# Time-triggered Scheduling for Multiprocessor Mixed-criticality Systems

Lalatendu Behera and Purandar Bhaduri

Indian Institute of Technology Guwahati 781039, India
{lalatendu,pbhaduri}@iitg.ernet.in

**Abstract.** Real-time safety-critical systems are getting more complex by integrating multiple applications with different criticality levels on a single platform. The increasing complexity in the design of mixed-criticality real-time systems has motivated researchers to move from uniprocessor to multiprocessor platforms. In this paper, we focus on the time-triggered scheduling of both independent and dependent mixed-criticality jobs on an identical multiprocessor platform. We show that our algorithm is more efficient than the Mixed criticality Priority Improvement (MCPI) algorithm, the only existing such algorithm for a multiprocessor platform.

## 1 Introduction

A *mixed-criticality real-time system* (MCRTS) [1, 2] has two or more distinct levels of criticality, such as, safety-critical, mission-critical, non-critical, etc. For example in the domain of unmanned aerial vehicles (UAV's) [3, 2] the functionalities are classified into two levels of criticality, viz., *mission-critical* (e.g., capturing and transmitting images) and *flight-critical* (e.g., safe operation of the UAV). The flight-critical functionality, due to its safety critical nature, is subject to certification by a certification authority (CA). The CAs are very conservative, using tools and techniques that estimate more pessimistic worst-case execution times (WCET) than that of the system designers. On the other hand, the CAs are not concerned with the mission-critical functionalities. The system designers are interested in both flight-critical and mission-critical functionalities but their tools are less conservative in estimating the WCETs.

The challenge in scheduling such mixed critical systems is to find a single scheduling policy so that the requirements of both the system designers and the CAs are met. This means that in a scenario where all the jobs complete their executions by their LO-criticality WCETs, they must all be scheduled correctly. On the other hand, in a scneario where the execution time of any one HI-criticality job exceeds its LO-criticality WCET, then all the HI-criticality jobs need to meet their deadlines assuming their HI-criticality WCET to satisfy the CAs.

In this paper, we describe an approach to find a preemptive, global, time-triggered schedule of mixed-criticality, non-recurrent task systems on identical

multiprocessor platforms that can satisfy the assumption of both the CAs and SDs. We show that the worst-case time complexity of our proposed algorithm is better than the existing algorithm in [4], the only existing time-triggered algorithm for such systems.

## 2 System Model

A mixed-criticality system consists of $n$ jobs $\{j_1, j_2, \ldots, j_n\}$, each with a criticality level. Here we focus on dual-criticality jobs, i.e., LO-criticality and HI-criticality. A job $j_i$ is characterized by a 5-tuple of parameters: $j_i = (a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$, where

- $a_i \in \mathbb{N}$ denotes the *arrival time*, $a_i \geq 0$.
- $d_i \in \mathbb{N}^+$ denotes the *absolute deadline*, $d_i \geq a_i$.
- $\chi_i \in \{LO, HI\}$ denotes the *criticality* level.
- $C_i(\text{LO}) \in \mathbb{N}^+$ denotes the LO-criticality *worst-case execution time*.
- $C_i(\text{HI}) \in \mathbb{N}^+$ denotes the HI-criticality *worst-case execution time*.

We assume that the system is *preemptive* and $C_i(\text{LO}) \leq C_i(\text{HI})$ for $1 \leq i \leq n$. Note that in this paper, we consider arbitrary arrival times of jobs. An instance of mixed-criticality job set can be defined as a finite collection of mixed-criticality jobs, i.e., $I = \{j_1, j_2, \ldots, j_n\}$. Generally, a job in the instance $I$ is available for execution at time $a_i$ and should finish its execution before $d_i$. The job $j_i$ must execute for $c_i$ amount of time which is the actual execution time between $a_i$ and $d_i$, but this can be known only at the time of execution. The collection of actual execution times ($c_i$) of the jobs in an instance $I$ at run time is called a ***scenario***. Scenarios in our model can be of two types, i.e., *LO-criticality scenarios* and *HI-criticality scenarios*. When each job $j_i$ in instance $I$ executes $c_i$ units of time and signals completion before its $C_i(\text{LO})$ execution time, it is called a LO-criticality scenario. If any job $j_i$ in instance $I$ executes $c_i$ units of time and doesn't signal its completion after it completes the $C_i(\text{LO})$ execution time, then this is called a HI-criticality scenario. Now we define a schedulability condition for a mixed-criticality instance $I$.

**Definition 1:** A scheduling strategy is *feasible or correct* if and only if the following conditions are true:

1. If all the jobs finish their $C_i(\text{LO})$ units of execution time on or before their deadlines.
2. If any job doesn't declare its completion after executing its $C_i(\text{LO})$ units of execution time, then all the HI-criticality jobs must finish their $C_i(\text{HI})$ units of execution time on or before their deadlines.

Here we focus on the **time-triggered schedule** [5] of MC instances on a multiprocessor system with identical processors. We will construct two tables $S_{\text{HI}}$ and $S_{\text{LO}}$ for each processor for a given instance $I$ for use at run time. The length of the tables is the length of the interval $[\min_{j_i \in I}\{a_i\}, \max_{j_i \in I}\{d_i\}]$. The rules to use the tables $S_{\text{HI}}$ and $S_{\text{LO}}$ at run time, (i.e., the *scheduler*) are as follows:

– The criticality level indicator $\Gamma$ is initialized to LO.
– While ($\Gamma = LO$), at each time instant t the job available at time t in the table $S_{\text{LO}}$ for processor $P_i$ will execute on $P_i$.
– If a job executes for more than its LO-criticality WCET without signaling completion in any processor $P_i$, then $\Gamma$ is changed to HI.
– While ($\Gamma = HI$), at each time instant t the job available at time t in the table $S_{\text{HI}}$ for processor $P_i$ will execute on $P_i$.

## 3 Related Work

Most research on mixed-criticality systems focuses on the uniprocessor case (see for example, [2, 6]). The increasing functionalities in mixed-criticality systems motivate researchers to turn to multiprocessor systems (see [7–10, 4]). Among the above cited work only [5, 6] focus on a time-triggered scheduling algorithm for uniprocessor systems and [4] introduces a time-triggered scheduling algorithm for multiprocessor systems. To the best of our knowledge, there has not been *any other work* studying time-triggered mixed-criticality scheduling for multiprocessor systems.

Socci et al. [4] proposed the Mixed criticality Priority Improvement (MCPI) algorithm to schedule jobs with precedence constraints. In this algorithm, they construct a priority order of jobs from the support algorithm (i.e., a multiprocessor algorithm for non-critical jobs) which is used to find a table for the LO-scenario and the support algorithm is used to schedule the HI-criticality jobs in HI-scenarios. They showed the worst-case time complexity of the algorithm to be $O(n^2 + mn^3 \log n)$ for independent jobs, where $n$ is the number of jobs in the instance $I$ and $m$ is the number of processors.

## 4 The proposed algorithm

In this section, we propose an algorithm for mixed-criticality jobs on multiprocessor systems which not only schedules the same set of instances as the existing algorithm [4] but also has a better worst-case time complexity.

The time-triggered scheduling approach to mixed-criticality jobs [4] constructs two scheduling tables $S_{\text{LO}}$ and $S_{\text{HI}}$ to schedule a dual-criticality instance. Since we consider mixed-criticality jobs for a multiprocessor system, we need two separate scheduling tables for each processor. The schedule constructed by our algorithm is a global one, i.e., a job can be preempted in one processor and resume its execution in another processor. Here we assume that the system is a closely coupled synchronous homogeneous multiprocessor system with shared last level cache and the job context switch time is negligible. We also assume that the cache miss penalty is negligible.

Algorithm 1 determines a priority order, which is used to construct the scheduling tables for all the processors, in steps 1 to 11. First, our algorithm finds the LO-scenario deadline (${d_i}^{\Delta}$) of each job. For the LO-criticality jobs ${d_i}^{\Delta} = d_i$, but for HI-criticality ones ${d_i}^{\Delta} \leq d_i$. Then the algorithm starts to

**Algorithm 1** LoCBP (LO-criticality based Priority)

**Notation**:
$I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i,\ d_i,\ \chi_i,\ C_i(\text{LO}),\ C_i(\text{HI}) >$.
**Input** : $I$
**Output** : Priority Order ($\Psi$) of Instance $I$
Assume the earliest arrival time is 0.

1: Compute the LO-scenario deadline (${d_i}^{\Delta}$) of each job $j_i$ as ${d_i}^{\Delta} = d_i - (C_i(\text{HI}) - C_i(\text{LO}))$;
2: **while** $I$ is not empty **do**
3: Assign a LO-criticality latest deadline[1] job $j_i$ as the lowest priority job if $j_i$ can finish its execution in the interval $[a_i, {d_i}^{\Delta}]$ after all other jobs finish their execution in LO-scenario under the global EDF scheme;
4: If any LO-criticality job cannot be given a lowest priority then a HI-criticality latest deadline[1] job $j_i$ is assigned as the lowest priority job if $j_i$ can finish its execution in the interval $[a_i, {d_i}^{\Delta}]$ after all other jobs finish their execution in LO-scenario under the global EDF scheme;
5: **if** No job is assigned a lowest priority **then**
6: Declare FAIL and EXIT;
7: **else**
8: Add job $j_i$ to the priority order $\Psi$;
9: Remove job $j_i$ from the instance and continue;
10: **end if**
11: **end while**
12: Construct table $S_{\text{LO}}$ for each processor using the priority order;
13: **if** $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ **then**
14: return FAIL and EXIT;
15: **end if**
16: The same order as $S_{\text{LO}}$ is followed to allocate the jobs in $S_{\text{HI}}$;
17: After a HI-criticality job $j_i$ is allocated its $C_i(\text{LO})$ execution time in $S_{\text{HI}}$, $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time of job $j_i$ is allocated after the rightmost segment of job $j_i$ in $S_{\text{LO}}$ without disturbing the priority order $\Psi$ and overwriting LO-criticality jobs in the process, if any;

assign the lowest priority jobs from the instance $I$. It always selects the latest deadline job to be assigned as the lowest priority job, but LO-criticality jobs are considered before the HI-criticality jobs. A job $j_i$ can be assigned the lowest priority if and only if all other jobs $j_k$ finish their executions when run according to the global EDF algorithm and there remains sufficient time for $j_i$ to complete its $C_i(\text{LO})$ units of execution time before ${d_i}^{\Delta}$. After job $j_i$ is assigned the lowest priority, it is removed from the instance, and the remaining jobs are considered for priority assignment. If at any step a job cannot be assigned a priority, the algorithm declares failure. In step 10, the algorithm constructs table $S_{\text{LO}}$. In steps 11 to 13, it checks for any possible HI-criticality scenario failure. The subroutine $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ checks if at least one job runs at its $C_i(\text{HI})$ execution time, then all HI-criticality jobs must complete their HI-criticality execution before their deadline. If it doesn't find a HI-criticality scenario failure from the subroutine $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$, then the priority order constructed by Algorithm 1 can successfully schedule the instance $I$. Algorithm 1 constructs table $S_{\text{LO}}$ for each processor. Then Table $S_{\text{HI}}$ is constructed for each processor by allocating the remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time of each HI-criticality job after its $C_i(\text{LO})$ units of execution time in $S_{\text{HI}}$ using

[1] The original deadline and not the LO-scenario one.

the same priority order and also a HI-criticality job is given higher priority over LO-criticality jobs. This means a HI-criticality job can overwrite a LO-criticality job in the process of allocating its $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time.

We illustrate the operation of this algorithm by an example.

**Example 1:** Consider the mixed-criticality instance given in Table 1 to be scheduled on a multiprocessor system having two identical processors $P_0$ and $P_1$.

**Table 1.** The instance for Example 1

| Job | Arrival time | Deadline | Criticality | $C_i(\text{LO})$ | $C_i(\text{HI})$ |
|---|---|---|---|---|---|
| $j_1$ | 1 | 5 | LO | 3 | 3 |
| $j_2$ | 0 | 8 | LO | 4 | 4 |
| $j_3$ | 0 | 7 | HI | 3 | 5 |
| $j_4$ | 0 | 4 | HI | 2 | 2 |

Now we construct a priority order using our algorithm. The LO-scenario deadlines $d_i^{\Delta}$ of jobs $j_1, j_2, j_3, j_4$ are $5, 8, 5, 4$ respectively. Now we start assigning priorities to each job.

- The job $j_2$ is the latest LO-criticality deadline job. If $j_2$ is assigned the lowest priority, then $j_3$ and $j_4$ can run simultaneously in $P_0$ and $P_1$ over $[0, 3]$ and $[0, 2]$ respectively. Then $j_1$ will run over $[2, 5]$ in $P_1$. So $j_2$ can execute its 4 units of execution time in $P_0$ over $[3, 7]$ to finish by its deadline. Now we can assign job $j_2$ the lowest priority. We remove job $j_2$ and consider $\{j_1, j_3, j_4\}$ to find the next lowest priority job.
- If $j_1$ is assigned the lowest priority, then $j_3$ and $j_4$ can run simultaneously on $P_0$ and $P_1$ over $[0, 3]$ and $[0, 2]$ respectively. Then $j_1$ will run over $[2, 5]$ in $P_1$. So $j_1$ can execute its 3 units of execution time in $P_1$ over $[2, 5]$ to finish by its deadline. Now we can assign job $j_1$ the lowest priority. Next, we remove the job $j_1$ and consider $\{j_3, j_4\}$ to assign the next lowest priority.
- Since there are two jobs and two processors, any job can be given lower priority among the two. But our algorithm assigns the latest deadline job as the lowest priority job. So job $j_3$ is given the lowest priority.

Finally, the priority order of the jobs in instance $I$ is $j_4 \triangleright j_3 \triangleright j_1 \triangleright j_2$. Now Algorithm 1 constructs the table $S_{\text{LO}}$ for each processor using the above priority order. The table $S_{\text{LO}}$ for each processor is given in Fig. 1.

Then the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine checks for all possible HI-criticality scenarios. We can check that all HI-criticality scenarios are schedulable using the priority order $\{j_4, j_3, j_1, j_2\}$ of $I$. Finally, table $S_{\text{HI}}$ is constructed for each processor by allocating the remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time of each HI-criticality job after its $C_i(\text{LO})$ units of execution time in $S_{\text{HI}}$ using the same priority order, where a HI-criticality job is given higher priority over LO-criticality jobs. The table $S_{\text{HI}}$ for each processor is given in Fig. 2. □

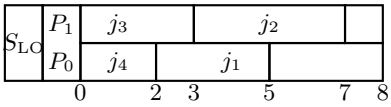


**Fig. 1.** Table $S_{\mathrm{LO}}$ for processor $P_0$ and $P_1$

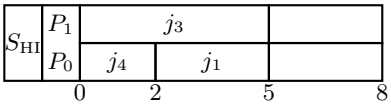


**Fig. 2.** Table $S_{\mathrm{HI}}$ for processor $P_0$ and $P_1$

### 4.1 Correctness Proof

For correctness, we have to show that if our algorithm finds a priority order for instance $I$ and the $anyHIscenarioFailure(S_{\mathrm{LO}}, I)$ subroutine doesn't fail, then the scheduling tables $S_{\mathrm{LO}}$ and $S_{\mathrm{HI}}$ will give a correct scheduling strategy. We start with the proof of some properties of the schedule.

**Lemma 1:** If Algorithm 1 doesn't declare failure and finds a priority order, then each job $j_i$ receives $C_i(\mathrm{LO})$ units of execution time in $S_{\mathrm{LO}}$ and each HI-criticality job $j_k$ receives $C_k(\mathrm{HI})$ units of execution time in $S_{\mathrm{HI}}$.

*Proof.* First, we show that any job $j_i$ receives $C_i(\mathrm{LO})$ units of execution time in $S_{\mathrm{LO}}$. This follows directly from the algorithm as each job $j_i$ must finish its $C_i(\mathrm{LO})$ units of execution time before ${d_i}^{\Delta} \leq d_i$ to be assigned the lowest priority job.

Next we show that any HI-criticality job $j_k$ receives $C_k(\mathrm{HI})$ units of execution time in $S_{\mathrm{HI}}$. We construct the table $S_{\mathrm{HI}}$ according to the same priority order. Since $anyHIscenarioFailure(S_{\mathrm{LO}}, I, \Psi)$ subroutine doesn't find any HI-criticality scenario failure, so all the HI-criticality jobs have received their $C_i(\mathrm{HI})$ units of execution time. □

**Lemma 2:** At any time t, if a job $j_i$ is present in $S_{\mathrm{HI}}$ but not in $S_{\mathrm{LO}}$, then the job $j_i$ has finished its execution in $S_{\mathrm{LO}}$.

*Proof.* We use the same order of jobs in $S_{\mathrm{LO}}$ to construct $S_{\mathrm{HI}}$. Whenever a job $j_i$ has executed for time $c_i \leq C_i(\mathrm{LO})$ at time t, then it is present in both the tables $S_{\mathrm{LO}}$ and $S_{\mathrm{HI}}$. We know the HI-criticality jobs are allocated their $C_i(\mathrm{HI}) - C_i(\mathrm{LO})$ units of execution time after the allocation of $C_i(\mathrm{LO})$ units of execution time in both $S_{\mathrm{HI}}$ and $S_{\mathrm{LO}}$. In $S_{\mathrm{HI}}$, the HI-criticality jobs are higher priority job than LO-criticality jobs. When a job $j_i$ is present in $S_{\mathrm{HI}}$ and not in $S_{\mathrm{LO}}$ at time t, it means this has already completed its execution in $S_{\mathrm{LO}}$. □

**Lemma 3:** At any time t, when a mode change occurs, each HI-criticality job still has $C_i(\mathrm{HI}) - c_i$ units of execution time in $S_{\mathrm{HI}}$ after time $t$ to complete its execution, where $c_i$ is the execution time already completed by job $j_i$ before time t in $S_{\mathrm{LO}}$.

*Proof.* Let a mode change occur at time t. This means that the following statements hold: (i) all the HI-criticality jobs other than the current job, or none

of them has completed their $C_i(\text{LO})$ units of execution time at time t, (ii) the current HI-criticality job is the first one to complete its $C_i(\text{LO})$ units of execution time without signaling its completion. We know that all the HI-criticality jobs are allocated their $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time in $S_{\text{HI}}$ after the completion of their $C_i(\text{LO})$ units of execution time in both $S_{\text{LO}}$ and $S_{\text{HI}}$. If a job $j_i$ has already executed its $C_i(\text{LO})$ units of execution time in $S_{\text{LO}}$, then it requires $C_i(\text{HI}) - C_i(\text{LO})$ units of time to be completed in $S_{\text{HI}}$. When job $j_i$ initiates the mode change, this is the first job which doesn't signal its completion after completing its $C_i(\text{LO})$ units of execution time. Before time t, the scheduler uses the table $S_{\text{LO}}$ to schedule the jobs, while subsequently the scheduler uses table $S_{\text{HI}}$ due to the mode change. If a job $j_i$ has already executed its $c_i$ units of execution time in $S_{\text{LO}}$, then it requires $C_i(\text{HI}) - c_i$ units of time to be completed its execution in $S_{\text{HI}}$. We know that the tables $S_{\text{HI}}$ and $S_{\text{LO}}$ have the same order and according to Lemma 1 and 2, each job will get sufficient time to complete its $C_i(\text{HI})$ units of execution time. Hence, each HI-criticality job will get $C_i(\text{HI}) - c_i$ units of time in $S_{\text{HI}}$ to complete its execution after the mode change at time t. □

**Theorem 1:** If the scheduler dispatches the jobs according to $S_{\text{LO}}$ and $S_{\text{HI}}$, then it will be a correct scheduling strategy.

*Proof.* For the LO-criticality scenarios, all the jobs can be correctly scheduled by the table $S_{\text{LO}}$ as proved in Lemma 1. Now, we need to prove that in a HI-criticality scenario, all the HI-criticality jobs can be correctly scheduled by the table $S_{\text{HI}}$. In Lemma 1, we have already proved that all the HI-criticality jobs get sufficient units of time to complete their execution in $S_{\text{HI}}$. In Lemma 3, we have proved that when the mode change occurs at time t, all the HI-criticality jobs can be scheduled without missing their deadline. So from Lemma 1 and Lemma 3, it is clear that if the scheduler uses tables $S_{\text{LO}}$ and $S_{\text{HI}}$ to dispatch the jobs then it will be a correct scheduling strategy. □

### 4.2 Comparison with MCPI algorithm

**Theorem 2:** An instance $I$ is schedulable by the MCPI algorithm [4] if and only if it is schedulable by our algorithm.

*Proof.* ($\Rightarrow$) The MCPI algorithm generates a priority order for an instance $I$ which is used to find table $S_{\text{LO}}$. When a mode change occurs, it uses a support algorithm to schedule the HI-criticality jobs of instance $I$. We need to show that if MCPI generates a priority order for an instance $I$, then our algorithm will always find a priority order for instance $I$ and the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine will not fail.

Suppose the MCPI algorithm finds a priority order for an instance $I$. Now the least priority job of the priority order (according to the MCPI algorithm) can be either a LO-criticality or HI-criticality job. First, we consider the case where a job is of LO-criticality. Let $j_i$ be the lowest priority job and its criticality

be low. So at the time of construction of the table $S_{\mathrm{LO}}$, every higher priority job $j_k$ finishes its $C_k(\mathrm{LO})$ units of execution time and there remains sufficient time for the lowest priority job $j_i$ to finish its $C_i(\mathrm{LO})$ units of execution time in the interval $[a_i, d_i]$. So this condition is the same as our proposed algorithm.

Let job $j_i$ be the lowest priority job and its criticality be high. Since MCPI successfully finds the priority order, it must have checked all the scenarios and didn't find any failure. Now after every higher priority job $j_k$ finishes its $C_k(\mathrm{LO})$ units of execution time, there remains sufficient time for the lowest priority job $j_i$ to finish its $C_i(\mathrm{LO})$ units of execution time in the interval $[a_i, d_i^{\Delta}]$. Unlike the LO-criticality job, the HI-criticality jobs need to finish their LO-criticality execution on or before $d_i^{\Delta}$. So this condition is the same as our proposed algorithm.

Then $j_i$ is removed from the instance and the next priority can be assigned from the remaining jobs. We can argue in the same way for the remaining jobs. From the above argument, it is proved that our proposed algorithm finds the same priority order for instance $I$ as the MCPI algorithm. Since the priority order is the same and the MCPI algorithm doesn't find any HI-scenario or LO-scenario failure, the $anyHIscenarioFailure(S_{\mathrm{LO}}, I, \Psi)$ subroutine in our algorithm will not fail as well. Thus, for a MCPI schedulable instance, our algorithm can also construct priority tables $S_{\mathrm{LO}}$ and $S_{\mathrm{HI}}$.

($\Leftarrow$) Our algorithm generates a priority order for an instance $I$ which is used to find the table $S_{\mathrm{LO}}$. When a mode change occurs, our algorithm uses the table $S_{\mathrm{HI}}$ to schedule the HI-criticality jobs which is constructed from the job ordering in $S_{\mathrm{LO}}$. We need to show that if our algorithm generates a priority order for an instance $I$, then the MCPI algorithm will always find a priority order and the $anyHIScenarioFailure(PT, T)$ subroutine will not fail.

Suppose our algorithm finds a priority order for an instance $I$. The least priority job assigned by our algorithm can be either a HI-criticality or a LO-criticality job. First, we consider the case where the lowest priority job is LO-criticality. Let $j_i$ be the lowest priority job and its criticality be LO which means the job $j_i$ finishes its execution between its arrival time and deadline after all other jobs finish their execution. So according to the priority table ($SPT$) of MCPI, job $j_i$ can be given the lowest priority among the LO-criticality jobs. Since the job can meet its deadline after all other jobs finish their execution, the $PullUp()$ subroutine [4] will pull up the HI-criticality jobs upward in the priority tree. So according to the MCPI algorithm the job $j_i$ is the lowest priority job among the HI-criticality jobs as well. This shows that the job $j_i$ is the lowest priority job according to the MCPI algorithm.

Now assume $j_i$ is the lowest priority job and its criticality is HI which means the job $j_i$ can finish its execution between its arrival time and deadline after all other jobs finish their execution. Since our algorithm prefers LO-criticality jobs to assign the lowest priority over HI-criticality jobs, there are no LO-criticality jobs available which can be assigned the lower priority. As in the previous case, job $j_i$ is the lowest priority job in the$SPT$ priority table of the MCPI algorithm. Since no LO-criticality job can finish its execution after the execution of job $j_i$, the $PullUp()$ subroutine will not be able to pull up the HI-criticality job upward

in the priority tree. So job $j_i$ is the lowest priority job according to the MCPI algorithm.

So both the algorithms generate the same priority order for instance $I$. Since our algorithm doesn't find any HI-scenario failure in the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine, the MCPI algorithm also doesn't find any HI-scenario failure in its $anyHIscenarioFailure()$ subroutine. □

**Theorem 3:** The computational complexity of LoBCP is $O(mn^3)$, where $n$ is the number of jobs in an instance $I$ and $m$ is the number of processors.

*Proof.* Line 1 takes $O(n)$ time. In lines 3 and 4, finding the latest deadline job takes $O(n \log n)$ time, simulation of global EDF on $m$ processors takes $O(mn^2)$ times [11]. So the total time taken by lines 3 and 4 is $O(n \log n + mn^2)$. Lines 5 to 10 take $O(1)$ time. Since the while loop in line 2 runs $n$ times, line 3 to 10 require a total of $O(n^2 \log n + mn^3)$ time, i.e., $O(mn^3)$. Lines 12, 13 to 15, 16 and 17 takes $O(mn^2)$ time each. So the overall time complexity of our algorithm is $O(mn^3)$. □

This is in contrast to MCPI [4], the only existing time-triggered scheduling algorithm for mixed-criticality systems on multiprocessors, whose complexity is $O(mn^3 \log n)$.

## 5 Extension for Dependent Jobs

In previous sections, we have discussed instances with independent jobs. Now, we discuss the case of the dual-criticality instances with dependent jobs. In this section, we modify the algorithm given in Section 4 to find the scheduling tables such that if the scheduler discussed in Section 2 dispatches the jobs according to these scheduling tables then it will be a correct online scheduling strategy without disturbing the dependencies between them. There exists an algorithm [4] which can schedule the jobs of an instance $I$ with dependencies with worst-case time complexity $O(En^2 + mn^3 \log n)$, where $n$ is the number of jobs, $E$ the number of edges in the DAG and $m$ the number of processors. We claim that our algorithm has a better worst-case time complexity than the existing algorithm.

### 5.1 Model

We use the same model as discussed in Section 2. Additionally, an instance of a mixed-criticality system containing dependent jobs can be defined as a *directed acyclic graph* (DAG). An instance $I$ is represented in the form of $I(V, E)$, where $V$ represents the set of jobs, i.e., $\{j_1, j_2, \ldots, j_n\}$ and $E$ represents the edges which depict dependencies between jobs. We assume that a HI-criticality job can depend on a LO-criticality job only if the HI-criticality job depends upon another HI-criticality job. This means, there are some instances where an outward edge from a LO-criticality job $j_l$ becomes an inward edge to a HI-criticality job $j_{h1}$ with another inward edge from a HI-criticality job $j_h$ to job $j_{h1}$.

**Definition 2:** A dual-criticality MC instance $I$ with job dependencies is said to be **time-triggered schedulable** on a multiprocessor system if it is possible to construct the two scheduling tables $S_{LO}$ and $S_{HI}$ for each processor of instance $I$ without violating the dependencies, such that the run-time algorithm described in Section 2 schedules $I$ correctly.

### 5.2 The Algorithm

Here we propose an algorithm which can construct two scheduling tables $S_{\mathrm{LO}}$ and $S_{\mathrm{HI}}$ for a dual-criticality instance with dependent jobs. A DAG of mixed-criticality jobs is *MC-schedulable* if there exists a correct online scheduling policy for it. Our algorithm finds a LO-criticality priority order for the jobs of instance $I$ which is used to construct the table $S_{\mathrm{LO}}$. Then the same job allocation order of $S_{\mathrm{LO}}$ is used to construct the table $S_{\mathrm{HI}}$, where HI-criticality jobs have greater priority than LO-criticality jobs, and the HI-criticality jobs are allocated their $C_i$HI units of execution time in $S_{\mathrm{HI}}$ without violating the dependency constraints. The priority between two jobs $j_i$ and $j_k$ is denoted by $j_i \triangleright j_k$, where $j_i$ is higher priority than $j_k$. This priority ordering must satisfy two properties:

- If a node $j_i$ is assigned higher priority than node $j_k$ (i.e., $j_i \triangleright j_k$), then there should not be a path in the DAG from node $j_k$ to node $j_i$.
- If the DAG is scheduled according to this priority ordering then each job $j_i$ of the DAG must finish its $C_i(\mathrm{LO})$ units of execution time before ${d_i}^{\Delta}$.

Now we present the algorithm DP_LoCBP which finds a priority order for mixed-criticality dependent jobs.

Algorithm 2 finds a priority order which is used to construct the scheduling tables for all the processors in steps 1 to 11. First, our algorithm finds the LO-scenario deadline (${d_i}^{\Delta}$) of each job. For the LO-criticality jobs ${d_i}^{\Delta} = d_i$, but ${d_i}^{\Delta} \leq d_i$ for the HI-criticality jobs. Then the algorithm starts to assign the lowest priority jobs from the instance $I$. It always selects the latest deadline job which doesn't have an outward edge as the lowest priority job, but LO-criticality jobs are considered before the HI-criticality jobs. A job $j_i$ can be assigned the lowest priority if and only if all other jobs $j_k$ finish their execution and there remains sufficient time for $j_i$ to complete its $C_i(\mathrm{LO})$ units of execution time before ${d_i}^{\Delta}$. After a job $j_i$ is assigned the lowest priority, it is removed from the instance and added to the priority order $\Psi$. Then the remaining jobs are considered for priority assignment. If at any step a job cannot be assigned a priority, the algorithm declares failure. In step 12, the algorithm constructs the table $S_{\mathrm{LO}}$. In steps 13 to 15, it checks for any possible HI-criticality scenario failure. If it doesn't find a HI-criticality scenario failure, then the priority order constructed by Algorithm 2 can successfully schedule the instance $I$. Then the table $S_{\mathrm{HI}}$ is constructed for each processor by allocating $C_i(\mathrm{HI})$ units of execution time of each HI-criticality job using the same order of allocated jobs as $S_{\mathrm{LO}}$ where a HI-criticality job is given higher priority over LO-criticality jobs. In $S_{\mathrm{HI}}$ each HI-criticality job is allocated its $C_i(\mathrm{LO})$ units of execution time without

**Algorithm 2** DP_LoCBP

**Notation**:
$I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.
**Input** : $I$
**Output** : Tables $S_{\text{LO}}$ and $S_{\text{HI}}$
Assume earliest arrival time is 0.

1: Compute the LO-scenario deadline ($d_i{}^{\Delta}$) of each job $j_i$ as $d_i{}^{\Delta} = d_i - (C_i(\text{HI}) - C_i(\text{LO}))$;
2: **while** $I$ is not empty **do**
3: Assign a LO-criticality latest deadline job $j_i$ which doesn't have an outward edge as the lowest priority job if $j_i$ can finish its execution in the interval $[a_i, d_i{}^{\Delta}]$ after all other jobs finish their execution in LO-scenario under the global EDF scheme;
4: If any LO-criticality job with no outward edge cannot be given the lowest priority then a HI-criticality latest deadline job $j_i$ which doesn't have an outward edge is assigned as the lowest priority job if $j_i$ can finish its execution in the interval $[a_i, d_i{}^{\Delta}]$ after all other jobs finishes their execution in LO-scenario under the global EDF scheme;
5: **if** No job is assigned a lowest priority **then**
6: Declare FAIL and EXIT;
7: **else**
8: Add the job $j_i$ to the priority order $\Psi$.
9: Remove job $j_i$ from the instance and continue;
10: **end if**
11: **end while**
12: Construct table $S_{\text{LO}}$ for each processor $P_i$ using the priority order $\Psi$;
13: **if** $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ **then**
14: return FAIL and EXIT;
15: **else**
16: Construct table $S_{\text{HI}}$ for each processor $P_i$ using the same order of allocated jobs in $S_{\text{LO}}$.
17: The same order as $S_{\text{LO}}$ is followed to allocate the jobs in $S_{\text{HI}}$;
18: After a HI-criticality job $j_i$ is allocated its $C_i(\text{LO})$ execution time in $S_{\text{HI}}$, $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time of job $j_i$ is allocated after the rightmost segment of job $j_i$ in $S_{\text{LO}}$ without violating the dependency constraints and without disturbing the priority order $\Psi$;
19: **end if**

violating the dependency constraints. Once the $C_i(\text{LO})$ units of execution time are allocated for HI-criticality jobs in $S_{\text{HI}}$, the remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time are allocated immediately without disturbing the priority order $\Psi$ and without violating the dependency constraints. At each instant, the allocation is done without violating the dependency constraints.

We illustrate the operation of this algorithm by an example.

**Example 2:** Consider the mixed-criticality instance given in Table 3 which is going to be scheduled on a multiprocessor system having two homogeneous processors, i.e., $P_0$ and $P_1$. The corresponding DAG is given in Fig. 4.

Now we construct a priority order using our proposed algorithm. The LO-criticality scenario $d_i^{\Delta}$ of the jobs $j_1, j_2, j_3, j_4.j_5$ are $3, 3, 3, 2, 4$ respectively. Next we start assigning priorities to each job.

– We start with a node having no outward edges from it. The only such node is job $j_5$. So Algorithm 2 assigns job $j_5$ the lowest priority. If $j_5$ is assigned the lowest priority, then $j_1$ and $j_2$ can run simultaneously in $P_0$ and $P_1$ over $[0, 1]$ and $[0, 1]$ respectively. Then $j_3$ and $j_4$ can run over $[1, 2]$ in $P_0$ and $P_1$ respectively. Then $j_5$ can easily execute its 1 unit of execution on either

**Fig. 3.** Instance for Example 2

| Job | Arrival time | Deadline | Criticality | $C_i$(LO) | $C_i$(HI) |
|---|---|---|---|---|---|
| $j_1$ | 0 | 3 | LO | 1 | 1 |
| $j_2$ | 0 | 3 | LO | 1 | 1 |
| $j_3$ | 0 | 3 | LO | 1 | 1 |
| $j_4$ | 0 | 4 | HI | 1 | 3 |
| $j_5$ | 1 | 6 | HI | 1 | 3 |

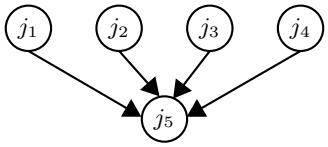


**Fig. 4.** A DAG showing job dependency among the jobs given in Table 3

$P_0$ or $P_1$ over $[2, 3]$ to finish by its LO-scenario deadline ($d_i^{\Delta}$). Now we can assign job $j_5$ the lowest priority job.

We remove job $j_5$ and consider $\{j_1, j_2, j_3, j_4\}$ to find the next lowest priority job.

– Since the LO-criticality jobs are given the lowest priority by the proposed algorithm, it is easy to verify that the successive lowest priority jobs will be $j_1, j_2$ and $j_3$ respectively. Finally, $j_4$ is the highest priority job.

So the final priority order of jobs in instance $I$ is $j_4 \triangleright j_3 \triangleright j_2 \triangleright j_1 \triangleright j_5$. The table $S_{\text{LO}}$ for each processor is given in Fig. 5.

Now the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine checks for all possible HI-criticality scenarios. We can check that all HI-criticality scenarios are schedulable using the priority order $j_4 \triangleright j_3 \triangleright j_2 \triangleright j_1 \triangleright j_5$ of the instance $I$. Finally, the table $S_{\text{HI}}$ is constructed for each processor by allocating $C_i$(HI) units of execution time of each HI-criticality job using the same order of allocated jobs in $S_{\text{LO}}$ where a HI-criticality job is given higher priority over a LO-criticality job. On processor $P_0$, the job order of $S_{\text{HI}}$ remains the same as in $S_{\text{LO}}$. Job $j_4$ is a HI-criticality job and doesn't depend on any other job, so it is allocated its $C_i$(LO) units of execution time over $[0, 1]$ and the remaining $C_i$(HI) $-$ $C_i$(LO) units of execution time are allocated in the interval $[1, 3]$. Job $j_5$ is allocated in the interval $[2, 3]$ in table $S_{\text{LO}}$ of $P_0$. But $j_5$ is allocated in the interval $[3, 6]$ due to dependency constraints which doesn't affect the scheduling after a mode change. On processor $P_1$, job $j_3$ and $j_2$ (LO-criticality) which don't depend on any other jobs, are allocated their one unit of execution time in the intervals $[0, 1]$ and $[1, 2]$ respectively. The table $S_{\text{HI}}$ for each processor is given in Fig. 6. □

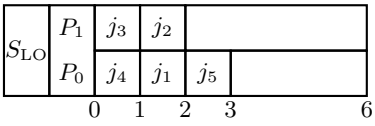


**Fig. 5.** Table $S_{\text{LO}}$ for processor $P_0$ and $P_1$

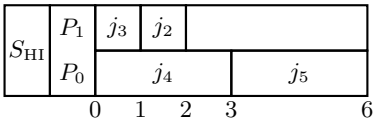


**Fig. 6.** Table $S_{\text{HI}}$ for processor $P_0$ and $P_1$

### 5.3 Comparison with MCPI algorithm

**Theorem 4:** An instance $I$ is schedulable by the MCPI algorithm [4] if and only if it is schedulable by our algorithm.

*Proof.* $\Rightarrow$ We need to show that if MCPI generates a priority order for an instance $I$, then our algorithm will always find a priority order for instance $I$ and the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine will not fail.

Suppose the MCPI algorithm finds a priority order for instance $I$. Now the lowest priority job of the priority order (according to the MCPI algorithm) can be either a LO-criticality or HI-criticality job. First, we prove the case where a job is LO-criticality and then HI-criticality. Let $j_i$ be the lowest priority job and its criticality be LO which means no other job depends on $j_i$. So at the time of construction of table $S_{\text{LO}}$, every higher priority job $j_k$ finishes its $C_k(\text{LO})$ units of execution time without violating the dependency constraints and there remains sufficient time for the lowest priority job $j_i$ to finish its $C_i(\text{LO})$ units of execution time in the interval $[a_i, (d_i^{\Delta})]$. So this condition is the same as our proposed algorithm.

Let job $j_i$ be the lowest priority, and its criticality be HI which means no other job depends on $j_i$. Since MCPI successfully finds the priority order, it must have checked all the scenarios and doesn't find any failure in the HI-scenario situations. After every higher priority job $j_k$ finishes its $C_k(\text{LO})$ units of execution time, there remains sufficient time for the lowest priority job $j_i$ to finish its $C_i(\text{LO})$ units of execution time in the interval $[a_i, d_i^{\Delta}]$ without violating the dependency constraints. The HI-criticality jobs need to finish their LO-criticality execution on or before $d_i^{\Delta}$ in LO-scenario, so that they have sufficient time to finish their remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time before their deadline $d_i$. This condition doesn't violate the dependency constraints as it is the job which doesn't have an outward edge from it. So this condition is the same as our proposed algorithm.

Then $j_i$ is removed from the instance $I$ and the next priority can be assigned from the remaining jobs. We can argue in the same way for the remaining jobs. From the above argument, it is proved that our proposed algorithm finds the same priority order, for instance $I$ as the MCPI algorithm. Since the priority order is the same and MCPI doesn't find any HI-scenario or LO-scenario failure, $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine in our algorithm will not fail as well. Thus, for a MCPI schedulable instance, our algorithm can also construct priority tables $S_{\text{LO}}$ and $S_{\text{HI}}$.

($\Leftarrow$) Our algorithm generates a priority order for instance $I$ which is used to find the table $S_{\text{LO}}$. When a mode change occurs, our algorithm uses the table $S_{\text{HI}}$ which is constructed from the job ordering in $S_{\text{LO}}$ to schedule the HI-criticality jobs. We need to show that if our algorithm generates a priority order for instance $I$, then the MCPI algorithm will always find a priority order and the $anyHIScenarioFailure(PT, T)$ subroutine will not fail.

Suppose our algorithm finds a priority order, for instance $I$. The lowest priority job assigned by our algorithm can be either a HI-criticality or a LO-criticality job. First, we consider the case where a job is LO-criticality. Let $j_i$ be the lowest priority job, and its criticality be LO which means the job $j_i$ can finish its execution between its arrival time and deadline after all other job finishes their execution without violating the dependency constraints. So according to the priority table ($SPT$) of MCPI, job $j_i$ can be given the lowest priority among the LO-criticality jobs. Since the job can meet its deadline after all other jobs finished their execution, the $PullUp()$ subroutine will pull up the HI-criticality jobs upward in the priority tree. So according to the MCPI algorithm, the job $j_i$ is the lowest priority job among the HI-criticality jobs as well. This shows that the job $j_i$ is the lowest priority job according to the MCPI algorithm.

Let $j_i$ be the lowest priority job, and its criticality be HI which means the job $j_i$ can finish its execution between its arrival time and deadline after all other job finishes their execution without violating the dependency constraints. Since our algorithm prefers LO-criticality jobs to assign the lowest priority over HI-criticality jobs, there are no LO-criticality jobs available which can be assigned lower priority than job $j_i$. Our algorithm chooses the job with no outward edges which means no job depends on the lowest priority job. So due to the dependency constraints, all the LO-criticality jobs finish before job $j_i$. Since no LO-criticality job can finish its execution after the execution of job $j_i$, the $PullUp()$ subroutine will not be able to pull up the HI-criticality jobs upward in the priority tree. So job $j_i$ is the lowest priority job according to the MCPI algorithm.

In the same way, we argue for the next priority assignment of jobs of instance $I$. □

**Theorem 5:** The computational complexity of DP-LoCBP (Algorithm 2 on page 11) is $O(nE + mn^3)$, where $n$ is the number of jobs, $E$ the dependency relations among the jobs in the instance $I$ and $m$ the number of processors in the system.

*Proof.* Line 1 takes $O(n)$ time. In lines 3 - 4, traversing each edges takes $O(E)$ time, simulation of global EDF on $m$ processors takes $O(mn^2)$ times [12]. So the total time taken by lines 3 and 4 is $O(E + n \log n + mn^2)$. Lines 5 to 9 take $O(1)$ time in each execution of the loop body. Since the while loop in line 2 runs $n$ times, lines 3 to 9 require a total of $O(nE + n^2 \log n + mn^3)$ time, i.e, $O(nE + mn^3)$ time each. Lines 12, 13 to 14, 16 and 17 to 18 takes $O(mn^2)$ time each. So the overall time complexity of our algorithm is $O(nE + mn^3)$. □

This is in contrast to the MCPI algorithm [4], the only existing time-triggered scheduling algorithm for the dependent jobs of mixed-criticality systems on multiprocessors is $O(n^2E + mn^3 \log n)$.

## 6 Results and Discussion

In this section, we present the experiments conducted to evaluate the LoCBP algorithm for the dual-criticality case. The experiments compare the running times of LoCBP and MCPI. The comparison is done over numerous instances with randomly generated parameters.

The job generation policy may have significant effect on the experiments. The details of the job generation policy are given below.

- The utilization ($u_i$) of the jobs of instance $I$ are generated according to the Staffords randfixedsum algorithm [13].
- We use the exponential distribution proposed by Davis *et al* [14] to generate the deadline ($d_i$) of the jobs of instance $I$.
- The $C_i$(LO) units of execution of the jobs are calculated by $u_i \times d_i$.
- The $C_i$(HI) units of execution of the jobs are calculated as $C_i(\text{HI}) = \text{CF} \times C_i(\text{LO})$ where CF is the criticality factor which varies between 2 and 6 for each HI-criticality job $j_i$ in our experiments.
- Each instance $I$ contains at least one HI-criticality job and one LO-criticality job. We have generated random instances for 2, 4, 8 and 16 processors, where each instance has atleast $m+1$ number of jobs. Each instance is LO-scenario schedulable. We have used an intel core 2 duo processor machine with speed of 2.3 Ghz to conduct the experiments.

In the first experiment, we fix the number of processors to 2 and let the deadline of the jobs vary between 1 and 2000. The graph in Fig. 7 shows the time consumption by each schedulable instances from different numbers of randomly generated instances.

From the graph in Fig. 7, it is clear that our algorithm consumes significantly less time than the MCPI algorithm. As can be seen from Fig. 7, for a multiprocessor with two processors the time consumption by MCPI is much higher than our algorithm. The ratio of time consumed also increases with the increase of number of jobs per instance and is close to five for 1000 jobs. In another experiment, we have shown that the time consumption decreases for $m = 4$, but the ratio of time consumed by our algorithm in comparison to the MCPI algorithm is very much similar to the case $m = 2$, as can be seen in Fig. 8.

## 7 Conclusion

In this paper, we proposed a new algorithm for time-triggered scheduling of mixed-criticality jobs for multiprocessor systems. We proved that our algorithm has a better worst-case time complexity than the previous algorithm (MCPI). We

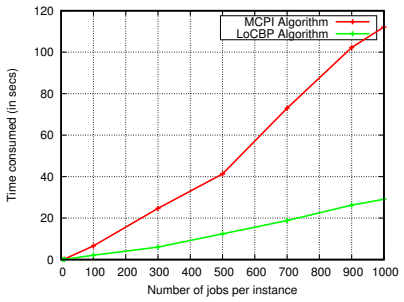


**Fig. 7.** Comparison of time consumption of MC-schedulable instances for $m = 2$

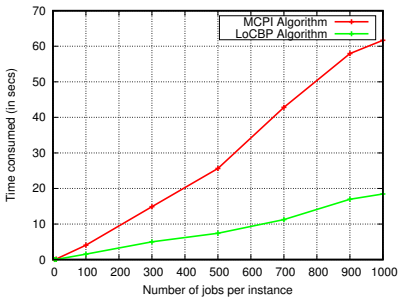


**Fig. 8.** Comparison of time consumption of MC-schedulable instances for $m = 4$

also proved the correctness of our algorithm. Then we extended our algorithm for dependent jobs and compared the worst-case time complexity with the existing algorithm. We examined the theoretical result by comparing the actual time consumption between LoCBP and MCPI.

## References


1. S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium, 2007. RTSS 2007.*, pages 239–243, Dec 2007.
2. S. Baruah, V. Bonifaci, G. D'Angelo, Haohan Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, Aug 2012.
3. Kimon P Valavanis. *Advances in unmanned aerial vehicles: state of the art and the road to autonomy*, volume 33. Springer Science & Business Media, 2008.
4. D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed-critical scheduler on single and multi-processor platforms. In *HPCC / CSS /ICESS*, pages 684–687, Aug 2015.
5. Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 3–12. IEEE, 2011.
6. D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Mixed critical earliest deadline first. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 93–102, July 2013.
7. Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.
8. G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Mapping mixed-criticality applications on multi-core architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.
9. G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 17:1–17:15. IEEE Press, 2013.

10. R. M. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *24th Euromicro Conference on Real-Time Systems*, pages 309–320, July 2012.
11. WA Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.
12. Houssine Chetto, Maryline Silly, and T Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.
13. Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.
14. Robert I. Davis, Attila Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers,*, 57(9):1261–1276, 2008.