Mixed Critical Earliest Deadline First

Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga

Verimag Research Report n^o TR-2012-22

September 2012

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

Mixed Critical Earliest Deadline First

Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga

September 2012

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 error probability requirements may differ by an order of magnitude 10^6. In the context of mixed-critical scheduling theory, we studied the problem of scheduling a finite set of jobs. In this work we propose an algorithm which is proved to dominate OCBP, one of the best scheduling algorithms for this problem. We show through empirical studies that our algorithm can reduce the set of non-schedulable instances by a factor of 2.5 or, under certain assumptions, by a factor of 4.5, when compared to OCBP.

Keywords: real-time, mixed critical, scheduling

Reviewers:

How to cite this report:

```
@techreport {TR-2012-22,
  title = {Mixed Critical Earliest Deadline First},
  author = {Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga},
  institution = {{Verimag} Research Report},
  number = {TR-2012-22},
  year = {2012}
}
```

1 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 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-priority per job* 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-priority scheduling algorithms for MCS. Recent extensions of the fixed-priority policy perform a switch between different priority tables for different modes, showing *in practice* a significant improvement of schedulability, while still reusing a lot in terms of implementation and theory from the classical fixed job priority scheduling [8, 9]. However, to the best of our knowledge none of such extended fixed-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 a schedulability failure 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 suggest 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 in [10], this assumption applies without restrictions when generating schedules for *time-triggered architecture*, an important major paradigm for designing safety-critical systems, in particular, in automotive and avionics application domains [12]. Moreover, when applied at run-time, it can be readily imported into computing the dynamic priorities for sporadic tasks [13]. Therefore, this approach is worth considering in many practically relevant RT scheduling contexts.

This paper is organized as follows. In Section 2 we introduce the MC scheduling problems, giving the

problem definition and a basic taxonomy. In Section 3 we present a new algorithm, show its dominance over OCBP, discuss the characterization of schedulability. The experimental results presented in Section 4 evaluate how often OCBP may fail to schedule an instance while MCEDF can. Section 5 compares the MCEDF to related work. Section 6 summarizes the paper and indicates the directions for future work.

2 Scheduling in Mixed Critical Systems

2.1 Background

Consider a set of hard real-time jobs where different jobs have different *levels of criticality*, *i.e.,* different tolerated levels of error rate. A common approach is to model different criticality requirements by giving different worst-case execution times (WCETs) for the same job. Indeed, one can imagine that a job's execution time can be bounded by, say, 20 Kcycles with a probability $(1 - 10^{-6})$ and by 30 Kcycles with a probability $(1 - 10^{-9})$. Usually the most safety-critical tasks of an embedded system should pass a so-called formal *certification* procedure, where one ensures the highest safety probabilities by using much more pessimistic WCET estimation algorithms than those normally be used in hard RT practice and by adding an extra safety margin into the computed WCET. Every highly critical job gets a vector of at least two WCET values, one for normal safety assurance, to assess the resource sharing with jobs that do not have the highest safety requirements, and the other one, a higher value, for extra high assurance, to ensure certification [1]. This approach for mixed-critical RT systems is called *certifiable* or *certification-cognizant* [3, 10], and it is this approach that we follow in this paper.

We consider *dual-criticality* systems, having two levels of criticality, the high level, denoted as 'HI', and the low level, denoted as 'LO'. Already two criticality levels elevate the complexity of the classical uniprocessor scheduling problem from polynomial to NP-complete [3]. Generalization of the proposed algorithm to more criticality levels is future work. 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, but he or she must 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 certification-cognizant scheduling lets the LO jobs miss their deadlines or even be aborted, which helps us to certify the timeliness of the HI jobs at the least resource requirements.

2.2 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
- $D_j \in \mathbb{Q}_+$ 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 mode* of a scenario $(c_1, c_2, \ldots, c_K)$ is the least critical χ such that $c_j \leq C_j(\chi),\ \forall j \in [1, K]$. A scenario is *basic* if for each $j = 1, \ldots, n$ there exists $l_j \leq \chi_j$ such that the execution time of J_j is exactly $C_j(l_j)$.

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 no earlier than A_j 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 their LO WCET, then both critical and non-critical jobs must complete before their deadline.*

Condition 2. *If at least one job runs at its HI WCET, then all critical jobs must complete before their deadline.*

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 instance. 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 i.e.,* those whose execution times are a WCET at one of the levels, *i.e.,* where $c_j \in \{C_j(\text{LO}), C_j(\text{HI})\}$ [3]. A *mode-switched* scheduling policy uses χ_{mode} in the scheduling decisions, otherwise it is *mode-ignorant*. An instance $\mathbf{J}$ is *MC-schedulable* if 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 no such PT exists, the scheduling policy is called *dynamic-priority*.

2.3 Optimal Priority Assignment

For the 'ordinary' non-MC scheduling, the fixed priority policy is sufficient and the EDF (earliest-deadline first) priority assignment algorithm 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) [7]. 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" [14], where the priorities are assigned by repeatedly assigning the lowest priority to a job that can be conservatively proven to meet its deadline even when executing at the lowest priority. The job that has got the lowest priority assigned is removed from the working set of jobs, as it has no impact on the behavior of the higher-priority jobs, and then the process is repeated until no jobs remain in the working set. If at some step, no job can be selected for the lowest priority in the set, then the instance is considered non-schedulable.

OCBP selects the lowest-priority job J_i using the following criterion: when having the lowest priority in the working set, job J_i still meets its deadline in the scenario $c_j = C_j(\chi_i)$, *i.e.,* the basic scenario with the WCET at the level that is '*own*' for J_i. This is checked by simulating the scheduling with *any* priorities for the other jobs in the subset provided that they are higher than J_i. The correctness 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.*

The following example shows how OCBP works:

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

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

At the first iteration OCBP tries to find a job to assign the lowest priority. We check job J_1 first. We simulate the execution of $\mathbf{J}$ assuming that J_1 has the lowest priority, under the hypothesis that every job executes for its $C(LO)$ (since $\chi_1 = LO$). At time 0, only J_1 and J_4 are ready. So J_4 executes for 1 time unit and then J_1 executes for 2 time units. At time 3 J_1 completes meeting its deadline. So the lowest priority can be assigned to J_1.

In the next step we reiterate the algorithm on the instance $\mathbf{J}' = \mathbf{J} \setminus J_1$. We first check J_2. At time 0, only J_4 is ready, so it executes for 1 time unit. Then the CPU is idle for 1 time unit, until at time 3 J_2 and J_3 arrive. J_3 has higher priority, so it executes for 1 time unit, completing its execution at time 4. We can now schedule J_2, but it already missed its deadline. So now we check whether job J_3 can have the lowest priority instead. Since $\chi_3 = HI$, J_4 now has a WCET of 4. At time 0, J_4 will be scheduled, and it will execute for 3 time units. At time 3 J_2 and J_3 arrive. The fixed priority policy, followed by OCBP, keeps the same priority of jobs in the HI mode and does not drop the LO job J_2. Since J_3 has the lowest priority, at time 3 only J_2 and J_4 compete for the CPU. J_4 and J_2 will then execute for a total of 2 time units (for lemma 2.1 we do not care about their order), terminating at time 5. In this case J_3 will miss its deadline. We then check J_4 for the lowest priority. At time 0, J_4 will be scheduled and it will execute until time 3. Then we have to execute J_2 and J_3 for 2 time units. At time 5 we can schedule J_4 again, that will execute for another time unit, terminating at time 6, thus meeting its deadline.

At the third iteration we will do the same for instance $\mathbf{J}'' = \mathbf{J}' \setminus J_4$. By the same reasoning as above, job J_2 cannot have the lowest priority also in this case. But J_3 can be delayed by 1 time unit due to J_2. J_2 meets its deadline when it has the highest priority. Thus, we obtain the following priority table for $\mathbf{J}$:

$$PT = (J_2, J_3, J_4, J_1)$$

3 MCEDF Algorithm

3.1 Fixed Priority per Mode

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 proposed algorithm 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 lowest 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.

3.2 MCEDF

Our proposed mixed-criticality earliest deadline first (MCEDF) algorithm computes the priority table PT for FPM. An outline is given in Figure 1. Initially, we order the jobs in the EDF order (this is done by subroutine $JobsOrderedByEDF$). We first check schedulability of the basic LO scenario of the initial priority order PT. By optimality of EDF for single priority, if a job misses a deadline, then the instance is not schedulable. Past this point, the initial PT satisfies Condition 1 and this remains invariant of the algorithm, whereas it calls subroutine $ImproveHIJobs$, which is a best-effort procedure for trying to satisfy Condition 2. This is done by increasing the priority of HI jobs. Finally, the subroutine $anyHIscenarioFailure$

Algorithm: $MCEDF$
Input: **job instance J**
Output: **priority vector** PT
$PT \leftarrow JobsOrderedByEDF(\mathbf{J})$;
if $LOscenarioFailure(PT, \mathbf{J})$ **then**
    return (FAILURE-NON-SCHEDULABLE-INSTANCE)
end if
$PT \leftarrow ImproveHIJobs(PT, \mathbf{J})$
if $anyHIscenarioFailure(PT, \mathbf{J})$ **then**
    return (FAILURE-MCEDF-SCHEDULABILITY)
end if

Figure 1: The MCEDF algorithm for computing priorities

Algorithm: $MonotonicImproveHIJobs$
In/out: **priority vector** PT
Input: **job instance J**
$i \leftarrow 2$
while $i \leq K$ **do**
    $Swapped \leftarrow$ **False**
    if $i \geq 2 \wedge \chi_{PT[i]} = \text{HI} \wedge \chi_{PT[i-1]} = \text{LO}$ **then**
        if $CanSwap(i, i-1, \ PT, \mathbf{J})$ **then**
            $PT \leftarrow Swap(i, i-1, \ PT)$;
            $Swapped \leftarrow$ **True**
        end if
    end if
    if $Swapped$ **then**
        $i \leftarrow i - 1$
    else
        $i \leftarrow i + 1$
    end if
end while

Figure 2: Deadline-monotonic improvement for Condition 2 – the "bubble-sort"

evaluates if Condition 2 is met. In this case the algorithm succeeds. The check is done by simulation over basic scenarios, using a similar approach as described in the proof of Theorem 2 in [3]. The latter shows how this can be done in polynomial time and this method, in fact, applies to all monotonic scheduling policies, of which FPM is an example.

The reminder of this subsection is dedicated to the HI job priority improvement procedure. In terms of schedulability, this procedure is constrained by meeting the LO scenario deadlines, postponing the HI scenario checks until the final check of the MCEDF. The simpler variant of the HI job priority improvement is shown in Figure 2. 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 deadline-monotonic. This is done in a manner similar to a bubble-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 $CanSwap(i, i-1, \ldots)$ simulates the fixed priority schedule PT with entries i and $i-1$ swapped and returns whether all deadlines are met. Subroutine $Swap$ performs the actual swapping.

Example 3.1. *Let* **J** *be the instance defined by the following table:*

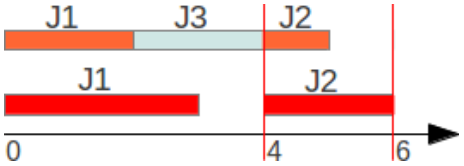


Figure 3: The possible execution of the Job set

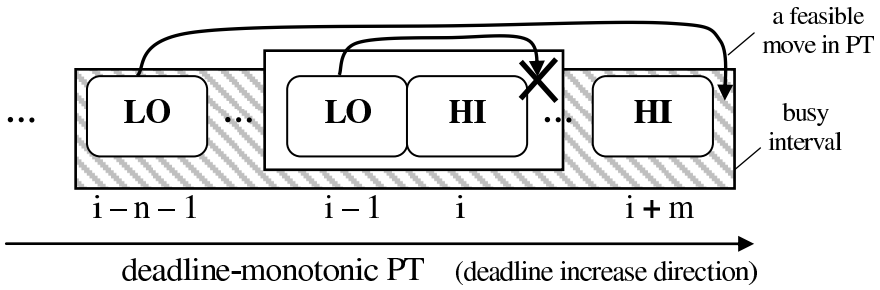


Figure 4: The blocking of the deadline-monotonic improvement

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

The algorithm of Figure 2 will first give EDF priorities to the jobs, thus generating the following priority table:

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

then it will check condition of line 7 on J_1 and J_3. Since the condition holds, it will check if the swap is possible by checking by simulation if J_3 will met its deadline in a 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 perform the swap, thus obtaining:

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

Since there are no other possible swap, the algorithm terminates. Figure 3 shows that using this priority order all deadlines are met in all the possible scenarios of the instance.

As sketched in Figure 4, the fact that a LO job PT_{i-1} cannot be moved behind the HI job PT_i in the priority table still does not always exclude the possibility that a tighter-deadline LO job PT_{i-1-n} can be moved behind a looser-deadline HI job PT_{i+m}, whose completion time would thereby improve, which can be crucial for the HI-scenario schedulability. In the bubble sort-like procedure, the job pair (PT_{i-1}, PT_i) would *block* this possibility, whereas OCBP would not exclude this assignment, as it evaluates all jobs for the lower position in the priority table.

To increase MCEDF effectiveness, we allow more swapping between HI and LO jobs, as long as they are adjacent in busy intervals. A *busy interval* for subinstance $\mathbf{J}'$, $\mathbf{J}' \subseteq \mathbf{J}$ 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). For convenience, we apply the term 'busy interval' also to the subset of jobs running in that interval.

Obviously, if two jobs are in different busy intervals then the relation between their priorities makes no difference for the schedule. Swapping in different busy intervals independently eliminates the blocking problem described in Figure 4. This statement is supported by the following (easy to prove) lemma stated informally below.

1: **Algorithm:** $ImproveHIJobs$
2: **In/out:** **priority vector** PT
3: **Input:** **job instance J**
4: $IvList \leftarrow ListLOBusyIntervals(\mathbf{J})$
5: **for** $\mathbf{J}' \in IvList$ **do**
6: $PT' \leftarrow (PT \mid \mathbf{J}')$
7: $PT' \leftarrow MonotonicImproveHIJobs(PT', \mathbf{J}')$
8: **if** $|\mathbf{J}'| > 3$ **then**
9: $PT' \leftarrow ImproveHIJobs(PT',\ \mathbf{J}' \setminus \{last(PT')\})$
10: **end if**
11: $(PT \mid \mathbf{J}') \leftarrow PT'$
12: **end for**

Figure 5: Generalized improvement for Condition 2 used in MCEDF

Lemma 3.1. *Given an instance $\{PT_1, \ldots, PT_{i+m}\}$ with a priority table PT that is deadline-monotonic per criticality level. Suppose (Fig. 4) that a LO job PT_{i-1} cannot be moved behind the HI job PT_i in the priority table whereas LO job PT_{i-1-n} can be moved behind a HI job PT_{i+m}, and it would improve the completion time of the latter. Then there must be at least two distinct busy intervals of this subinstance: one with jobs PT_i and PT_{i-1} and the other one with PT_{i-1-n} and PT_{i+m}.*

The busy intervals are also shown in Fig. 4 and it is obvious that the basic bubble-sort improvement would succeed in moving PT_{i-n-1} if applied to the second interval. Informally this lemma states that if we isolate the smallest *subinstance* $\{PT_1, \ldots PT_j\}$ that includes all the jobs involved in the blocking then all "blocking" job pairs (PT_{i-1}, PT_i) will be isolated into separate busy intervals of this subinstance, which eliminates these obstacles for improving the priority of the lowest-priority HI job.

The generalized priority improvement subroutine is based on this observation and is described in Figure 5. First we split the instance into busy intervals. Then for every interval, we extract from PT the subvector PT' of jobs belonging to the current busy interval, preserving their relative order, this is denoted $(PT \mid \mathbf{J}')$. Then we invoke the individual bubble-sort improvement for the given interval. After this we remove the lowest priority job from the interval and fix its priority assignment (note here a similarity to OCBP). As argued earlier, if this lowest priority job is a HI job, its priority cannot be improved anymore. Removing the lowest-priority job isolates a smaller subinstance of the remaining jobs of that interval and thus may reveal further busy interval fragmentation of the remaining jobs. Therefore we apply the same procedure recursively for the finer busy intervals. Finally, we reinsert the jobs of $\mathbf{J}'$ in the updated order into the original subset of entries in PT.

The instance of Example 2.1 illustrates the blocking problem, as it may not be solved with a simple monotonic improvement. In fact in this example, there are two busy intervals: $(0, 3]$ with jobs $\{J_1, J_4\}$ and $(3, 5]$ with jobs $\{J_2, J_3\}$. The bubble sorting will not change the priority table (J_1, J_2, J_3, J_4), because job pair J_2, J_3 cannot be swapped. In this case, HI job J_4 will start at time 2 and may miss its deadline if executed in HI scenario, being preempted by J_3. With the new procedure, priorities in the interval $\{J_1, J_4\}$ will be swapped by a bubble sort applied to this interval individually, and we get a correct priority assignment: (J_4, J_2, J_3, J_1). The reader can check that with this priority assignment all deadlines will be met.

However, in Example 2.1 we have shown that this instance is OCBP-schedulable. So, taking the busy intervals into account is a necessary feature for our algorithm to be at least as powerful as OCBP. In the next subsection, we claim that, in fact, it dominates OCBP.

Another important property of MCEDF is described below.

Lemma 3.2. *The priority table computed by MCEDF can be reordered without impact on the behavior but letting the HI jobs occur in EDF order* w.r.t. *each other.*

Reordering the priorities as described in this lemma is in fact part of the proof for a different claim presented in Appendix A. Section 5 explains the practical meaning of such a reordering.

3.3 Dominance over OCBP

In this subsection we provide the theoretical evidence that MCEDF dominates over OCBP. The set of jobs described in Example 3.1 is MCEDF schedulable, while we will show that is non-OCBP schedulable.

Reminding that if all the arrival time are equal to 0, we can check the following condition[7] for job J_j, instead of doing simulation:

$$\sum_{J_i: pr(J_i) > pr(J_j)} C(\chi_j) \leq D_j$$

no job can get the last priority, since we have for J_1: $3+2+2 > 6$, for J_2: $3+2+2 > 6$, for J_3: $2+2+1 > 4$.

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.*

Proof. See Appendix A. □

3.4 Characterization

The characterization of a scheduling algorithm means defining certain metrics that estimate the schedulability of problem instances under different scheduling algorithms. Sometimes they are used to conservatively evaluate the schedulability of an instance. Instead, MCEDF executes an *exact schedulability test* offline, in subroutine $anyHIscenarioFailure$. Thus, in our work, the characterization metrics are only used as convenient indicators of algorithm performance, but not for a schedulability test.

One metric, proved useful for mixed-critical systems is *speedup factor*. 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 know that MCEDF also has the optimal speedup factor.

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 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 these metrics for the OCBP algorithm. The authors determine the load a CPU can experience in a LO and in a HI scenario as shown below:

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

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

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

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

This is also a sufficient condition for clairvoyant scheduling, but not for the online policies, [15], because they do not 'know' in advance the time instance of an occasional mode switch, which may interleave the processing demands of LO and HI scenario such the result might exceed the available processing capacity.

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

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

Due to the dominance over OCBP (Theorem 3.3) we can state that it can be applied to MCEDF as well.

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 clairvoyantly schedulable one. 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 $\Delta C_j = C_j(\text{HI}) - C_j(\text{LO})$ (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 in a HI scenario. Therefore in [11] the metric $Load_{\text{LO}}$ is replaced by a new metric $Load_{\text{MIX}}$, defined as the one equal to $Load_{\text{LO}}$ after substituting D'_j into D_j:

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

The *necessary* condition (1) is thus rewritten to:

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

In fact, in [11] a dynamic-priority scheduling approach is proposed, for which the condition (2) is claimed sufficient for schedulability. However, should this claim ever be true then (2) would be 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 (2) is only a necessary but not a sufficient schedulability check. The table below gives a counter-example for the sufficiency of (2):

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 (2). 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 one with the highest priority we will not have enough time to execute both J_2 and J_3 if they both show an HI behavior. 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.

$Load_{\text{MIX}}$ is a better indicator of schedulability than $Load_{\text{LO}}$, especially for mode-switched policies. To demonstrate this, consider *splitting*, a transformation 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 the reduced uncertainty by construction. These observations are confirmed in our experiments in Section 4.

The following example demonstrates the effect of splitting. It has $Load_{\text{MIX}} = 1.66\ldots$:

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 (2) is broken and due to uncertainty of the execution mode. If J_1 executes first then J_2 starts at time 5. In the LO mode there would be no problem,

but J_2 misses its deadline should it 'decide' to execute in the HI mode, for 12 time units. Otherwise , if J_2 starts first, then even in the HI mode it meets its deadlines (and the LO job J_1 can be dropped), but there is a problem in the LO mode, as J_2 would delay J_1 by two time units, leading to a missed deadline. The clairvoyant scheduler would know the mode 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
2	0	12	HI	1	6
2	0	12	HI	1	6

The scheduler can execute J_3 until completion first getting from it the online knowledge of the mode that was missing in the previous case. If job J_3 runs in LO mode, J_2 can follow, starting at time 1, and then J_2 can run from time 6 even until time 12 in the HI mode. If job J_3 will run the HI mode, J_1 will be skipped, and J_3 together with J_2 meet the deadline. Compared to the instance before the split, $Load_{\text{MIX}}$ reduces from $1.66\ldots$ 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.* to the original one.

Note that the splitting 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, their code blocks 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.

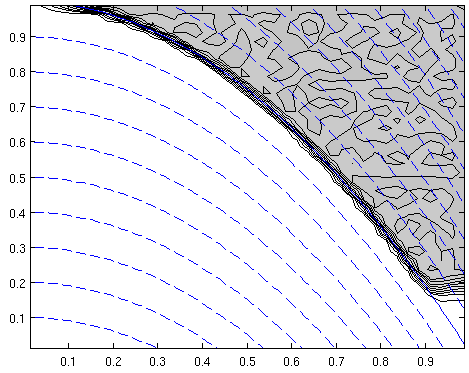
4 Implementation and Experiments

We evaluated the schedulability performance of MCEDF relative to OCBP in experiments with randomly generated job instances of 20 jobs [1] each having arrival/deadline/execution times being integers, simulating CPU clock cycle count of some imaginary machine. Every job instance was generated for a target LO and HI load pair.

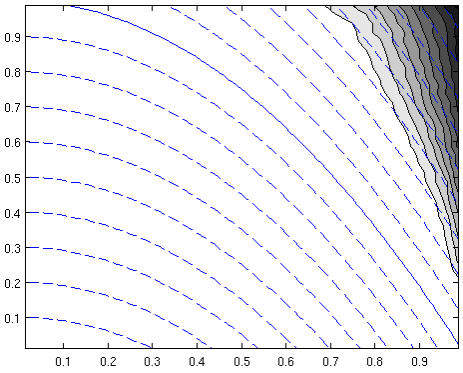
The method to generate a job instance was as follows. First we randomly generated a tentative instance, not paying attention to the target loads. This was done by repeatedly generating a new sporadic task, i.e. sequence of jobs arriving one after another at random arrival intervals (thus modeling a realistic situation where our algorithm would be applied at run time for jobs generated by sporadic tasks). 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.,* χ = 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(LO) of each job was uniformly distributed between 0 and the relative deadline, each HI job had a C_j(HI) obtained by scaling the value C_j(LO) by a random factor [1..1000]. 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. 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 until multiple attempts produced no satisfactory load scaling result within a timeout. Due to this the job generation process itself took a considerable time in the experiments. Nevertheless one or a few seconds were typically only required for very low $Load_{\text{HI}}$ (≤ 0.2) combined with a high $Load_{\text{LO}}$ (> 0.9). We limited the timeout for repeating the attempts by 10 s per one experiment, resulting in up to 6000 attempts on a 1.2 GHz machine.

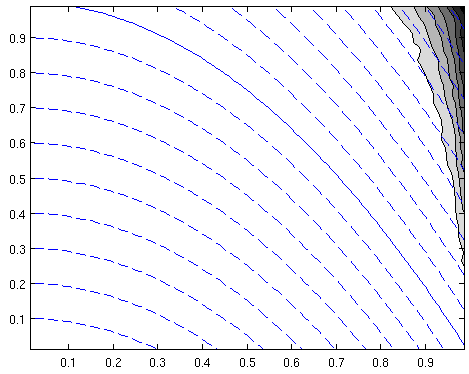
[1] Note that the instance size was restricted by the job generation algorithm and not by the scheduling algorithm.



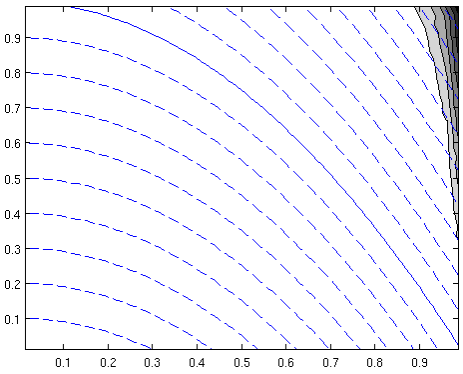
(a) Density of Generated Jobs



(b) non OCBP-schedulable 14%



(c) non MCEDF-schedulable 5%



(d) non MCEDF-schedulable after split 3%

Figure 6: Experimental results

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. 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. Per each target, ten experiments were run, generating the points lying near the target with tolerance 1%. Fig 6(a) gives the contour graph of the density of the generated points in grayscale. The horizontal axis is $Load_{\text{LO}}$, the vertical is $Load_{\text{HI}}$. The grid follows the parabolic lines of equal $Load^2_{\text{LO}}(\mathbf{J}) + Load_{\text{HI}}(\mathbf{J})$. The total number of experiments was 537460, whereas we failed to generate 2.7% (14427) of points due to the load scaling problem, all these missing points are at the right bottom corner in Fig 6(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 instance by almost one third. The density distributions in Figure 6 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 uncertainty. After splitting the instances remained to be non-OCBP schedulable (as OCBP cannot make profit of less uncertainty) but the number of non-MCEDF schedulable instances has reduced, and it came to 3% (16991). So if we can accept this load-preserving transformation, we go from original 14% to 3% non-schedulability due to MCEDF. Note that 2% (11812) became schedulable after split, while 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 optimistic assumption for the WCET tools, but still doing this is worth a try.

5 Discussion and Related Work

MCEDF uses the flexibility of FPM to be dominant over fixed priority algorithms. The main advantage of MCEDF over other non fixed priority solutions is that this algorithm can be implemented on existing systems that support fixed priority algorithms with minimal modification. In fact, due to Lemma 3.2, we can always use one priority table and signal to the LO jobs to speedily return when the HI mode is active.

To the best of our knowledge no FPM algorithm that are theoretically dominant over OCBP has been proposed in the past. The priority assignment of [13] applies OCBP to compute PT_{LO}, thus having equivalent schedulability. [8] proposes a low-overhead online computation of priorities. [9] presents a high performance priority computation that dominates OCBP empirically. Note that [8, 9] are not directly applicable to the problem studied in this paper as they are restricted to the periodic job model.

FPM provides better results than fixed priority, but in general dynamic priority may be necessary for optimality. Consider the following example instance $\mathbf{J}_d$:

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. (Nevertheless the time instances for job preemption can be restricted to job arrivals and the mode switch [3].) The only correct scheduling policy for $\mathbf{J}_d$ is to execute J_1 for 1 time unit, then J_2. If J_2 terminates 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 schedule is not fixed priority (J_1 changes its priority *w.r.t.* J_3).

In [11], an idea for a dynamic priority scheduling is proposed. In this work however the algorithm performance is evaluated experimentally using (2) as a sufficient condition for schedulability with their technique. However we showed in Section 3.4 that this condition is not even sufficient for MC-schedulability.

6 Conclusions and Future Work

This paper presents a new real-time scheduling algorithm for mixed-critical systems, fitting into the context of 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 without resource sharing is NP-complete and cannot be solved in general by classical fixed-priority per job scheduling policies. In this paper, we present a scheduling algorithm that can be implemented as an extension of fixed priority scheduler, enjoying the advantages of relative ease of practical implementation and theoretical 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-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 mixed-critical sporadic 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 and dataflow graphs, where the jobs have mutual data dependencies. 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 *Proc. Real-Time Systems Symposium*, RTSS'07, pp. 239–243, IEEE, 2007. 1, 2.1

[2] D. d. Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *Proc. Real-Time Systems Symposium*, RTSS'09, pp. 291–300, IEEE, 2009. 1

[3] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Trans. Comput.*, vol. 61, pp. 1140 –1152, aug. 2012. 1, 2.1, 2.2, 2.3, 3.2, 3.4, 3.4, 5

[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 *Proc. Int. Conf. Computer and Information Technology*, CIT '10, pp. 1864–1871, IEEE, 2010. 1

[5] L. A. Johnson, "Do-178b: Software considerations in airborne systems and equipment certification.," in *Radio Technical Commission for Aeronautics.* 1

[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. 1, 2.3

[7] S. K. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Proc. Real-Time and Embedded Technology and Applications Symposium*, RTAS'10, pp. 13–22, IEEE, 2010. 1, 2.3, 3.3, 3.4

[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 *Proc. Euromicro Conf. on Real-Time Systems*, ECRTS'12, pp. 145–154, IEEE, 2012. 1, 5

[9] P. Ekberg and W. Yi, "Bounding and shaping the demand of mixed-criticality sporadic tasks," in *Proc. Euromicro Conf. on Real-Time Systems*, ECRTS'12, pp. 145–154, IEEE, 2012. 1, 5

[10] S. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *Proc. Real-Time Systems Symposium*, RTSS '11, pp. 3–12, IEEE, 2011. 1, 2.1

[11] T. Park and S. Kim, "Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems," in *Proc. Intern. Conf. on Embedded software*, EMSOFT '11, pp. 253–262, ACM, 2011. 1, 3.4, 3.4, 5

[12] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003. 1

[13] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems," in *Proc. Real-Time Systems Symposium*, RTSS '11, pp. 13–23, IEEE, 2011. 1, 5

[14] N. Audsley, *Flexible Scheduling in Hard-Real-Time Systems*. PhD thesis, Dept. of Computer Science, Univ. of York, 1993. 2.3

[15] H. Li and S. Baruah, "Load-based schedulability analysis of certifiable mixed-criticality systems," in *Proc. Intern. Conf. on Embedded Software*, EMSOFT '10, pp. 99–108, ACM, 2010. 3.4, 3.4

A Proof of Theorem 3.3

Proof. Suppose that we have an OCBP-schedulable instance **J**. In the proof we show that MCEDF can be modified without impact on the schedulability to produce the same priority table PT as OCBP for instance **J**. This table will be used by MCEDF only before the mode switch, thus having exactly the same behavior as OCBP in this case. After the mode switch OCBP manages to meet 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 applies the optimal algorithm – EDF – for the remaining HI jobs.

The modified MCEDF has higher complexity while preserving the original MCEDF schedule. The schedule is preserved because, when compared to the default MCEDF, the modified variant of MCEDF only changes the relative priorities of two jobs in PT when they lie in different busy intervals of the basic LO scenario. This implies that the LO scenario schedule remains the same, and for the FPM policy it means that the complete schedule remains the same.

Consider the priority table PT computed by MCEDF. Observe that by construction all jobs meet their deadlines in the LO scenario. Let us split the basic LO scenario schedule of MCEDF into busy intervals. There are 2 cases:

Case 1: There is a busy interval in which the lowest priority job is a LO job In this case, for MCEDF, we modify PT by moving arbitrary such LO job J_j to the end of PT. Obviously, it remains to be the lowest-priority job in its busy interval, so the MCEDF behavior is not changed. As for OCBP, observe that to evaluate whether a LO job can be assigned the lowest priority, OCBP also uses the basic LO scenario. Therefore, by Lemma 2.1, OCBP can select the same job J_j for the lowest priority as the MCEDF.

Case 2: The lowest-priority job in every busy interval is a HI job MCEDF selects the latest-deadline job among all HI jobs in the instance – J_j – and moves it to the end of PT. Observe that the MCEDF priority improvement in every busy interval is deadline-monotonic per criticality level and hence it assigns the lowest priority inside the busy interval to either a latest-deadline HI job or to the latest-deadline LO job among all jobs in the busy interval. Therefore, J_j will be the lowest-priority in its busy interval. Therefore, by the same argument as in Case 1, the MCEDF schedule remains the same after this modification of the priority table PT.

Now what remains to be proved for Case 2 is that OCBP can select the HI job J_j as the lowest-priority job. For this we just need to prove that OCBP cannot select any LO job in this case. If so, then in OCBP instance there must be a selectable HI job which completes the last among all HI jobs in the maximal HI scenario (*i.e.,* with execution times $C_j(\text{HI})$) and still meets its deadline. The latest deadline HI job J_j if selected for the least priority cannot complete later and thus will meet its deadline, thus being selectable.

We prove that no LO job can be selected by OCBP in Case 2 by contradiction. Consider any busy interval. Let a LO job be selectable in this busy interval for the lowest priority, then the latest-deadline LO job in the same busy interval $J_{j'}$ would be selectable as well. Just after the monotonic improvement of Figure 2 was calculated for this busy interval, let p' be the final position of $J_{j'}$ in the priority table PT' for that interval. Let J_i be the latest priority job in the given busy interval, which is a HI job by construction of Case 2. Then we have that the final priority table looks like:

$$(\ldots J_{j'},\ \ PT'(p'+1),\ \ PT'(p'+2),\ \ \ldots,\ \ J_i)$$

where $J_{j'}$ are followed by HI jobs until the end of the priority table. At some point the monotonic improvement of Figure 2 has evaluated the possibility to swap the jobs $PT(p'+1)$ and $J_{j'}$ and it has failed. This could only happen if after this swap $J_{j'}$ misses its deadline in the LO scenario. Because the completion time is monotonic in the priority, from this we can conclude that $J_{j'}$ misses its deadline in the LO scenario when moved in the end of the priority table. However, this is in contradiction with our current hypothesis that in OCBP could select $J_{j'}$ as the least-priority job.

Due to this contradiction, we conclude that OCBP can select J_j in Case 2, thus making the same choice as the modified MCEDF. This completes the proof.

□