Directed Acyclic Graph Scheduling for Mixed-Criticality Systems

Roberto Medina(✉), Etienne Borde, and Laurent Pautet

LTCI, Télécom ParisTech, Université Paris-Saclay, 75013 Paris, France
{roberto.medina,etienne.borde,laurent.pautet}@telecom-paristech.fr

Abstract. Deploying safety-critical systems into constrained embedded platforms is a challenge for developers who must arbitrate between two conflicting objectives: software has to be safe and resources need to be used efficiently. Mixed-criticality (MC) has been proposed to meet a trade-off between these two aspects. Nonetheless, most task models considered in the literature of MC scheduling, do not take into account precedence constraints among tasks. In this paper, we propose a multi-core scheduling approach for a model presenting MC tasks and their dependencies as a Directed Acyclic Graph (DAG). We also introduce an evaluation framework for this model, released as an open source software. Evaluation of our scheduling algorithm provides evidence of the difficulty to find correct scheduling for DAGs of MC tasks. Besides, experimentation results provided in this paper show that our scheduling algorithm outperforms existing algorithms for scheduling DAGs of MC tasks.

Keywords: Mixed-Criticality · Directed acyclic graphs · Mode transition · Real-time scheduling

1 Introduction

Having certified software is imperative to deploy applications in safety-critical systems. To ensure that critical tasks always meet their timing requirements (*i.e.* deadline), Certification Authorities (CA) require an overestimated Worst-Case Execution Times (WCET).

The Mixed-Criticality (MC) model was proposed in [15] to guarantee safety while efficiently using embedded resources. In this model, tasks with different *criticality* levels share the same hardware platform. This model ensures that, (i) *high-criticality* tasks of the system always perform their execution within their deadlines and (ii) resources are efficiently used by redistributing WCET overestimation of *high-criticality* tasks to *low-criticality* tasks.

Nonetheless, data dependencies between tasks on a MC model has seen very few contributions [6]. Our model represents MC tasks with a Directed Acyclic Graph (DAG), where vertices represent tasks and edges represent precedence constraints among them. Vertices that are not related by an edge can be executed in parallel. This is very interesting since embedded platforms use multi-core architecture nowadays.

© Springer International Publishing AG 2017

In this paper, we propose a new approach based on static List Scheduling (LS) to schedule DAGs of MC tasks for multi-core architectures. We also propose a random generation method of MC-DAGs[1], used to evaluate our scheduling algorithm. Our evaluation shows that our scheduler has a better schedulability rate compared to the reference algorithm of the literature [2].

The remainder of the paper is organized as follows: Sect. 2 presents the task model used in our contribution. The main difficulties that need to be overcome by our scheduling approach are presented in Sect. 3. In order to schedule the MC-DAG on multi-core architectures we propose a new scheduling algorithm in Sect. 4. The implementation of the MC-DAG test generator is described in Sect. 5. Section 6 presents the evaluation of our algorithm on the generated MC-DAG tests. Related works are discussed in Sect. 7 and we conclude in Sect. 8 with future research perspectives.

2 Task Model

In this section we present the task model our contribution relies on: DAGs of MC tasks. DAGs and the synchronous model of computation are widely used in industrial tools like SCADE from Esterel, Simulink from MathWorks among others. Therefore, including the MC approach to this model is of great interest.

2.1 Mixed-Criticality Tasks

MC scheduling [6] has become an appealing solution to integrate various tasks with different levels of criticality onto the same hardware platform.

MC scheduling was first presented by Vestal in [15]. Vestal's task model is based on the following observation: the higher the criticality level becomes, the more overestimated the WCET is. For instance, in a low criticality level tasks could have their WCET determined empirically (*i.e.* measuring execution times over multiple executions). While on a high criticality level, code coverage analysis and validation from a CA to determine a pessimistic WCET that cannot be exceed at any time, is required. Therefore the low criticality levels have a smaller WCET than high criticality levels. In order to mitigate the impact of overestimated WCET on resource dimensioning, MC models propose to identify operational modes, and to define different timing configurations of tasks for each operational mode. In particular when high-criticality tasks exceed their WCET of a low-criticality mode, the system performs a mode transition into a high-criticality mode where low-criticality tasks are stopped (discard approach) or have less processing power (elastic approach). However, this mode transition needs to be safe: **deadlines of high-criticality tasks must still be satisfied.**

We consider MC systems with two operational modes noted HI and LO. When the system is in LO mode (initial mode), all tasks can be executed on the platform until their WCET in LO mode (noted $C_i(LO)$ for a task τ_i). For

[1] Open source: https://github.com/robertoxmed/ls_mxc.

each task τ_i, $C_i(LO) \leq C_i(HI)$ ($C_i(HI)$ is the WCET of τ_i in mode HI, it is a pessimistic WCET). If a task is able to complete its execution before its $C_i(LO)$, we suppose all estimated time budget is used, *i.e.* the processor would be idle until the $C_i(LO)$ is consumed. A Timing Failure Event (TFE) occurs when a task τ_i runs for a longer time than its $C_i(LO)$, and the occurrence of a TFE triggers a mode switch from LO to HI mode. In HI mode, tasks considered as highly critical (noted as HI tasks) are able to run until $C_i(HI)$ while lower criticality tasks (noted as LO tasks) are stopped: we adopt the discard approach.

2.2 DAG Mixed-Criticality Model

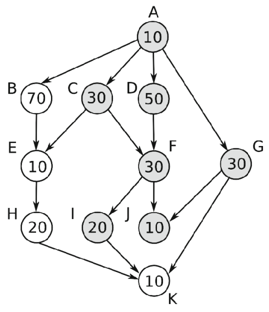
In addition to their criticality level, we consider tasks with precedence constraints modeled as DAGs. This representation allows us to identify clearly which parts of a computation can be run in parallel. Multi-core platforms are more and more used in embedded systems, parallel computation is an important challenge to improve resource usage. At the same time, MC studies often use independent task sets, however real applications are most likely going to have tasks communicating with each other. For example, Simulink and SCADE are tools that are used for designing and implementing embedded control systems [12]. We consider real-time systems modeled with **a single-DAG**, *i.e.* only one DAG of mixed-critical tasks is being executed by the platform. All tasks forming the DAG are constraint to meet a single deadline, can be preempted and can migrate from one CPU to another.

Our model, noted MC-DAG, is composed of tasks represented by vertices in the graph. Precedence constraints are materialized by edges. Each task τ_i is characterized by a criticality level $\chi_i \in \{\text{LO, HI}\}$, a WCET in LO mode $C_i(LO)$ and a WCET time in HI mode $C_i(HI)$ ($C_i(HI) = 0$ if $\chi_i = \text{LO}$). HI criticality tasks cannot depend on the output of LO criticality tasks for safety reasons: if the LO tasks fails to deliver its output, the HI criticality task can be compromised. For this reason we only allow three types of communications in our model: from HI to HI, from HI to LO and from LO to LO. Industrial standards like ARINC653 also apply this communication constraint for partitions for example.

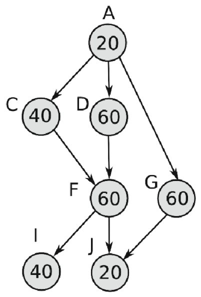
In the remaining, we shall illustrate our contribution with the example of the MC-DAG presented in Fig. 1. White vertices represent LO criticality tasks and gray vertices are HI criticality tasks. Numbers on each vertex represent the execution times of tasks. The graph on the left has WCET for tasks in LO mode, while the right graph gives the WCET of HI tasks in HI mode. WCET are presented in Time Units (TU).

3 Problem Statement

Scheduling MC tasks on multi-core platforms is a difficult problem, specially due to transitions to higher criticality modes: deadlines of high-criticality tasks must be met, even when a TFE occurs. This scheduling problem becomes even more complex when there exist precedence constraints between tasks: if a task



(a) DAG in LO mode



(b) DAG in HI mode

Fig. 1. Mixed-Criticality DAG example.

increases its WCET due to a switch to HI mode, all its successors are delayed in a domino effect.

In this paper, we aim at making sure a safe scheduling of a given MC-DAG exists. A MC system is considered to be safe if (i) tasks meet their deadlines in HI and LO modes, and (ii) the mode transition from LO to HI is safe: HI tasks meet their deadline even when a TFE occurs: the incrementation of the WCET (from $C_i(LO)$ to $C_i(HI)$) for HI tasks cannot compromise their deadline. Allocating MC tasks with data dependencies to a multi-core architecture is equivalent to an optimization problem that aims at minimizing the execution time of a MC-DAG while enforcing safe mode transitions for each possible date of a TFE. Ensuring the safety of a LO to HI mode transition is therefore a challenging objective.

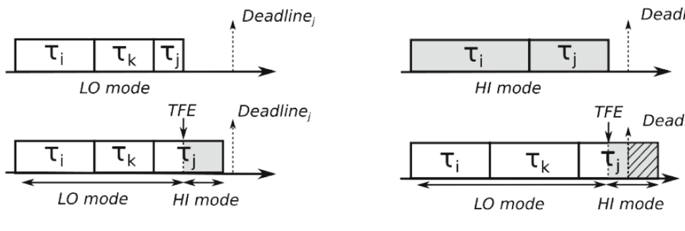


Fig. 2. Safe and unsafe mode transitions

Figure 2 provides scenarii for safe and unsafe mode transitions. The HI task τ_i completes its execution. Then, both the HI task τ_j and the LO task τ_k become ready to execute. Let us assume LO task τ_k starts its execution before the HI task τ_j. A TFE occurs during the execution of task τ_j. In other words, τ_j executes for a longer time than its LO WCET, $C_j(LO)$. Thus, a mode switch occurs. τ_k is stopped and τ_j's WCET is extended up to its HI WCET, $C_j(HI)$. In these two scenarii, the WCET $C_k(LO)$ differs: the WCET in the second scenario is greater than the one in the first scenario. In the first scenario, illustrated on the low left part of Fig. 2, the deadline of task τ_j is satisfied during the transition. In the second scenario, $C_k(LO)$ is greater and when the TFE occurs, task τ_j has not enough execution time available and eventually causes a deadline miss.

In addition to this problem, another issue is to evaluate scheduling algorithms for MC-DAGs in multi-core architectures. Since the MC-DAG model has mostly seen theoretical contributions, there does not exist yet a benchmarking framework to evaluate such DAG scheduling algorithms. Besides, benchmarking frameworks for MC-DAG have to consider lots of parameters that influence the degree of parallelism of tasks, the distribution of CPU utilization among tasks, the number of cores that are assigned to the DAG, as well as the topology of graph.

In the rest of this paper, we present the technical solutions we propose to answer these difficult problems.

4 Multi-core Scheduling for MC-DAGs

4.1 Scheduling Algorithms for DAGs on Multi-cores

Finding an optimal scheduling of DAGs on a limited number of processors (respecting the deadline and minimizing the scheduling time of the DAG) is a NP-complete problem [9]. List Scheduling (LS) is a polynomial approach to find near-optimal scheduling. It aims at producing a static scheduling for DAGs that minimizes the completion time of the DAG, also called *makespan*. Several different heuristics are based on LS and improve the resulting scheduling under some hypothesis. As explained before, the MC-DAG model increases the complexity of DAG scheduling, mainly because of the safety constraints on the LO to HI mode transition (satisfying the deadline of HI tasks after their WCET is incremented).

In order to reduce the complexity of this problem, we consider the following hypotheses: the scheduler executes tasks with a time-triggered semantics, and tasks execute exactly for their WCET: if tasks finish before their WCET, idle time is enforced at run time. These hypotheses increase the determinism of the schedule and reduce the number of possible scheduling scenarii, as well as the number of instants for which TFEs may occur.

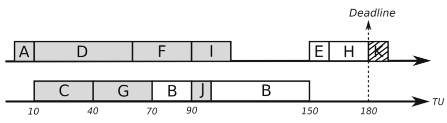
4.2 Scheduling Algorithms for MC-DAG on Multi-cores

Scheduling MC-DAGs was first proposed by Baruah for uni-core architectures in [4]. LS was then used for scheduling this model into multi-core platforms [2]. The idea is to build two scheduling tables: one per mode. In HI mode, a static scheduling allocates time slots to HI tasks in a table by applying LS to the DAG in HI mode. In LO mode, another static scheduling table allocates time slots to tasks almost the same way. However, time slots are allocated for HI tasks as soon as they are ready. Then, time slots are allocated to LO tasks according to

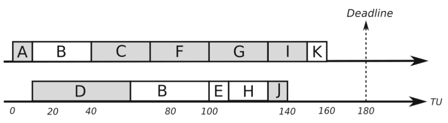
LS when they are not preempted by HI tasks. In case a TFE occurs, the system switches to HI mode and applies the scheduling table in HI mode. Since HI tasks are always scheduled prior to LO tasks, there is a guarantee to have a safe mode transition.

We applied this approach to schedule MC-DAGs on uni-core architectures [11]. The objective was to evaluate availability of MC-DAG systems enriched with our recovery mechanisms to switch back to LO mode. In this context, we realized that preempting LO tasks as soon as a HI task is ready can produce inefficient schedules and could result in deadline misses whereas valid schedules exist.

Figure 3 highlights the problem with this approach. Figure 3a provides the schedule obtained with Baruah's method on the MC-DAG we described on Fig. 1. On this example, the deadline (180 TUs) is missed. Figure 3b illustrates the schedule obtained with our method on the same example.



(a) Baruah's MC DAG scheduling



(b) Valid MC DAG scheduling

Fig. 3. Scheduling tables for the MC DAG

Intuitively, the main idea of our approach is to privilege HI tasks over LO tasks, **only** at specific instants for which it is required to prioritize HI tasks in order to ensure a safe mode transition. These specific instants are called Latest Safe Activation Instant (LSAIs). The main steps of our algorithm consist in calculating the HI scheduling table (S_{HI}) starting from the deadline of the graph. This way we obtain a LSAI for each HI task. The LO mode scheduling table (S_{LO}) is then obtained thanks to LS with preemption of HI tasks at their LSAI. We explain this method in the remaining of this section: in Sect. 4.3, we define a necessary condition in order to ensure mode transitions are safe. As explained in Sect. 4.4, we use this definition to compute the LSAIs of HI tasks in LO mode, as well as the scheduling table of HI tasks in HI mode. Finally, we explain in Sect. 4.5 how we compute the scheduling table of tasks in LO mode.

4.3 Safe Mode Transition: Necessary Condition

We introduce in this section a necessary condition (see Eq. (1)) ensuring that a mode transition can be performed safely. This condition is then used in Sect. 4.4 to compute (i) the LSAIs of HI tasks in LO mode, and (ii) the scheduling of HI tasks in HI mode.

$$\forall \tau_i \in \tau_{HI}, WCRT_i(LO) + C_i(HI) - C_i(LO) \leq D. \tag{1}$$

In this equation, τ_{HI} is the set of HI tasks of the graph and D is the deadline of the graph. $WCRT_i(LO)$ corresponds to the Worst Case Response Time, that is the time required for task τ_i to finish its execution in LO mode. The intuition behind Eq. (1) is that, for all $\tau_i \in \tau_{HI}$, τ_i has enough time to finish its execution in mode HI in case a TFE occurs while executing τ_i in LO mode. Figure 4 gives an illustration of Eq. (1):

– the upper part of the figure illustrates a scheduling scenario in which Eq. (1) is not respected, leading to a deadline miss;
– the lower part of the figure illustrates a scheduling scenario in which Eq. (1) is respected. This illustration helps to understand why, in the worst case (*i.e.* the time of the TFE equals $WCRT_i(LO)$) the deadline is still met after the transition to HI mode.

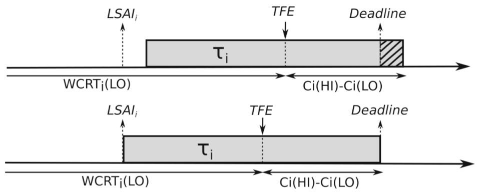


Fig. 4. Example of unsafe mode transition

Respecting Eq. (1) forces HI tasks in LO mode to start at a time that allows τ_i to switch to HI mode safely. The LSAI of τ_i, an exit vertex of the MC-DAG in HI mode, is illustrated on Fig. 4. As one can see on the figure, computing the LSAI of an exit vertex of the DAG is very easy: it is a straightforward application of Eq. (1). In the next subsection, we explain how we compute the LSAIs of all the HI tasks, which boils to compute the scheduling table of the MC-DAG in HI mode.

4.4 Building the HI Mode Table

We schedule HI tasks in HI mode **as late as possible** in order to compute the LSAIs of HI tasks in LO mode, leaving as much time as possible to schedule LO tasks in LO mode. In order to do so, we first compute the LSAIs of exit vertices as illustrated on Fig. 4. The LSAI of a task τ_i then becomes the virtual deadline for the predecessors of τ_i and we can compute the LSAIs of these predecessors by applying Eq. (1) with this virtual deadline. In other words, we *reverse schedule* the MC-DAG in HI mode: from the *deadline* of the graph, backwards time 0, we allocate time slots for HI tasks in HI mode. As a result, we obtain a scheduling table of HI tasks in HI mode, called S_{HI}, and the starting date of a task in this table corresponds to its LSAI. However, in order to start HI tasks in HI mode **as late as possible** we need to minimize the makespan of the *reverse schedule.*

For this purpose, we use a LS algorithm called Highest Level First with Estimated Time (HLFET) [1]. Indeed, HLFET is the most efficient LS algorithm to schedule DAGs on multi-cores [9]: HLFET is less computationally expensive than other LS algorithms (which are in general of polynomial complexity), and provides near-optimal makespans. In HLFET, the *level* of a vertex is given by the longest path from that vertex to an exit vertex, and the level of an exit vertex is equal to its execution time. For example, applying HLFET to the graph presented in Fig. 1b leads to the following levels for HI tasks: $\langle(A, 180), (C, 140), (D, 160), (F, 100), (G, 80), (I, 40), (J, 20)\rangle$.

Algorithm 1. S_{HI} computation

```
 1: function CalcSHI
 2:     Calculate levels for each vertex in HI mode
 3:     for all HI tasks $\tau_i$ do
 4:         $RET[\tau_i] \leftarrow C_i(HI)$ /* Remaining execution time*/
 5:     $t \leftarrow Deadline$
 6:     while $t > 0$ do
 7:         for all cores $c$ do
 8:             $\tau \leftarrow$ lowest level task s.t. all successors have been fully scheduled
 9:             $S_{HI}[t][c] \leftarrow \tau$
10:             $RET[\tau] \leftarrow RET[\tau] - 1$
11:             if $RET[\tau] = 0$ then $LSAI[\tau] \leftarrow t$
12:         $t \leftarrow t - 1$
13:         if $\sum RET > t * NbCores$ then
14:             return NotSchedulable
15: return $S_{HI}$ and $LSAI$
```

Algorithm 1 describes the algorithm we propose to compute the scheduling table of HI tasks in HI mode, called S_{HI}. The first step of the algorithm is to compute the levels of tasks with HLFET (using $C_i(HI)$ for their execution time). We build S_{HI} starting from the deadline and from the exit vertices of the DAG in HI mode. For each time slot, we schedule on each core the tasks (i) having the lowest level, and (ii) having all their successors completely scheduled. If all the tasks are completely scheduled before time 0 is reached, the system is schedulable in mode HI and table S_{HI} provides its scheduling in mode HI. Besides, the start date of τ_i in this table is also the LSAI of τ_i.

We illustrate the execution of this algorithm on the DAG provided in Fig. 1b. We assume we have two cores to execute this DAG and its deadline is 180 TUs. At the beginning we have two exit vertices with no successors: I and J. Tasks I and J are selected, they have the lowest levels. Once J has been fully allocated, at time 160 TU, the only task that can be executed is G (F has to wait until I is completely scheduled). AT 140 TU, F is scheduled until 80 TU, activating tasks C and D. Once D has been fully allocated, task A executes from TU 20 to 0. The final S_{HI} table is shown in Fig. 5, and the DAG is schedulable in HI mode. With S_{HI}, we also obtain the LSAIs of each task, depicted with vertical

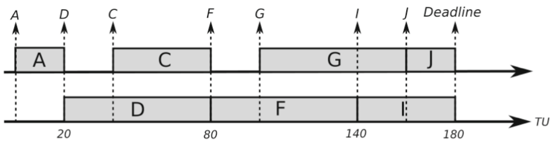


Fig. 5. S_{HI} table

arrows in Fig. 5: 160 for I, 140 for J, 100 for G, 80 for F, etc. As explained in the next section, these LSAIs are then used to calculate the scheduling table of tasks in LO mode, called S_{LO}.

4.5 Building the LO Mode Table

Algorithm 2 describes the algorithm we propose to compute the scheduling table of HI and LO tasks in LO mode. First, we calculate tasks levels using HLFET (using $C_i(LO)$ for tasks execution time). Then, we start allocating time slots to tasks from time 0, scheduling tasks towards the deadline. We schedule tasks with the highest level first, but we promote a HI task when its LSAI is reached. LSAI behaves as a virtual deadline that guarantees safe mode transitions. Promoted tasks preempt other tasks, and execute until completion. Preempted tasks can be resumed in another processor since task migration is allowed in our model. As explained previously, we thus guarantee that Eq. 1 is satisfied for all HI task.

Algorithm 2. S_{LO} computation

```
 1: function CalcSLO
 2:     Calculate levels for each vertex in LO mode
 3:     for all tasks $tau_i$ do
 4:         $RET[\tau_i] \leftarrow C_i(LO)$ /* Remaining execution time */
 5:     $t \leftarrow 1$
 6:     for all timeslots $t \leq Deadline$ do
 7:         if $t$ is a LSAI then promote the corresponding HI task(s)
 8:         for all cores $c$ do
 9:             $\tau \leftarrow$ highest level task s.t. all predecessors have been fully scheduled
10:             $S_{LO}[t][c] \leftarrow \tau$
11:             $RET[\tau] \leftarrow RET[\tau] - 1$
12:         if $\sum RET > (Deadline - t) * NbCores$ then
13:             return SchedulingException
14: return $S_{LO}$
```

Considering the MC-DAG provided in Fig. 1a, the levels of each task are given by: $\langle (A, 120), (B, 110), (C, 90), (D, 110), (E, 40), (F, 60), (G, 40), (H, 30), (I, 30), (J, 10), (K, 10) \rangle$.

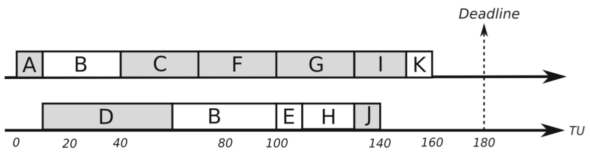


Fig. 6. S_{LO} table

Time 0 TU is a LSAI for task A so it is promoted and starts its execution. At 10 TUs, tasks B, C, D and G can run and no LSAI occurs. Thus, we select B and D, the tasks with the highest levels. At 20 TUs, we have a LSAI for D. However, D is already running so there is no preemption at this point. But, at 40 TUs, it is a LSAI for C which preempts B. At 60 TUs, D has finished its execution, B is resumed (on a different processor). Once C has finished its execution at 70 TUs, F runs until it completes at 100 TUs in parallel with B. Tasks G, E have met their precedence constraints and it is a LSAI for G, so G and E are selected. We continue this procedure until there are no more tasks to schedule. The final S_{LO} is presented in Fig. 6. Vertical arrows correspond to LSAIs of HI tasks.

In this section, we presented a scheduling approach for MC-DAGs. To evaluate this approach, we propose the benchmarking tool described in Sect. 5.

5 Mixed-Criticality DAG Synthesis

To evaluate our scheduling algorithm we need a benchmarking tool that automatically generates a significant number of MC-DAGs. No such tool is available in the literature: contributions to this subject have only presented theoretical work [2] or evaluation frameworks have not been released publicly [10]. We explain the different aspects we considered for developing our benchmarking tool.

The benchmarking tool takes into account different aspects of the various communities that are part of our work: DAG, Real-time on multi-core architectures, and MC scheduling.

- The objective of generating graphs randomly is to avoid topologies that might influence schedulability. We developed a DAG generation tool based on the Erdös-Rényi's method, which has been used in several research on DAGs scheduling for real-time systems [8,14].
- An important parameter for scheduling tasks sets on multi-core systems is the *utilization*. Distribution of utilization [5] is a method widely used in the real-time systems domain in order to benchmark scheduling algorithms. Task sets can be generated quite efficiently with a uniform distribution of utilization among tasks. However, these methods usually create independent tasks, *i.e.* without precedence constraints among them.
- When it comes to MC in DAG generation tools, it is important to parameterize the utilization of HI and LO tasks, as well as utilization of HI tasks in LO mode.

Our generator has three main stages: (i) generation of the DAG of HI tasks, (ii) reduction of HI tasks utilization in LO criticality mode, and (iii) completion of the DAG with LO tasks. The following parameters are used by the tool: e is the probability of having an edge between two vertices. p the maximum degree of "parallelism" in the DAG, *i.e.* the maximum number of vertices that are not transitively connected by an edge. CP, the critical path of the graph, *i.e.* the longest path in the DAG between an entry vertex to an exit vertex. U_{HI}, the utilization (of HI tasks) in HI mode. U_{HIinLO}, the utilization of HI tasks in LO mode ($U_{HIinLO} < U_{HI}$). U_{LO}, the utilization (of all tasks) in LO mode.

The first step of the MC-DAG generation creates the DAG in HI mode using a parameter U_{HI}. This step is iterative: each iteration adds vertices until the utilization U_{HI} is reached. More precisely, we create in each iteration a random number (between 1 and p) of vertices. When creating vertices, we distribute U_{HI} by giving each vertex a $C_i(HI)$. An edge can be added between two vertices, with a probability e, if (i) the vertices were created in different iterations (enforcing the degree of paralellism p), and (ii) if adding the edge does not increase the critical path (thus enforcing parameter CP).

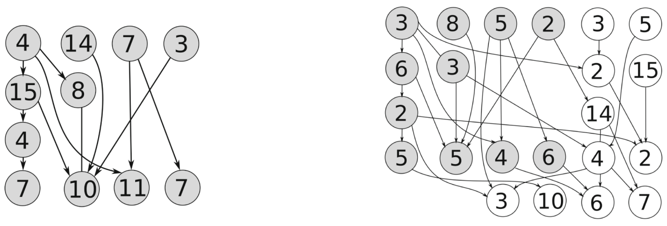
As a second step, we generate the LO part of HI tasks. Parameter U_{HIinLO} gives an upper bound of the utilization of HI tasks in LO mode. HI tasks' $C_i(LO)$ is randomly generated between 1 and a bound starting at $C_i(HI)$.

We iteratively try a $C_i(LO)$ for each task and check if U_{HIinLO} is satisfied after the reduction. If it is not the case, a new iteration tries other values for $C_i(LO)$, but this time between 1 and the previous value tested. As a consequence, values for $C_i(LO)$ decrease until U_{HIinLO} is satisfied. In addition, if all HI tasks become unitary (*i.e.* $C_i(LO) = 1$) the reduction phase stops.

On the last step of the generation, LO tasks are added to the graph. We distribute a utilization of $U_{LO} - U_{HIinLO}$ to LO tasks (U_{HIinLO} is the final real value obtained after the reduction phase). We use a process similar to the one used in step one in order to complement the DAG of HI tasks. However, we prevent the process from adding edges from LO tasks to HI tasks. Finally, we check whether the CP was reached while creating the DAG. If not, we add a last task (either HI or LO) that completes the CP.

Our benchmarking tool is open sourced and can be found on GitHub[2]. Figure 7 shows a MC-DAG that was created with our generator. HI tasks are presented in gray and LO tasks are presented in white. Numbers represent estimated execution times in TUs. This MC-DAG was obtained using the following parameters: $U_{LO} = 4$; $U_{HI} = 3$; $U_{HIinLO} = 1.5$; $p = 6$; $CP = 30$; $e = 40\%$.

[2] https://github.com/robertoxmed/ls_mxc.



(a) HI mode (b) LO mode

Fig. 7. Generated MC DAG

6 Evaluation of the Scheduling Algorithm

In this section we present our experimental results, and compare our approach to Baruah's [2] algorithm. The comparison criteria we used is the *acceptance rate*, defined as follows: given a set of MC-DAGs of tasks, supposed to be schedulable in HI and LO mode, the *acceptance rate* is the ratio of MC-DAGs for which a safe schedule (*i.e.* also ensuring safe mode transitions) was found.

6.1 DAG Generator Parameters

In our experiments, we have considered execution platforms of 2, 4, and 8 cores. The DAG parameters used for generation were chosen as follows: parallelism degree p was set twice the number of cores. Here is the rationale: for p greater than the number of cores, DAGs have mainly entry vertices. For p much smaller than the number of cores, we consider the hardware platform is oversized. The probability of having an edge between two vertices, e, was increased progressively from 20 to 60%. Utilization in HI and LO mode U_{LO}, U_{HI}, was increased progressively as well from half of the number of cores, to the number of cores (*e.g.* $U_{LO/HI}$ varied from 4 to 8 for a hardware platform of 8 cores). Utilization of HI tasks in LO mode U_{HIinLO} is given by $\frac{\min(U_{HI},U_{LO})}{2}$, that way we always reduce HI tasks' execution time in LO mode. The CP was fixed to 30 TUs for all the tests, and we considered the deadline equals to CP (see definition on Sect. 5).

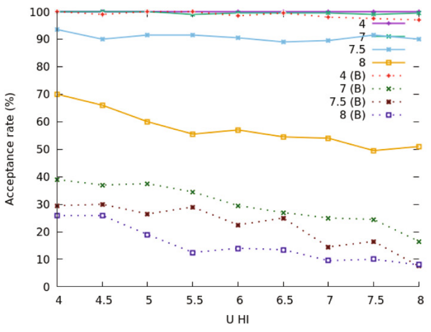
6.2 Results

Obtained results, with 8 cores, are shown in Fig. 8. We do not provide results obtained with 4 or 2 cores because they are very similar to results presented here. Each subfigure represents results obtained with an edge probability e set to 20, 40 and 60%. Each line represents the acceptance rate for a given U_{LO}, varying from 4 to 8. Continuous lines correspond to results obtained with our method, whereas dashed lines are results obtained with Baruah's MC-DAG scheduler. The x-axis represents the U_{HI}, also varying from 4 to 8. Each point of the figure gives the acceptance rate obtained on a set of 200 DAGs for each combination of parameters U_{LO}; U_{HI}; U_{HIinLO}; p; e; CP.

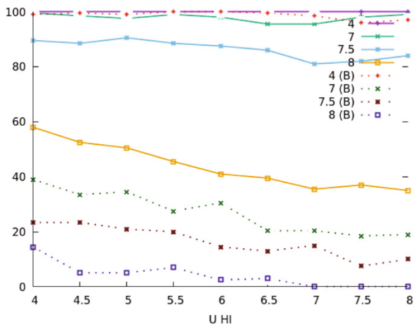
6.3 Analysis of Results

Except when $U_{LO} = 4$, Baruah's scheduler has a much lower acceptance rate than our algorithm. This was predictable: forcing LO preemptions each time a HI tasks can be executed is suboptimal. The difference becomes very significant when U_{LO} increases: on Fig. 8a, with $e = 20$ and $U_{LO} = 7$, our method has an acceptance rate very close to 100% whereas Baruah's algorithm produces an acceptance rate below 40%. Clearly, relaxing the preemption condition to only LSAIs gives a better acceptance rate.

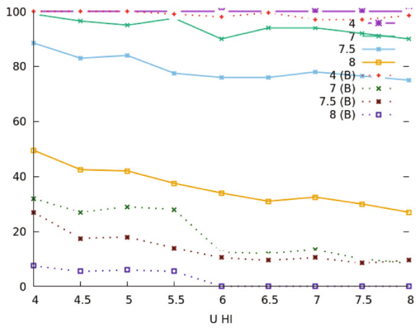
More generally, we can see on Fig. 8a that increasing U_{LO} impacts significantly the acceptance rate of a scheduling method. In practice, U_{LO} is expected to be high in a MC-DAG: increasing U_{LO} may either enable the inclusion of more functionalities, or reduce the probability of TFEs by overestimating WCET in mode LO. Experimental results show that the acceptance rate obtained with our method begins to decrease when U_{LO} is above 7. For instance, on Fig. 8a, with



(a) Edge probability of 20%



(b) Edge probability of 40%



(c) Edge probability of 60%

Fig. 8. Acceptance rates for different edge probabilities.

$e = 20$ and $U_{LO} = 7$, the acceptance rate is around 100%. With $U_{LO} = 7.5$, the acceptance rate drops to approximately 90%. It decreases progressively between 70 to 50% when $U_{LO} = 8$ and U_{HI} increases. LSAIs of HI tasks are the main reason for this behavior. Each time a HI tasks preempts a LO one, the completion time of this LO task is increased, potentially ending in a deadline miss. However, the acceptance rate provided is still very good for high levels of utilization $U_{HI} = 8$ and $U_{LO} = 7.5$ on 8 cores. Therefore a vast number of LO tasks can be included into the system and we would still be able to ensure a safe scheduling.

With higher values of e, 40 and 60%, (*i.e.* the DAGs have more edges) we have similar results except that the acceptance rate for U_{LO} above 7.5 decreases. However, it remains above 75% in all subfigures of Fig. 8.

Our scheduling algorithm is very efficient when it comes to finding schedulers for DAGs. At each step of the allocation phase, the scheduler tests if there are enough slots to schedule the remaining of the DAG, which discards non-feasible cases rapidly. On average, the scheduling phase of our experiments took 70 s for 200 DAGs that were a combination of parameters $U_{LO}; U_{HI}; U_{HIinLO}; p$; $e; CP$. However, since the complexity of HLFET is polynomial, the running time of our scheduling algorithm can increase significantly depending on the number of nodes contained in the graph.

7 Related Works

In this section, we position our approach with respect to existing contributions aiming at scheduling DAGs for Real-Time systems.

Saifullah *et al.* [14], adapted preemptive and non-preemptive Earliest Deadline First (EDF) to schedule DAGs. In this work, authors transform the DAG of tasks into a set of independent tasks scheduled with EDF by synthesizing the scheduling parameters of these tasks (*i.e.* period, deadline). However, this approach *requires idle time* between the completion of the DAG and the deadline in order to compute the tasks parameters, which leads to an underutilization of the platform. DAGs in this work do not include tasks with different criticality levels.

Scheduling tests based on Worst-Case Response Time (WCRT) for multiple DAGs using DM and EDF are presented in [13]. Necessary conditions are found for systems running multiple DAGs by finding safe upper-bounds on their WCRT. However, these safe upper-bounds can be very pessimistic if applied to a single DAG, which is the scope of our contribution. DAGs can be judged as non-schedulable when in fact a valid schedule exists. In addition, each DAG has only one criticality level in this work.

A scheduling approach for multiple DAGs with mixed-criticality levels was presented in [10]. Authors use a federated approach to allocate cores to tasks: a single DAG can have various exclusive cores for its execution, while less demanding DAGs are scheduled in processors that are left. The task model used in this paper differs from ours since they use DAGs to describe the internal structure of a task: a task is modeled by jobs with precedence constraints among them. Criticality levels being assigned to tasks, means that this model forbids dependencies among tasks of different criticality levels (even dependencies of LO tasks on HI tasks). As opposed to our proposal, mixed-criticality tasks are independent tasks in this model.

The federated approach was also considered on the latest work of Baruah [3] to schedule multiple DAGs into one multi-core architecture. DAGs with a high utilization value will have exclusive cores assigned to them and LS is applied to find the scheduling tables for these exclusive cores. Tasks with a low utilization are distributed to the remaining cores and are considered to be sequential, so any real-time scheduling algorithm can be applied for them. The model differs from the one in [10] where DAGs are assigned with a single criticality level, while Baruah's model allows vertices to have different criticality levels in the same DAG. Nonetheless the approach to schedule multiple DAGs still considers HI tasks with the highest priority over LO tasks (like in [2]), which still causes delays on LO tasks for the scheduler in LO mode, this can be avoided with our scheduling approach.

Existing contributions do not aim at finding a minimal execution time for a single DAG of tasks with mixed criticality levels. As a consequence, these contributions would perform poorly when aiming at scheduling a single-DAG with criticality levels.

8 Conclusion

This paper presents an efficient and safe scheduling algorithm for real-time systems modelled with a DAG of MC tasks. Being based on a heuristic that minimizes the completion time of the DAG, our algorithm takes advantage of multi-core platforms to find a near optimal allocation of tasks. Evaluation results provided in the paper show the capacity of our algorithm to find feasible schedules, even when the utilization of the multi-core platform is significant. Last but not least, our algorithm ensures safe mode transitions: higher criticality tasks will meet their deadline even in case timing failures occur. This paper also presents the very first benchmarking tool that generate randomly DAGs of MC tasks. As we believe this task model will become more and more popular in real-time domain, this open-source tool should be of great interest for the community.

Our future works will consider multiple DAGs being executed into a single multi-core platform with different periods and deadlines. In addition, we plan to integrate our work in design methodologies aiming at code generation [7] and safety analysis [11].

Acknowledgment. This research work has been funded by the academic and research chair Engineering of Complex Systems.

References

1. Adam, T.L., Chandy, K.M., Dickson, J.R.: A comparison of list schedules for parallel processing systems. Commun. ACM **17**(12), 685–690 (1974)
2. Baruah, S.: Implementing mixed-criticality synchronous reactive systems upon multiprocessor platforms. University of North Carolina at Chapel Hill, Technical report (2013)
3. Baruah, S.: The federated scheduling of systems of mixed-criticality sporadic dag tasks. In: 2016 IEEE Real-Time Systems Symposium (RTSS), pp. 227–236. IEEE (2016)
4. Baruah, S.K.: Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In: RTNS (2012)
5. Bini, E., Buttazzo, G.C.: Measuring the performance of schedulability tests. Real-Time Syst. **30**(1–2), 129–154 (2005)
6. Burns, A., Davis, R.: Mixed Criticality Systems - A Review. Department of Computer Science, University of York, Technical report, January 2016 (2013)
7. Cadoret, F., Robert, T., Borde, E., Pautet, L., Singhoff, F.: Deterministic implementation of periodic-delayed communications and experimentation in aadl. In: ISORC, June 2013
8. Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J.M., Wagner, F.: Random graph generation for scheduling simulations. In: Proceedings - ICST (2010)
9. Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. J. Parallel Distrib. Comput. **59**(3), 381–422 (1999)
10. Li, J., Ferry, D., Ahuja, S., Agrawal, K., Gill, C., Lu, C.: Mixed-criticality federated scheduling for parallel real-time tasks. In: RTAS (2016)
11. Medina, R., Borde, E., Pautet, L.: Availability analysis for synchronous data-flow graphs in mixed-criticality systems. In: Proceedings - SIES (2016)
12. Pagetti, C., Saussié, D., Gratia, R., Noulard, E., Siron, P.: The rosace case study: From simulink specification to multi/many-core execution. In: 2014 IEEE 20th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 309–318. IEEE (2014)
13. Parri, A., Biondi, A., Marinoni, M.: Response time analysis for G-EDF and G-DM scheduling of sporadic DAG-tasks with arbitrary deadline. In: RTNS (2015)
14. Saifullah, A., Ferry, D., Li, J., Agrawal, K., Lu, C., Gill, C.: Parallel real-time scheduling of DAGs. IEEE Trans. Parallel Distrib. Syst. **25**, 3242–3252 (2014)
15. Vestal, S.: Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: RTSS (2007)