problem instance and its execution in MCEDF using Gantt charts. $J[\chi][i]$ defines the jobs of criticality χ by the intervals between arrival, earliest completion for the LO and HI WCETs, and deadline. Figure 6(b) shows the basic LO scenario and all basic HI scenarios where a particular HI job is the first to switch to the HI mode. The instance of Figure 6 is non OCBP-schedulable.

[4]The instance size was restricted due to the computation delays of job generation algorithm and our intention to evaluate a large number of points.

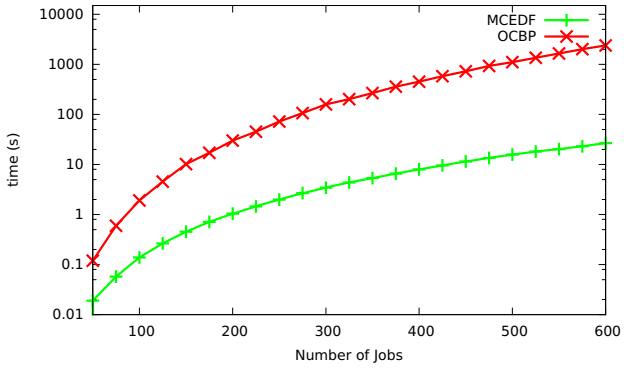


Fig. 7. The measured computation times of OCBP and MCEDF

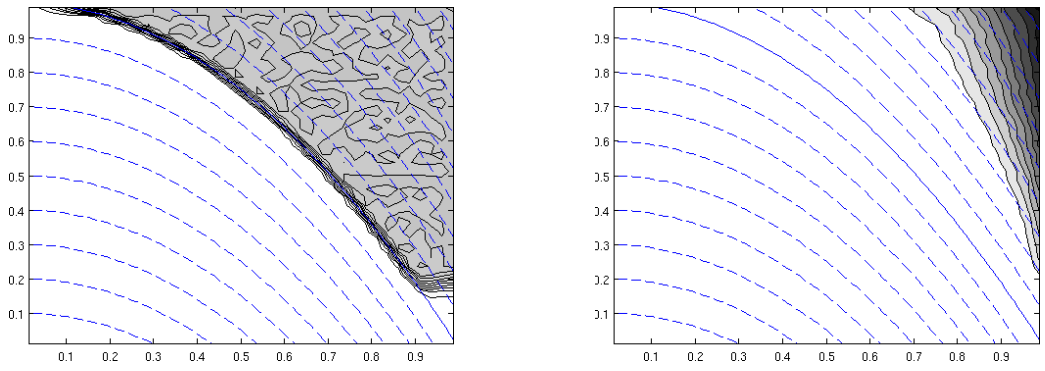
We also performed some experiments to evaluate the computation times of both algorithms, implemented using the same software library. Every point was obtained as the average computation time for 20 different randomly generated instances with $Load_{\text{LO}} = Load_{\text{HI}} = 0.8$. The results are shown in Figure 7. 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 $O(K^2 \log K)$, according to Lemma 3.2.

V. Discussion and Related Work

MCEDF uses the flexibility of fixed-priority per mode policies to be dominant over fixed priority algorithms. An important advantage of MCEDF over other non fixed-priority schedulers is that this algorithm can be implemented on existing systems that support fixed priority algorithms with minimal modification. In fact, if the MCEDF topological ordering proceeds according to the rules given in the OCBP dominance proof then one can trivially restrict it to select the latest deadline HI job whenever a HI job should be selected. Hence, the MCEDF will produce a PT where the HI jobs are placed in the EDF order relatively to each other. Consequently, instead of dropping the LO jobs and switching to the EDF priority table at the mode switch, we can get exactly the same result if we use a fixed-priority scheduler, just signalling to the LO jobs to speedily return if a mode switch occurs.

To the best of our knowledge, in the previous work no FPM algorithm [12], [8], [9] has been proven to be theoretically dominant over OCBP. The priority assignment of [12] applies OCBP to compute PT_{LO}, thus having equivalent schedulability. [8] proposes an efficient online algorithm with the optimal scaling factor and [9] presents a highly efficient priority computation method that dominates OCBP and several other algorithms empirically. Note, however, that [8], [9] are not directly applicable to the problem studied in this paper as they are designed for a periodic job model with unknown arrival times.

The FPM policy provides better results than fixed priority, but in general dynamic-priority policies are necessary for optimality. Fortunately, according to [3], also for these policies the time instances for job preemption can be restricted to job arrivals and the mode switch. Consider the following example instance $\mathbf{J}_d$:



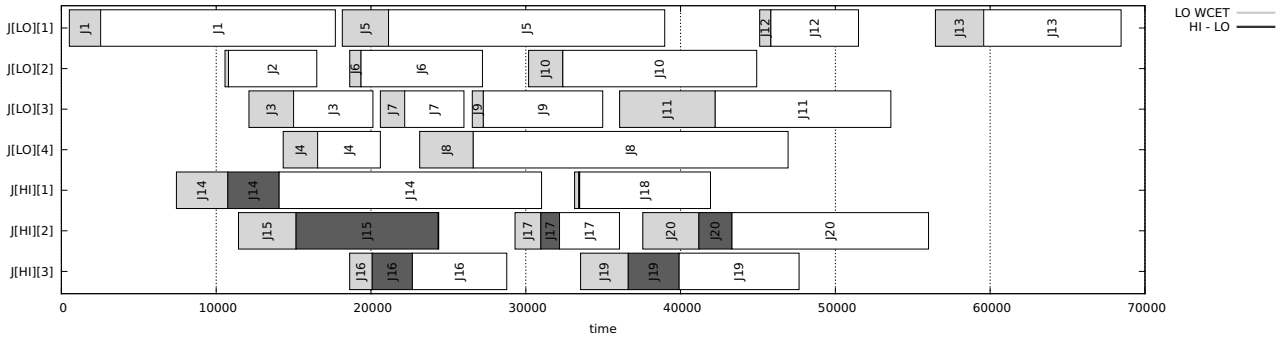
(a) Density of Generated Jobs

(b) non OCBP-schedulable 14%

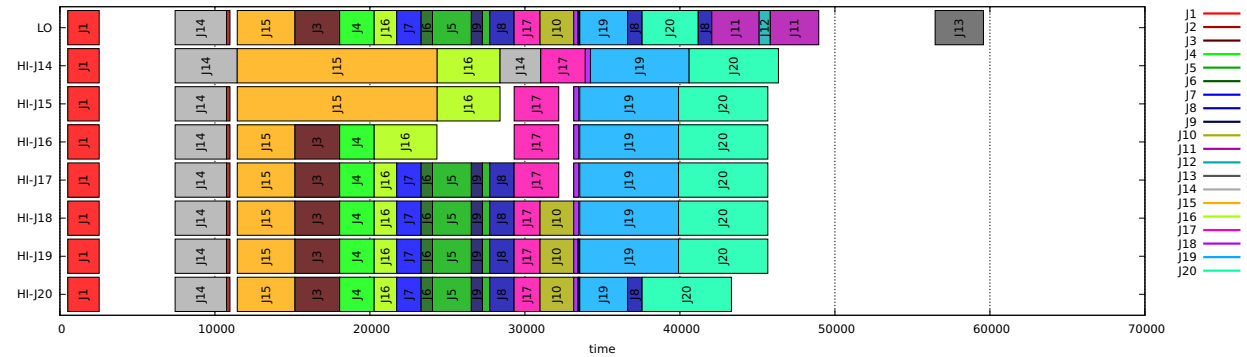
(c) non MCEDF-schedulable 5%

(d) non MCEDF-schedulable after split 3%

Fig. 5. The contour graphs of random instances; the horizontal axis is $Load_{\mathrm{LO}}$, the vertical is $Load_{\mathrm{HI}}$.



(a) Job instance, visualized by the intervals between arrival, earliest LO completion (light gray), earliest HI completion (dark gray) and deadline (white)



(b) MCEDF execution, PT=(J15, J3, J4, J16, J14, J17, J18, J19, J20, J2, J1, J7, J6, J9, J5, J10, J8, J12, J11, J13)

Fig. 6. The Gantt charts of a randomly generated job instance, $Load_{\mathrm{LO}} = 0.85$, $Load_{\mathrm{HI}} = 1.0$

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

No FPM policy would schedule $\mathbf{J}_d$, a dynamic-priority one is required. The only correct scheduling policy for $\mathbf{J}_d$ is to execute J_1 for 1 time unit, then J_2. If J_2 completes after 1 time unit, we execute J_3 and then J_1 again, otherwise we drop J_3 and execute J_1. It is easy to see that this is not an FPM schedule, as J_1 changes its priority *w.r.t.* J_3 in the LO scenario.

In [11], an idea for a dynamic-priority scheduling algorithm is proposed. Unfortunately, this idea was evaluated only indirectly, by claiming condition (4) sufficient for schedulability and checking this condition experimentally over a set of randomly generated instances. However, should this claim ever be true then (4) would be both a necessary and sufficient condition, and we would thus have a polynomial-to-compute exact check for an NP-complete decision problem, which already raises doubts about this claim. Indeed, this sufficiency claim is *erroneous* and (4) is only a necessary but not a sufficient schedulability check. The table below gives a counter-example for the sufficiency of (4):

Job	A	D	χ	C(LO)	C(HI)
1	0	20	LO	10	10
2	0	40	HI	5	10
3	0	40	HI	15	30

It is easy to check that this instance satisfies (4). Lemma 2 in [3] implies that if all jobs arrive simultaneously then the MC schedulability can be checked by enumerating all possible FPM priority assignments. If we choose J_1 as the highest-priority job then we will not have enough time to execute both J_2 and J_3 if they both execute in HI scenario. If we choose J_3, then J_1 would miss its deadline. And if we choose J_2 then if we execute J_1 next, J_3 will not have enough time for its HI WCET, while if we execute J_3, then J_1 will miss its deadline.

VI. Conclusions and Future Work

This paper presents a new real-time scheduling algorithm for mixed-critical systems, fitting into the context of a real-time scheduling approach that supports the formal certification of safety-critical systems. In this context, even the basic problem of uniprocessor scheduling for a finite number of jobs is NP-complete and cannot be solved in general by classical fixed job priority scheduling policies. Our scheduling algorithm can be implemented as a simple extension of the fixed job priority scheduling, enjoying the advantages of relative ease of implementation and analysis of such schedulers. We both prove *in theory* and demonstrate *in practice* that the proposed algorithm dominates and significantly outperforms an algorithm that is optimal among all basic fixed job priority scheduling algorithms for this problem. In addition, our algorithm can take advantage of reduced uncertainty about worst-case execution time per job that can result from fragmentation of jobs into smaller jobs.

In future work we plan to extend this algorithm for sporadic and time-triggered tasks and to introduce support for more than two levels of criticality. Also, it is necessary to investigate the mixed-critical scheduling of task graphs, where there are data dependencies between jobs. For this variant of scheduling problem, it is important to extend the research from single-core to multicore systems and to manage the access conflicts at shared memory and on-chip interconnection framework. Where purely analytical techniques would fall short due to complexity of the problem, we plan to apply compositional verification techniques to ensure hard-real time and safety guarantees. Also we plan to apply our methodology to real-life avionics applications.

References

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium*, RTSS'07, pp. 239–243, IEEE, 2007.

[2] D. d. Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *Real-Time Systems Symposium*, RTSS'09, pp. 291–300, IEEE, 2009.

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

[4] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *Int. Conf. Computer and Information Technology*, CIT '10, pp. 1864–1871, IEEE, 2010.

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

[6] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, 1973.

[7] S. K. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium*, RTAS'10, pp. 13–22, IEEE, 2010.

[8] 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, pp. 145–154, IEEE, 2012.

[9] P. Ekberg and W. Yi, "Bounding and shaping the demand of mixed-criticality sporadic tasks," in *Euromicro Conf. on Real-Time Systems*, ECRTS'12, pp. 145–154, IEEE, 2012.

[10] S. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *Real-Time Systems Symposium*, RTSS '11, pp. 3–12, IEEE, 2011.

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

[12] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems," in *Real-Time Systems Symposium*, RTSS'11, pp. 13–23, IEEE, 2011.

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

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

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

[16] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Mixed critical earliest deadline first," technical report TR-2012-22, Verimag, 2012.