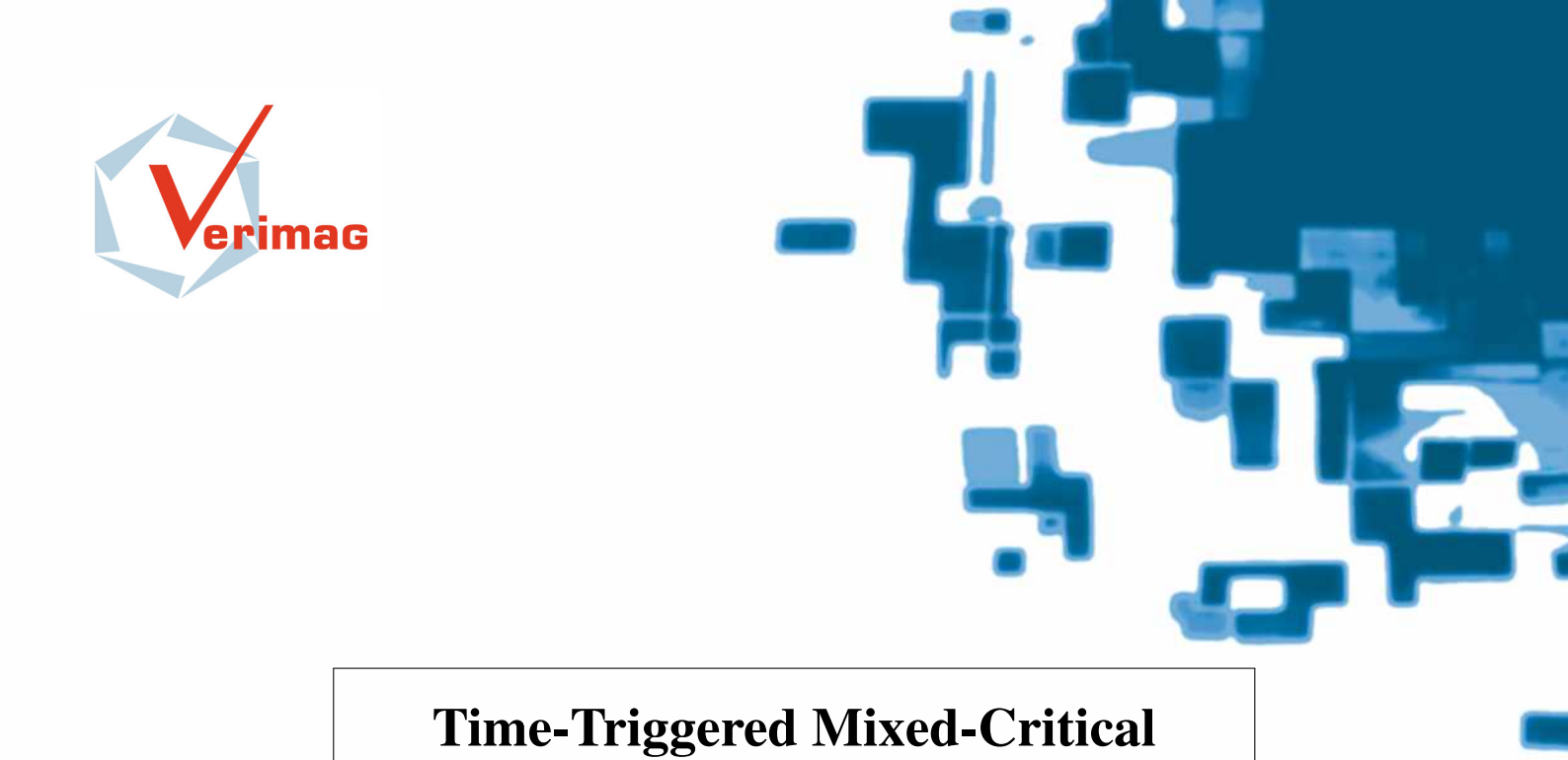


Time-Triggered Mixed-Critical Scheduler on Single- and Multi-processor Platforms (Revised Version)

Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga

Verimag Research Report n^o TR-2015-8

August 2015

Reports are downloadable at the following address
http://www-verimag.imag.fr

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

Time-Triggered Mixed-Critical Scheduler on Single- and Multi-processor Platforms (Revised Version)[1][2][3]

Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga

August 2015

Abstract

Modern safety-critical systems, such as avionics, tend to be mixed-critical, because integration of different tasks with different assurance requirements can effectively reduce their costs in terms of hardware, at the risk, however to increase the costs for certification, in particular in the context of proving their schedulability. To simplify the certification costs such systems use Time Triggered (TT) scheduling paradigm, and a generalization of the Time Triggered (TT) scheduling paradigm Single Time Table per Mode (STTM). We present a state-of-the art preemptive STTM algorithm which works optimally on single core and shows good experimental results for multi-cores. In addition, because the algorithm can be applied on top of any memoryless scheduling policy, we show that applying it to list scheduling leads to support of task graph (precedence) dependencies for which our algorithm also shows good experimental results.

Note that list scheduling also supports non-preemptive scheduling and the latter is very important for certain multi-core platforms. However, for applying our STTM approach for non-preemptive platform case we now do not have a completely correct algorithm, our previous tentative in that direction was recently discovered to still require some support of preemption. The latter discovery has lead to a revised version of this report, where we retract the non-preemption part of the results.

Keywords: time-triggered architecture, mixed criticality, multi-core scheduling, safety critical, hard real-time, precedence constraints, list scheduling, synchronous tasks

Reviewers:

How to cite this report:

```
@techreport {TR-2015-8,
  title = {Time-Triggered Mixed-Critical Scheduler on Single- and Multi-processor Platforms
(Revised Version)},
  author = {Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga},
  institution = {{Verimag} Research Report},
  number = {TR-2015-8},
  year = {2015}
}
```

[1] Revised in Janunary 2016: retracted the claim for the support of non-preemption

[2] This report is an extended version of [1]

[3] The research leading to these results has received funding from **MoSaTT-CMP** – European Space Agency project, Contract No. 4000111814/14/NL/MH, and from **CERTAINTY** – European Community's Seventh Framework Programme [FP7/2007-2013], grant agreement no. 288175.

1 Introduction

Advances in technology lead towards an increasing trend in integration of multiple functionalities on a single chip. Integration is an effective way of reducing cost and power consumption of embedded systems. This leads to the growing importance of *multi-core/multiprocessor* platforms in all areas of computer system design. In addition for safety-critical domains, such as avionics, this is leading to the integration of tasks with significantly *different* safety requirements on a single assembly of processing resources.

Such system have called into existence a special Mixed-Critical System (MCS) scheduling theory, that has been developed at least since 2007 [2]. This theory treats the different safety requirements by adequate scheduling methods, which leads to much more efficient resource usage compared to classical scheduling approaches [3].

Following a significant volume of previous MC scheduling work (*e.g.,* [3, 4]) in this paper we consider the basic problem of scheduling a *finite set of jobs* whose exact arrival times are known *a priori*. Such a problem formulation can result from considering a hyperperiod of a synchronous periodic task system. As pointed out in [4], this assumption is natural for a broad class of safety critical systems. We also restrict ourselves to dual-critical problems, which are also of practical interest, *e.g.,* for *Unmanned aerial vehicles* (UAVs), which assume two criticality levels: *safety critical* and *mission critical*.

The Own-Criticality Based Priority (OCBP) [3] is theoretically the best among all *Fixed Priority per job* (FP) scheduling algorithms for MCS, however, it works only for single-processor systems. Recent extensions of the fixed job priority policy, *e.g.,* [5], perform a switch between different priority tables for different modes. This Fixed Priority per Mode per job (FPM) policy can lead to better results due to their higher flexibility. In particular, Mixed Criticality Priority Improvement MCPI [6] is an FPM policy showing state-of-the art schedulability and supporting multiple processors.

In contrast to priority-based algorithms discussed above, the Time Triggered (TT) ones statically fix for all jobs their start times and the time intervals where they may execute. This class of schedulers is important because they considerably reduce the uncertainty of job execution intervals thus simplifying the safety-critical system certification (where simplicity is a decisive factor). They also simplify any auxiliary timing-based analysis that may be required to validate important extra-functional properties in embedded systems, such as interference on shared buses and caches, peak power dissipation, electromagnetic interference *etc.*.

Therefore, in [4, 7] S. Baruah *et al.* proposed the idea of *transformation* of non-TT based solutions into TT-ones, demonstrating this idea on OCBP for a single processor. To avoid the highly inefficient static reservation of resources, they proposed mode switching between a different TT tables, one per criticality mode. We call this scheduling approach Single Time Table per Mode (STTM). In [8] the STTM scheduling was extended from single to multiple processors, though being restricted to the systems where all jobs have the same deadline.

This work focuses on STTM algorithms. Our contribution is a novel algorithm that transforms practically *any scheduling algorithm* into an STTM one. This way one can, for example, profit from the state-of-the art FPM algorithms, such as MCPI, which, in general, perform significantly better than OCBP and have no restrictive assumptions on the deadlines and the number of processors.

An important theoretical result for our algorithm is that it can always perform a successful schedule transformation into STTM schedule for single processor dual-critical problem instances. Though this result does not extend directly to multiple processors in theory, our extensive experiments on synthetic multiprocessor benchmarks show a high probability of success even for high-utilisation problem instances.

2 Background

Consider a set of hard real-time jobs having different *levels of criticality*. It is common in literature to model different criticality requirements by giving different worst-case execution times (WCETs) for the same job. In *dual-criticality* systems we have the highly-critical level, denoted as 'HI', and the low-critical (normal) level, denoted as 'LO'. Every job gets a pair of WCET values: the LO WCET and the HI WCET. One important remark is that both HI and LO jobs are hard real-time, so both *must* terminate their executions before the deadlines. But only HI jobs undergo certification. This means that the designer is confident that the jobs will never exceed their LO WCET, calculated by exhaustive measurements and

adding some practical margin. However, for certification it is required to prove that the HI jobs will meet the deadlines even under the unlikely event that some jobs would execute at their HI WCET, calculated by safer, though more pessimistic, formal WCET analysis tools, required for certification. At the exceptionally high execution times, exceeding LO the low-critical jobs may be disabled, whereby the system will be able to compensate this by going into a safe emergency state.

2.1 Problem Definition

The scheduling problem instance consists of the number m of identical processors and a finite set of jobs. In a dual-criticality Mixed-Critical System (*MCS*), a *job* J_j is characterized by a 5-tuple $J_j = (j, A_j, D_j, \chi_j, C_j)$, where:

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

We assume that $C_j(\text{LO}) \leq C_j(\text{HI})$[3]. We also assume that the LO jobs are forced to terminate after $C_j(\text{LO})$ time units of execution, so $(\chi_j = \text{LO}) \Rightarrow C_j(\text{LO}) = C_j(\text{HI})$.

A *scenario* of an instance **J** is a vector of execution times of all jobs: $(c_1, c_2, \ldots, c_k)$, where k is the total number of jobs. If at least one c_j exceeds $C_j(\text{HI})$, the scenario is called *erroneous*. The *criticality of scenario* $(c_1, c_2, \ldots, c_k)$ is the least critical χ such that $c_j \leq C_j(\chi), \ \forall j \in [1, k]$. A scenario is *basic* if for each $j = 1, \ldots, k$ either $c_j = C_j(\text{LO})$ or $c_j = C_j(\text{HI})$. The basic scenario where for each $j = 1, \ldots, k$ $c_j = C_j(\text{LO})$, is called the 'LO' scenario.

A (preemptive) schedule S is a mapping: $S : \mathbb{R}^{\geq 0} \to \mathbb{N}^m$. S should split the time axis into a finite set of non-empty intervals inside which S takes a constant vector values. $S_p(t) = j$, $j > 0$ indicating that job J_j running on processor p and $S_p(t) = 0$ indicating idle processor. Every job should start at time A_j or later. If schedule is generated for a particular scenario c, then each job should run for no more than c_j time units. A job may be assigned to only one processor at time t, but we assume that job *migration* is possible to any processor at any time.

Based on the scheduling problem instance and the current online state, a *scheduling policy* (a deterministic online algorithm) decides which *ready jobs*, *i.e.,* those arrived and not terminated so far, are scheduled at every time instant on m processors and for how long they will continue running unless an arrival or termination job is signalled by the environment, requiring a new decision from the policy. Note that we assume that for each ready job its termination is signalled immediately when it stops running. The online state of a (memoryless) policy at every time instance consists of the set of *ready jobs* (*i.e.,* the jobs arrived and not yet terminated), the current progress of each job (*i.e.,* for how much a job has executed so far), and the current criticality mode, χ_{mode}, initialized as $\chi_{mode} = \text{LO}$ and *the mode is switched* to 'HI' as soon as a job reaches $C_j(\text{LO})$ without signalling its termination.

The ultimate goal of a scheduling policy is to ensure that the schedule is feasible. A schedule is *feasible* if the following conditions are met:

Condition 1. *If all jobs run at most for their LO WCET, then both HI and LO jobs must meet their deadlines.*

Condition 2. *If at least one HI job exceeds its LO WCET execution time, than all HI jobs must meet their deadlines, whereas LO jobs may be even dropped.*

A scheduling policy is *correct* for the given instance **J** if for each non-erroneous scenario it generates a feasible schedule. One also says that in this case the given instance **J** is *schedulable by the given policy*. If no policy is specified, an instance *schedulable* instance is an instance for which a correct scheduling policy exists.

A scheduling policy is *work-conserving* if it never lets any ready job to wait (*i.e.,* to not execute on any processor) if there is an idle processor available. Because we assume preemptive scheduling with zero preemption cost and allow job migration, in our model having a non work-conserving policy cannot improve schedulability of a policy.

Scheduling policies are split into classes (sets of policies), *e.g.,* fixed priority (FP) policies, single time-triggered table (STTM) policies, *etc.*. Different policies in a given class differ by certain class-specific parameters, *e.g.,* priority assignment in the case of FP policies. A scheduling policy is called *optimal* in a class of policies if for any instance **J** holds that if that instance is schedulable by a certain policy in the given class then it is also schedulable by the specified policy.

2.2 Correctness and Predictability

To verify the correctness of a scheduling policy one usually tests it for the maximal possible execution times, which in our case corresponds to HI WCET. However, to justify this test a scheduling policy must be *predictable*, which means that reducing execution time of any job 'A' (while keeping all other execution times the same) may not delay the termination of any other job 'B'. In other words, predictability means that the termination times have *monotonic* dependency on execution times.

For mixed-critical scheduling the predictability requirement is too restrictive, as it does not take into account that increase of an execution time of a HI job to a level that exceeds its LO WCET may lead to a mode switch and hence to dropping the LO jobs, which, in turn may lead to an earlier termination of another HI job, and hence non-monotonic dependency of termination times. Therefore, a weaker property is adopted in this case, which we call *predictable per mode*. This property poses almost the same requirement of non-increasing termination time of a job 'B', but now it is not required anymore to hold under *arbitrary* execution time reduction of a job 'A'. Now it is required only if the reduction does not lead to the change of the mode in which job 'B' terminates, *e.g.,* when the reduction keeps the execution time higher than LO WCET.

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

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

Definition 2.1 (Job-specific Basic Scenario)**.** *For a given problem instance, LO basic-scenario schedule S^{LO} and HI job J_h, the basic scenario defined above is called 'specific' for job J_h and is denoted HI-J_h, whereas its schedule is denoted $S^{HI\text{-}J_h}$.*

Note that $S^{HI\text{-}J_h}$ coincides with S^{LO} up to the time when job J_h switches, and after the switching time it starts using HI execution times for the jobs that did not terminate before the switch. Figure 1 shows Gantt charts for the basic scenarios of a certain single-processor problem instance. We see, for example that in the LO scenario job J_2 terminates at time 9, but in the HI-$J2$ scenario job J_2 switches at time 9 and continues to execute, because, apparently, it has a HI WCET larger than the LO WCET.

Theorem 2.2 (Correctness Verification by Job-specific Scenarios)**.** *To ensure correctness of policy that is predictable per mode it is enough to test it for the LO scenario and the scenarios HI-J_h of all HI jobs J_h.*

This theorem can be derived from the properties of the correctness verification algorithm presented in [3].

2.3 Priority-based Scheduling

A *fixed-priority* scheduling policy is a policy that can be defined by a priority table PT, which is a vector specifying all jobs in a certain order. The position of a job in PT is its *priority*, the earlier a job is to occur

in PT the higher the priority it has. Among all ready jobs, the fixed-priority scheduling policy always selects the m highest-priority jobs in PT. A priority table PT defines a total ordering relationship between the jobs. If job J_1 has higher priority than job J_2 in table PT, we write $J_1 \succ_{PT} J_2$ or simply $J_1 \succ J_2$, if PT is clear from the context. In this paper we assume *global* fixed-priority scheduling which allows unrestricted job migration.

A *fixed priority per mode* (FPM) policy uses two tables: PT_{LO} and PT_{HI}. The former includes all jobs. The latter includes only HI jobs. As long as the current mode is LO, this policy performs the fixed priority scheduling according to PT_{LO}. After a switch to the HI mode, this policy drops all pending LO jobs and applies priority table PT_{HI}.

EDF (earliest deadline first) [9] is a fixed-priority algorithm with a priority table PT that gives jobs with smaller deadlines higher priority: $D_i < D_j \Rightarrow J_i \succ_{PT} J_j$. *EDF is an optimal scheduling policy for ordinary (non-mixed critical) single-processor problems*. Given this, and observing that scheduling after the mode switch is an ordinary scheduling problem, we identify the following important class of single-processor policies.

Definition 2.3 ('Reasonable' Single-processor Policies)**.** *A single-processor dual-critical scheduling policy is called 'reasonable' if after the mode switch it applies EDF for the HI jobs and either drops the LO jobs altogether or gives them less priority than that of any HI job.*

For single-processor problems, any policy can be translated into a reasonable policy letting it behave according to the definition above whenever a mode switch occurs. All the instances schedulable in the original policy would remain so in the translated policy.

Consider a *reasonable FPM policy*. In this case PT_{HI} is known *a priori*, as it is an EDF table. Therefore only the calculation of PT_{LO} requires an optimization algorithm to achieve good schedulability. In contrast to the single-processor case, in the multi-processor case EDF is not optimal, so there is no straightforward way to extend the 'reasonable policy' definition to this case.

For computing PT_{LO} and PT_{HI} table for multiprocessor platforms, in [6], we proposed Mixed Criticality Priority Improvement (MCPI) algorithm. We use this algorithm as an integral part (the basis scheduler) for the STTM algorithm proposed here, although any other (memoryless) scheduling policy can be used for this role.

Fixed-priority (FP) policy is predictable, both in the single- and the multi-processor cases [10]. Intuitively, this implies that *FPM policy is predictable per mode*, and hence, by Theorem 2.2, *the schedulability of FPM can be verified by simulation in the LO and all the HI-J_h scenarios*.

Example 2.1. *Let us consider the following instance on single processor as an example:*

Job	*A*	*D*	χ	*C(LO)*	*C(HI)*
1	*0*	*12*	*HI*	*3*	*5*
2	*6*	*11*	*HI*	*2*	*4*
3	*7*	*8*	*LO*	*1*	*1*
4	*1*	*4*	*HI*	*1*	*2*

and assume the following FPM priority assignment (which can be computed using MCPI [6]):

$$PT_{LO} = \quad J_3 \succ J_2 \succ J_4 \succ J_1$$
$$PT_{HI} = \quad J_2 \succ J_4 \succ J_1$$

For this example, Fig. 1 shows the schedules obtained by simulation of the FPM policy in the basic LO scenario and all job-specific HI scenarios. It can be concluded that in the LO scenario all the jobs meet their deadlines and in the HI scenarios all the HI jobs meet their deadlines. Therefore, by Theorem 2.2, this instance is schedulable under the FPM policy with the given priority tables.

2.4 Time Triggered Scheduling

An ordinary (*i.e.,* non mixed-criticality aware) *Time Triggered* (TT) policy defines a static, pre-computed table that defines at every instant of time which job must be scheduled at each processor provided that it did not terminate yet and assuming that the job may require up to its WCET time units.

For MCS [4] introduced the *Single-Time Table per Mode* (STTM) policy, which defines one table per criticality mode. In a dual-critical system we call **LO** and **HI*** the tables for the LO and HI mode, respectively. The corresponding static schedules are denoted as S^{LO} and S^{HI*}. The two STTM tables are correct iff:

1. They schedule all jobs after their arrival and before their deadline, allocating each job $C_j(\text{LO})$ time units in **LO** table and each HI job $C_j(\text{HI})$ time units in **HI*** table.
2. If at any time we switch from **LO** to **HI***, then all not-yet-terminated HI jobs will have enough time to continue their execution so as to reach $C_j(\text{HI})$ time units.

For the previous example, Fig. 1 shows in the last row the **HI*** table which, together with the **LO** table in the first row, it satisfies the correctness requirements given above. For a given scheduling policy, **HI*** table is derived from the **LO** one by simulating a variant of the same scheduling policy transformed by the algorithm presented in the next section.

3 Transformation Rules

Our algorithm, denoted $\mathcal{T}(ALG)$ transforms a given 'basis' scheduling policy ALG into an STTM policy by augmenting it with additional rules. In practice, we use an FPM policy, in particular, MCPI, as the basis policy.

The table **LO** is simple to obtain, we just simulate ALG for the LO basic scenario and use the generated schedule S^{LO}. Our algorithm mainly consists of the method to obtain a correct table **HI***. We propose a method to generate this table by simulating ALG in the HI mode (*i.e.,* initializing $\chi_{mode} = \text{HI}$) for HI jobs with $C(\text{HI})$ times and assuming that a HI job can be *disabled* at any time when all three *enabling rules* defined in this section are false. These rules are based on the **LO** table and their purpose to ensure that any switch from table **LO** to the HI mode in table **HI*** will be correct.

Note that a *memoryless* basis policy by definition relies only on the set of current ready jobs to construct the schedules. For this reason, such a policy is always able to produce consistent results even if this set is changed by the environment by job enabling/disabling. In particular, when ALG is an FPM policy, this means executing fixed priority policy with priority table PT_{HI}, whereby whenever a ready HI job is (temporarily) disabled, a lower-priority HI job can execute instead, which means that a *priority inversion* may occur, but the schedule will still be correct. Enabling/disabling the jobs is exactly what is done by extra rules added into the basis policy by the transformation algorithm.

Let us give some supplementary definitions. Let $T_j^{LO}(t)$ (resp. $T_j^{HI}(t)$) be the cumulative execution progress of job J_j by time t in table **LO** (resp. **HI***). We call a HI job that has executed for more than its $C(\text{LO})$ a *switched job*. It is *non-switched* otherwise. We say that a job switches at time t when $T_j^{LO}(t)$ reaches $C_j(LO)$. A job J_j is *enabled* at time t if:

- the job has arrived: $t > A_j$
- the job has not yet terminated: $T_j^{HI}(t) < C_j(\text{HI})$
- at least one of the following *rules* is true:

$$T_j^{LO}(t) = C_j(\text{LO}) \tag{1a}$$

$$T_j^{HI}(t) < T_j^{LO}(t) \tag{1b}$$

$$T_j^{HI}(t) = T_j^{LO}(t) \wedge \exists p \,.\, S_p^{LO}(t) = j \tag{1c}$$

Informally, Rule (1a) permanently enables all switched jobs, while Rules (1b) and (1c) assure that a job will not run in **HI*** for more time than in **LO** before the switch.

Example 3.1. *Consider again the problem instance of Example 2.1, whose schedules in the basic scenarios is shown in Figure 1. The **LO** table coincides with the schedule in the LO scenario, and table **HI***, also shown in the figure, is obtained by the transformation algorithm based on table **LO**. The algorithm performs the simulation of HI jobs, explained below in detail.*

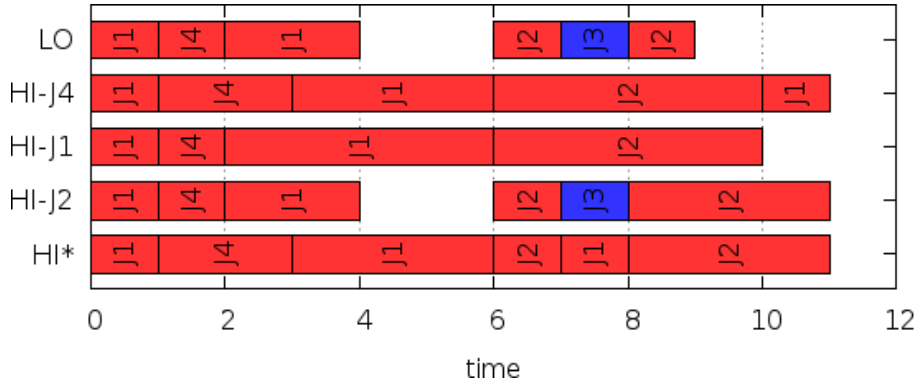


Figure 1: Basic scenarios and TT tables

At time 0, only J_1 has arrived, and it is enabled by Rule (1c). At time 1, J_4 arrives, it has higher priority then J_1 and it is enabled by Rule (1c), so it is chosen by the algorithm to be executed. At time 2 for job J_4 Rule (1c) will be false, but Rule (1a) will become true, so we will continue execute it until time 3. At time 3 J_4 will terminate, so J_1 will be enabled by Rule (1b) until time 5 and by Rule (1a) from 5 on. So J_1 will continue its execution till time 6, when J_2 arrives. J_2 is enabled by Rule (1c), and it has higher priority than J_1, so it will be executed until time 7. At this instant Rule (1c) becomes false for J_2, disabling it. So we execute J_1. At time 8 J_1 terminates and J_2 is enabled by Rule (1c). At time 9 Rule (1c) is false for J_2, while Rule (1a) becomes true. So J_2 continues its execution until time 11, when it terminates.

It is easy to verify the correctness of TT scheduling that uses **LO** and **HI*** as tables. In fact in table **LO** all the jobs meet the deadline. When there is a switch, at time t, from **LO** to **HI***, all HI job J_j must have from time t a quantity of time reserved for them in **HI*** equal to $C_j(HI) - T_j^{LO}(t)$. In our example, if there is a switch in the **LO** table at time 2, caused by job J_4, then J_1, J_4 and J_2 will have enough remaining time reserved in **HI*** (respectively $4 = C_1(\text{HI}) - T_1^{LO}(2) = 5 - 1$, $1 = C_4(\text{HI}) - 1$ and $4 = C_2(\text{HI}) - 0$), and will terminate before their deadlines. In this case we will drop job J_3, since we do not care about LO jobs when in HI mode. Similarly, in the case of a switch at time 4, caused by J_1, then J_1 and J_2 will have respectively $3 = C_1(\text{HI}) - 2$ and $4 = C_2(\text{HI}) - 0$. Note that in this case J_1 will have one time unit more then it actually needs. Finally, if there will be a switch at time 9, caused by job J_2, this job will have 2 other time units, terminating at time 11, meeting its deadline.

We have the following result, which shows that the first requirement of STTM correctness is always satisfied by our transformation rules.

Lemma 3.1. *If at any time we switch from **LO** to **HI***, then all the unterminated jobs will have enough time reserved in **HI*** to terminate their work.*

First, let us comment that, according to our rules to construct **HI***, no HI jobs get disabled forever because eventually Rule (1a) becomes true, since all LO jobs eventually terminate. Thus, all HI jobs get a total time $C(\text{HI})$ reserved in **HI***. Consequently, if a job switches at time t, then this and any other job is guaranteed to get $C(\text{HI}) - T_j^{HI*}(t)$, but needs to get at least $C(\text{HI}) - T_j^{LO}(t)$.

Therefore the lemma can be equivalently stated as follows:

*no non-switched HI job makes more progress in **HI*** than in **LO**.*

Formally:

$$\forall t\ ,\ T_j^{LO}(t) < C_j(\text{LO}) \Rightarrow T_j^{LO}(t) \geq T_j^{HI*}(t)$$

Proof of Lemma 3.1. At time $t = 0$ the lemma thesis is obviously true, and with progress of time it can be invalidated only during the time when a job is scheduled in **HI***. However, as long as $T_j^{LO}(t) < C_j(\text{LO})$ job J_j can only be scheduled when either (1b) or (1c) is true, but they both imply that we have $T_j^{LO}(t) \geq T_j^{HI*}(t)$. □

Also, for single processor, the transformation algorithm is optimal:

Theorem 3.2 (Transformation Correctness)**.** *For a given single-processor problem instance if the basis policy ALG is correct and reasonable then the policy $\mathcal{T}(ALG)$ is also correct.*

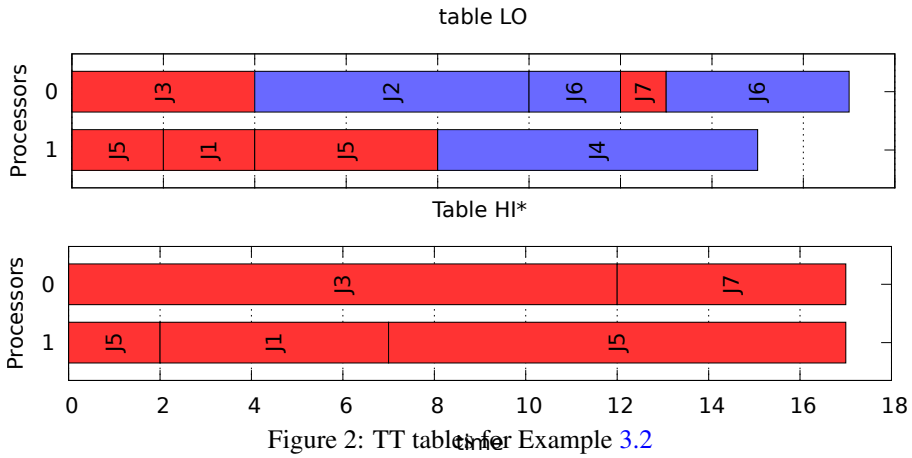


Figure 2: TT tables for Example 3.2

For proof, see Appendix A. There we also give a proof to another important property for single-processor scheduling, which states that also the reverse result is true in some quite general case.

Theorem 3.3 (Reverse Correctness)**.** *For a given single-processor problem instance, under the assumption that the basis policy ALG is reasonable, we have that if the policy $\mathcal{T}(ALG)$ is correct then policy ALG is correct as well.*

Corollary 3.4 (Testing Correctness based on two Tables)**.** *For single-processor problem instances and reasonable work-conserving policy ALG a necessary and sufficient correctness test is testing that both scheduling tables, **LO** and **HI***, obtained from $\mathcal{T}(ALG)$ meet their deadlines.*

Note that testing over only two tables may show a potential computational improvement over the correctness test proposed in Theorem 2.2, which tests the scheduling policy over $H + 1$ 'tables', where H is the number of HI jobs in the problem instance.

Unfortunately these correctness results do not extend to multiple processors. Nevertheless, from our experiments – see Section 5 – we observe that the 'direct' correctness result holds *almost always* for FPM policies. Only for very high-load problem instances there are rare cases that FPM meets the deadlines whereas the transformation algorithm does not. Below we give an example of a successful transformation for two processors:

Example 3.2. *Consider the following instance:*

Job	*A*	*D*	χ	$C(LO)$	$C(HI)$
1	*2*	*9*	*HI*	*2*	*5*
2	*0*	*10*	*LO*	*6*	*6*
3	*0*	*16*	*HI*	*4*	*12*
4	*4*	*17*	*LO*	*7*	*7*
5	*0*	*18*	*HI*	*6*	*12*
6	*8*	*19*	*LO*	*6*	*6*
7	*12*	*20*	*HI*	*1*	*5*

and the following FPM priority assignment:

$$PT_{LO} = J_1 \succ J_3 \succ J_5 \succ J_7 \succ J_2 \succ J_5 \succ J_6$$
$$PT_{HI} = J_1 \succ J_3 \succ J_5 \succ J_7$$

Fig. 2 presents the tables **LO** and **HI*** obtained from $\mathcal{T}(FPM)$ for this instance on two processors, using similar reasoning as in the previous example.

4 List Scheduling Transformation

4.1 Extending the Scope for Applying Transformations

In the previous section we formulated correctness theorems of the transformation algorithm for the case with the following assumptions: (1) only a single processor is available; (2) there are, implicitly, no task-graph dependencies between the jobs; (3) EDF scheduling policy is applied in the HI mode; (4) preemption support is provided by the platform. Nevertheless, our algorithm is applicable in practice also when the restrictions (1-3) are alleviated, though one has to check the correctness of the result even when the basis policy is correct.

As for restriction (4), in the revised version of the report we insist in keeping this restriction for the transformation into STTM. This is because we recently discovered that our previous tentative to avoid this restriction in STTM was not successful as preemption may still be required at mode switch. Therefore, for now we keep transformation of non-preemptive algorithms for future work. The non-preemption results that we still keep for legacy reasons do not concern the transformation.

In this section we describe a basis policy that works in practice beyond the restrictions (1-3). Thus, we abandon the usual assumption of previous works on mixed-critical translation of event-triggered to time-triggered table [4, 7, 8] which study only the cases where correctness of transformation can be proved by construction. Instead, we test the correctness after the translation. Our experiments in Section 5 show that even for hard scheduling problem instances the proposed approach results in a correct transformation in a grand majority of problem instances.

A classical scheduling policy that supports task graph dependencies and (optional) non-preemption is so-called *list scheduling*, denoted LS-SC. List scheduling policy is more general than fixed priority and EDF, as it admits a priority table PT and behaves in exactly the same way as fixed priority policy if the set of task-graph dependencies is empty. Similar to fixed priority, this policy is also memoryless, which makes it another valid input to our transformation algorithm. In this section we will describe the implementation of LS-SC and $\mathcal{T}$(LS-SC), the transformed list scheduling for generating the **HI*** table.

4.2 Extended Problem Formulation

Formally, a task graph $\mathbf{T}$ as an MC-scheduling problem instance is a directed acyclic graph $(\mathbf{J}, \rightarrow)$ whose set of nodes $\mathbf{J}$ consists of k jobs defined in Section 2.1 and whose set of arcs defines execution-order *dependency relation* $\rightarrow \subset \mathbf{J} \times \mathbf{J}$, also called *precedence relation*.

If $J_1 \rightarrow J_2$ then J_2 may not start until J_1 signals its termination. Formally, we say that J_2 is not *ready* until J_1 terminates. In fact, we modify the definition of job J_j readiness at time t by not only requiring that the job has arrived by time t ($A_j \leq t$) but also that each task-graph predecessor has terminated by time t. This extra requirement is referred to as *precedence constraint* or *task-graph dependency*. When both the job J_j and its predecessor J_i are HI jobs then $J_i \rightarrow J_j$ a referred to as a 'HI precedence (dependency)', when, on the contrary, at least one of these jobs is a LO job then we talk of a 'LO precedence (dependency)'.

Next to the conditions defined in Section 2, a feasible schedule should satisfy two additional conditions:

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

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

4.3 List Scheduling

For a given task graph $\mathbf{J}$, a number of processors m, and a priority table PT the list scheduling consists of simulating the fixed-priority (FP) policy at a single mode of criticality χ', being either LO or HI[4], while taking into account that a job can become ready only after the termination of its predecessors that should be respected at a given level of criticality. The pseudo-code of the classical list scheduling algorithm adapted for this purpose is given in Figure 3. For the set jobs we specify an array of arrival times $A[*]$, deadlines

[4] The algorithm can also be adapted for simulating the FPM policy, in a given mode switching scenario.

$D[*]$, and WCET times $C[*][LO..HI]$. We use the '$[*]$' to explicitly denote an array dimension of some range that can be deduced from the context. In the particular case considered now, the range is the set of jobs, whereas in the second dimension of the array C specifies the WCET criticality level.

The output of the algorithm is a schedule S, which is defined by two arrays: $S.start$ and $S.end$. Position $S.start[J]$ (resp. $S.end[J]$) specifies for job J the list of start (end) times of all timing intervals in which job J executes. The algorithm keeps track of job progress $prgs$, an array that for each job specifies for how long it has executed so far. Set $\mathbf{J}^{arr}$ specifies the jobs that have arrived so far. Two arrays, processor status $pstat$ and job status $jstat$, specify for each processor a job that runs there and vice versa, for each job the processor where the job runs. Note that for efficiency, also for the priority table PT, which for each priority specifies the corresponding job, we need a reverse array, that for each job specifies its priority, we denote this array PT^{-1}.

The pseudo-code in Figure 4 gives the two basic operations that the algorithm uses to start and finish an interval of job execution, manipulating the schedule, the job status, and the processor status. The $SchedStop$ operation contains a check to avoid empty schedule intervals.

The algorithm keeps two priority-queue data structures: Q_P, Q_E. A priority queue [11] is a collection that remembers the elements with their 'keys' and supports such operations as providing the highest-key element (called the '*front*' of the queue), eliminating the front element ('*pop*'), and adding a new element with a key ('*push*'). We use queue Q_P to keep the ready not (yet) running jobs at the order of decreasing priority, highest-priority job being at the front of the queue. Priority queue Q_E is used to keep the simulated schedule events at their time-stamp order, the earliest event being at the front. An event is identified by pair $[J, LBL]$ where LBL is a label indicating the event type, such as 'arrival' (LBL-ARR), 'getting ready' (LBL-READY), and 'termination' (LBL-TERM).

Further, the algorithm keeps an array that for job J gives the count $termPredecessors[J]$ of predecessors of job J that have terminated. Finally, the algorithm keeps some simple variables, not explicitly listed in the header, such as the time-stamps of current and last event.

When the algorithm starts, we first filter the set of jobs and dependencies from those that have criticality level smaller than χ', as they do not need to be taken into account in the mode χ'. In dual-critical scheduling this means filtering away the LO jobs and LO dependencies if the requested mode has criticality HI. Then we add the arrival events into priority queue Q_E for the remaining 'effective' set of jobs.

The main loop runs until the event queue is empty. The earliest event $[J, LBL]$ is first popped from the queue together with its time-stamp. The progress $prgs$ of all running jobs (*i.e.,* all non-empty entries $pstat[1..m].job$) is incremented by the delay since the last event $(lastTime - time)$. Then the algorithm handles different types of events (see the switch-case operators).

Observation 4.1 (Ordering of List-schedule Events)**.** *We implicitly assume that events with same time-stamp and different type are popped with the preference to the event types that are first mentioned in the switch case of the algorithm, in particular that events labeled as 'LBL-TERM' are always popped first if there are any for the current time stamp. This is needed to prevent that a job may terminate and get preempted at the same time.*

When a job terminates the processor is freed by $SchedStop$ operation and for all successors the counters $termPredecessors$ are incremented. If the current job is the last predecessor to terminate for some successors and the successors have already arrived then their 'getting ready' events are enqueued for processing. Note that we do not put the ready successors directly into the ready-job queue Q_P, because the **basic convention** of our list-schedule implementation is that *per iteration of the main while loop at most one job may change its 'readiness' status* either from ready to non-ready or vice versa. Therefore, making the successors ready is postponed to the future iterations.

When a job arrives it is registered in the set of arrived jobs. If by that time all the predecessors have terminated then the job is directly enqueued into the ready-job queue Q_P, without unnecessary passing through the event queue with an LBL-READY event, as we are allowed to change the status of a single job. Note that another important invariant of our algorithm is that a ready job is either waiting in the ready-job queue Q_P or is registered as a 'running' job through the $jstat$ and $pstat$ data structures. A ready job can move between the 'waiting' and 'running' states a few times if the preemption is allowed. Finally, upon its termination the job goes from 'running' to 'terminated' state, which is ensured in the pseudo-code by a $SchedStop$ operation that is not followed by a push to Q_P.

Algorithm: $SimulateListSchedule$
Input: **Boolean** $preemAllowed$ **integer** m **define** $k = size(\mathbf{J})$
Input: **criticality** χ' **task graph** $\mathbf{T}(\ \mathbf{J}(A[*], D[*], \chi[*], C[*][LO..HI]), \rightarrow_{LO..HI})$ **priority table** PT
Output: **schedule** S
Local: **array** $[1..k]$ **of time-type** $prgs$ **set of jobs** $\mathbf{J}^{arr}$ **set of jobs** $\mathbf{J}_{\text{EFF}}$ **dependencies** $\rightarrow_{\text{EFF}}$
Local: **array** $[1..m]$ **of processor status** $pstat$ **array** $[1..k]$ **of job status** $jstat$
Local: **priority queue** Q_E, Q_P **array** $[1..k]$ **of integer** termPredecessors

1: $\mathbf{J}_{\text{EFF}} \leftarrow \{\ J \in \mathbf{J} \mid \chi[J] \geq \chi'\ \}$
2: $\rightarrow_{\text{EFF}} \leftarrow \{\rightarrow_{\chi''} \mid \chi'' \geq \chi'\ \}$
3: $PQueuePushSet(\ Q_E,\ [\ \mathbf{J}_{\text{EFF}},\ \text{'LBL-ARR'}\ ],\ A[*]\ )$
4: $lastTime \leftarrow 0$
5: **while** $Q_E \neq \emptyset$ **do**
6: $(\ [J, \text{LBL}],\ time\ ) \leftarrow PQueuePop(Q_E)$
7: $UpdateProgress(\ lastTime,\ time,\ prgs,\ pstat\ )$
8: **switch** LBL **do**
9: **case** 'LBL-TERM'
10: $SchedStop(\ J,\ time,\ S,\ jstat,\ pstat\ )$
11: **for** $J' \in Successors(J, \rightarrow_{\text{EFF}})$ **do**
12: $termPredecessors[J'] \leftarrow termPredecessors[J'] + 1$
13: **if** $termPredecessors[J'] = PredecessorCount(J', \rightarrow_{\text{EFF}})\ \wedge\ J' \in \mathbf{J}^{arr}$ **then**
14: $PQueuePush(\ Q_E,\ [\ J',\ \text{'LBL-READY'}\ ],\ time\ )$
15: **end if**
16: **end for**
17: **case** 'LBL-ARR'
18: $SetAdd(\mathbf{J}^{arr}, J)$
19: **if** $termPredecessors[J] = PredecessorCount(J, \rightarrow_{\text{EFF}})$ **then**
20: $PQueuePush(\ Q_P,\ J,\ PT^{-1}[J]\ )$
21: **end if**
22: **case** 'LBL-READY'
23: $PQueuePush(\ Q_P,\ J,\ PT^{-1}[J]\ )$
24: **if** $Q_P \neq \emptyset$ **then**
25: $J \leftarrow PQueueFront(Q_P).value$
26: **if** $AllProcessorsBusy(pstat)$ **then**
27: $J' \leftarrow LeastPrioPreemptableJob(\ PT,\ pstat,\ preemAllowed,\ prgs\ )$
28: **if** $J' \neq \emptyset\ \wedge\ J \succ_{PT} J'$ **then**
29: $proc' \leftarrow jobstat[J'].proc$
30: $SchedStop(\ J',\ time,\ S,\ jstat,\ pstat\ )$
31: $PQueuePush(\ Q_P,\ J',\ PT^{-1}[J']\ )$
32: $SchedRun(\ J,\ time,\ proc',\ S,\ jstat,\ pstat\ )$
33: $PQueuePop(\ Q_P)$
34: **end if**
35: **else**
36: $proc \leftarrow GetAvailableProcessor(pstat)$
37: $SchedRun(\ J,\ time,\ proc,\ S,\ jstat,\ pstat\ )$
38: $PQueuePop(\ Q_P)$
39: **end if**
40: **end if**
41: $(J, terminationTime) \leftarrow EarliestTerminatingJob(\ pstat,\ prgs,\ C[*][\chi']\ )$
42: **if** $J \neq \emptyset \wedge (terminationTime \leq PQueueFront(Q_E).key\ \vee Q_E = \emptyset)$ **then**
43: $PQueuePush(\ Q_E,\ [\ J,\ \text{'LBL-TERM'}\ ],\ terminationTime\ )$
44: **end if**
45: $lastTime \leftarrow time$
46: **end while**

Figure 3: The List Scheduling Algorithm, 'LS-SC'

Algorithm: $SchedRun$
Input: **job-id** J
Input: **time-type** $time$
Input: **processor-id** p
In/out: **schedule** S
In/out: **array** $[1..k]$ **of job status** $jstat$
In/out: **array** $[1..m]$ **of processor status** $pstat$
1: $ListAppend(\ S.start[J],\ time\ )$
2: $pstat[p].job \leftarrow J$
3: $jstat[J].proc \leftarrow p$

Algorithm: $SchedStop$
Input: **job-id** J
Input: **time-type** $time$
In/out: **schedule** S
In/out: **array** $[1..k]$ **of job status** $jstat$
In/out: **array** $[1..m]$ **of processor status** $pstat$
1: **if** $time = ListTail(S.start[J])$ **then** $ListEraseTail(S.start[J])$
2: **else** $ListAppend(\ S.end[J],\ time\ )$
3: $p \leftarrow jstat[J].proc$
4: $pstat[p].job \leftarrow \emptyset$
5: $jstat[J].proc \leftarrow \emptyset$

Figure 4: Primitive Schedule Operations

After processing the events the algorithm picks the highest-priority job from the job-ready queue Q_P and tries to schedule it on a processor. Note that due to the fact that at most one job changes its status per iteration we know that also *at most one highest-priority job needs to be scheduled*. If all processors are busy then adding a job into the schedule is possible only if at least one running job is 'preemptable'. If preemption is allowed then all jobs are preemptable. Otherwise only those jobs are considered 'preemptable' that have not really run yet, *i.e.,* those having zero progress. In the latter case no preemption is done in the usual sense, but instead the algorithm 'changes its mind' and undoes its scheduling decision in favor of a higher-priority job. In any case, if there are preemptable jobs we assign the least-priority one to J'. If the top-priority job J has a higher priority then it replaces job J' on its processor. This operation ensures that when all processors are busy then the m jobs that are running are either the highest-priority ready jobs or non-preemptable jobs that have started before the higher-priority waiting jobs got ready. In the case when job J replaces job J' the latter goes back to the waiting queue ($PQueuePush$), whereas its processor is taken by the highest-priority job J, which is popped from the waiting queue. In another case, when there is an available processor there is no need to check job priorities and the highest-priority job occupies an available idle processor.

Finally, the algorithm checks all running jobs to find the one that would be the earliest to terminate if not preempted. If there are no events in the event queue before the termination time of that job then the termination event is enqueued in the event queue, as no job can possibly preempt the given job before its termination.

Lemma 4.2 (Complexity of List Scheduling)**.** *With k the number of jobs, E the number of dependencies and m the number of processors, the complexity of the offline list scheduling is:*

$$O(k(\log k + m) + E)$$

Proof. The initial forming of the arrival-event priority queue of size k costs $O(k \log k)$ time[5]. There are $O(k)$ number of events, and hence $O(k)$ main-loop iterations of the list scheduler. Except for vis-

[5] This can be also seen as the time necessary for an efficient sorting of the jobs by their arrival time

iting the successors of the job, in every iteration we have either $O(1)$ operations (*e.g.,* a schedule operation), or $O(\log k)$ operations (priority queue and set operations), or operations with complexity $O(m)$, whose scope is the set of currently running jobs, in particular: $UpdateProgress$, $AllProcessorsBusy$, $LeastPrioPreemptableJob$, $GetAvailableProcessor$, and $EarliestTerminatingJob$. Finally, the total number of all $termPredecessors$-update operations during the whole run of the algorithm is $O(E)$. This reasoning yields the result stated in the lemma. □

4.4 Transformed List Scheduling

Recall that the goal of transformation is to generate the **HI*** table based on the **LO** table. If the basis algorithm is FPM-based list scheduling then one could consider to generate the two tables running the list scheduler twice: the first time in the LO mode with PT_{LO} priorities to obtain **LO** and then in the HI mode with PT_{HI} priorities to obtain **HI***. However, as explained earlier, in general such a naïve approach does not ensure that that it will be always safe to switch from **LO** to **HI***, in the sense that all running safety-critical jobs will always have enough processor-time reservation to execute up to their $C_j(HI)$ execution times.

Therefore, to ensure safety, in the HI mode we transform the basis policy – the list scheduler in the case considered here – according to the three rules formulated in Section 3. These three rules take the **LO** table as input and temporarily disable the jobs whose execution progress in the **HI*** risks to exceed the one in the **LO** table. The disabling is effective until the time when the jobs (re-)appear in the **LO** table. Executing the transformed policy with a task graph annotated by HI-WCET job execution times and containing HI job dependencies would yield a safe policy. In the end, one also has to check the satisfaction of deadlines, as in the case of multiple processors[6] the correctness of transformation is not guaranteed by construction even for a correct basis policy.

In this section we apply the transformation to the variant of the list scheduling algorithm introduced in the previous section. One of the peculiarities of the transformed algorithm is that at a given level of criticality it needs to know the schedule generated for the previous level of criticality. In dual-criticality problems the algorithm takes as input the schedule S_{LO}, representing the **LO** table and the task graph that represents the jobs and dependencies in the HI mode, as defined below.

Recall that in the transformed version of the algorithm we by default assume that preemption is allowed, as transformation of non-preemptive scheduling policies is postponed to future work.

Definition 4.3 (HI-criticality graph)**.** *The HI-criticality graph characterizing an MC scheduling problem instance* $\mathbf{T}(\mathbf{J}, \rightarrow)$ *is a directed graph* $\mathbf{T}_{HI}(\mathbf{J}_{HI}, \rightarrow_{HI})$*, where the* $\mathbf{J}_{HI}$ *is the subset of* $\mathbf{J}$ *that contains only HI jobs, and* $\rightarrow_{HI}$ *is the subset of* $\rightarrow$ *that contains only the dependencies of HI criticality level.*

The pseudo-code of the transformed list scheduling algorithm is given in Fig. 5.

As mentioned before, to construct S_{HI}, representing the **HI*** table, the algorithm needs the S_{LO} table at the input, which can, for example be obtained from the list scheduling in the LO mode or from any other valid algorithm. It is assumed that S_{LO} is correct, in particular that the jobs execute with LO-WCET execution times on m processors.

Though the schedule S_{LO} is constructed for all jobs and must respect all dependencies, the transformed algorithm 'cares' only about the HI jobs and dependencies. However, to ensure safety and to implement the Rules (1a, 1b, 1c) the algorithm keeps track of the progress of jobs not only in the HI mode, but also in the **LO** table. Therefore the job progress array '$prgs$' is now two-dimensional, adding another dimension to take the LO mode into account. To facilitate the calculation of progress in the LO mode the algorithm also constructs on-the-fly a new copy of schedule S_{LO} and adds a second dimension also to the arrays of job and processor status: $jstat$ and $pstat$. The algorithm also keeps track of the jobs terminated in the LO mode – variable $\mathbf{J}^{LO\text{-}term}$ – and the set of disabled jobs – $\mathbf{J}^{dis}$. For the rest, the transformed list scheduler has the same set of variables as the non-transformed one.

Similarly to the list scheduler, the algorithm starts by filtering the jobs and dependencies by their level of criticality and then pushes the arrival-time events of the filtered job set into the event queue. However,

[6] According to our preliminary results only the presence of multiple processors is important, whereas the other factor – the presence of dependencies still preserves the transformation correctness for preemptive scheduling on single processor. We will present the proof in future work. Note that for the transformed algorithm we do not support yet another factor that impacts correctness, namely the possible absence of preemption support, as we leave time-triggered variant of non-preemptive scheduling for future work.

Algorithm: $SimulateTranformedListSchedule$
Input: **integer** m **define** $k = size(\mathbf{J})$
Input: **task graph** $\mathbf{T}(\mathbf{J}(A[*], D[*], \chi[*], C[*][LO..HI]), \rightarrow_{LO..HI})$ **priority table** PT_{HI} **schedule** S_{LO}
Output: **schedule** S_{HI}
Local: **array** $[1..k][LO..HI]$ **of time-type** $prgs$ **set of jobs** $\mathbf{J}^{arr}$ **set of jobs** $\mathbf{J}_{\text{HI}}$ **dependencies** $\rightarrow_{\text{HI}}$
Local: **set of jobs** $\mathbf{J}^{dis}$ **schedule** S_{LO}^{copy}
Local: **array** $[1..m][LO..HI]$ **of processor status** $pstat$
Local: **array** $[1..k][LO..HI]$ **of job status** $jstat$
Local: **priority queue** Q_E, Q_P **array** $[1..k]$ **of integer** termPredecessors

1: $\mathbf{J}_{\text{HI}} \leftarrow \{ J \in \mathbf{J} \mid \chi[J] = HI \}$
2: $\rightarrow_{\text{HI}} \leftarrow \{\rightarrow_{\chi''} \mid \chi'' = HI \}$
3: $PQueuePushSet(\ Q_E,\ [\ \mathbf{J}_{\text{HI}},\ \text{'LBL-ARR'}\ ],\ A[*]\ )$
4: $PQueuePushScheduleEvents(\ Q_E,\ S_{\text{LO}},\ \mathbf{J}_{\text{HI}},\ \text{'LBL-LO-RUN'},\ \text{'LBL-LO-STOP'})$
5: $lastTime \leftarrow 0$
6: **while** $Q_E \neq \emptyset$ **do**
7: $([J, \text{LBL}], time) \leftarrow PQueuePop(Q_E)$
8: $UpdateProgress(\ lastTime,\ time,\ prgs[*][LO],\ pstat[*][LO]\ )$
9: $UpdateProgress(\ lastTime,\ time,\ prgs[*][HI],\ pstat[*][HI]\ )$
10: **switch** LBL **do**
11: **case** 'LBL-LO-LAG'
12: $SchedStop(\ J,\ time,\ S_{\text{HI}},\ jstat[*][HI],\ pstat[*][HI]\ )$
13: $SetAdd(\mathbf{J}^{dis}, J)$
14: **case** 'LBL-LO-STOP'
15: $SchedStop(\ J,\ time,\ S_{\text{LO}}^{copy},\ jstat[*][LO],\ pstat[*][LO]\ )$
16: **if** $prgs[J][LO] = C[J][LO]$ **then**
17: $SetAdd(\mathbf{J}^{LO\text{-}term}, J)$
18: **end if**
19: **case** 'LBL-LO-RUN'
20: $proc \leftarrow GetAvailableProcessor(pstat[*][LO])$
21: $SchedRun(\ J,\ time,\ proc,\ S_{\text{LO}}^{copy},\ jstat[*][LO],\ pstat[*][LO]\ )$
22: **if** $J \in \mathbf{J}^{dis}$ **then**
23: $SetRemove(\mathbf{J}^{dis}, J)$
24: $PQueuePush(\ Q_P,\ J,\ PT_{\text{HI}}^{-1}[J]\ )$
25: **end if**
26: $HandleListSchedEvents(\ J,\ LBL,\ time,\ Q_E, Q_P, PT_{\text{HI}}, \mathbf{J}^{arr}, \rightarrow_{\text{HI}}, termPredecessors)$
27: $ScheduleHighestPrioririty Job(\ S_{\text{HI}},\ Q_P,\ pstat[*][HI], jstat[*][HI],\ prgs[*][HI])$
28: **define** $lag(J) = (prgs[J][LO] - prgs[J][HI])$
29: $(minLag, J) = \min(lag(J) \mid J \in Running(pstat[*][HI]) \backslash \mathbf{J}^{LO\text{-}term} \wedge jstat[J][LO].proc = \emptyset)$
30: **if** $J \neq \emptyset\ \wedge\ (\ time + minLag \leq PQueueFront(Q_E).key\ \vee\ Q_E = \emptyset\ )$ **then**
31: $PQueuePush(\ Q_E,\ [\ J,\ \text{'LBL-LO-LAG'}\ ],\ time + minLag\ )$
32: **end if**
33: $EnqueueTermination(\ Q_E,\ prgs[*][HI],\ pstat[*][HI],\ C[*][HI]\ )$
34: $lastTime \leftarrow time$
35: **end while**

Figure 5: The Transformed List Scheduling for Generating **HI*** Table, '$\mathcal{T}$(LS-SC)'

in addition, it takes the execution intervals of the HI jobs in the S_{LO} table and pushes their begin and end bounds as events labeled as 'LBL-LO-RUN' and 'LBL-LO-STOP', respectively.

The main loop of the algorithm extends that of the list scheduler by also following the progress in a non-principal mode – LO – and handling certain events of that mode. The HI-mode events are handled by the regular list-scheduler event-label switch-case, after the LO events. For brevity, the regular event handling is represented by a call to subroutine '$HandleListSchedEvents$'. The presented switch-case operators handle the LO events.

First of all, 'LBL-LO-LAG' events are handled. The latter are added on-the-fly at the time-stamps where a HI-mode running job which is not running in the LO mode is going to reach the same progress as in the LO mode, whereupon it should be disabled due to invalidating Rule (1b). The so-called **'lag' time**, *defined as difference between the LO and the HI progress indicates the time interval during which the HI job may continue to run without running in the **LO** table while still not exceeding the LO-mode progress* (see the lag calculation rule after the switch-case).

The 'LBL-LO-RUN' and 'LBL-LO-STOP' events indicate the execution intervals of jobs in the **LO** table. They are used to update the LO-mode job and processor status as well as to enable the HI jobs at least during the time intervals when they are running in the LO table.

When a LO-mode-running (and, hence, enabled) job stops in the LO mode, which is indicated by event 'LBL-LO-STOP', and when the reason for the stop is that the job *terminates* in the LO mode (see the if-statement in the corresponding switch case) this means that for the HI mode it gets enabled permanently, according to Rule (1a). To register this change, the job is added into 'LO-mode terminated set', which eliminates the possibility of disabling the given job later on.

When a disabled job starts a new interval of execution in the LO mode, indicated by a 'LBL-LO-RUN' event, then it should be enabled (see the if-statement in the 'LBL-LO-RUN' case). The point is that a job can only get disabled when its HI-mode progress reaches exactly the level of its LO-mode progress, so the progress in the two modes is equal. Therefore, when the job starts running in the LO mode again then Rules (1a) and (1c) ensure that the job is enabled at least as long as it is running in the LO mode.

The jobs which have not yet terminated and are idle in the LO mode should eventually get disabled for running in the HI mode. Therefore, after the switch-cases for the event processing, the algorithm checks the lag time of such jobs and picks the one with the smallest lag. If this job will continue running it will be the first to 'exhaust' its execution-time safety reserve. If this happens before any other event in the event queue the algorithm 'knows for sure' that no earlier change in the schedule will prevent this from happening and enqueues a 'LBL-LO-LAG' event. Note that jobs running on other processors may also 'run out of their lag' at exactly the same time, but then they will be detected in the future iterations of the algorithm one-by-one.

Finally, the algorithm executes the regular list-schedule check for the earliest terminating, see the last 'if' statement of the regular list-schedule pseudo-code. For brevity, in Fig. 5 it is represented as a call to $EnqueueTermination$ subroutine. Note that the termination events are planned after the lag events to prevent the situation where a disabled job would be wrongly considered terminated.

Observation 4.4 (Ordering of the LO Events)**.** *We assume a similar restriction for the handling the LO events as for the regular list scheduling. The simultaneous events at the front of the queue should be popped in a particular order which coincides with the order of cases in the switch operator. In particular, events 'LBL-LO-LAG' should be popped first, to prevent that the job would be disabled immediately after being by a 'LBL-LO-RUN' event. Also 'LBL-LO-STOP' should precede 'LBL-LO-RUN' to ensure that the latter will always find an available processor to reconstruct the copy of the LO table. For the given time-stamp LO-events should be given preference to the regular events, to prevent that a job may get disabled and preempted at the same time.*

The evolution of a ready job in the transformed list scheduling is more complex than in the case of regular list scheduling. When a job gets ready it is pushed in the waiting queue Q_P and then its is eventually scheduled on a processor. Then it may be, either immediately or later on, stopped from execution and put either into disabled set $\mathbf{J}^{dis}$ if it is disabled or to the waiting-job queue Q_P if it is preempted. Upon being enabled a job goes into the waiting queue Q_P.

		Experiment 1		Experiment 2	
m	δ	jobs	arcs	jobs	arcs
1	0.005	*not simulated*		*not simulated*	
2	0.01	30	0	30	20
4	0.02	60	0	60	40
8	0.05	120	0	120	80

Table 1: Experiments' parameters

Lemma 4.5 (Complexity of Transformed List Scheduling)**.** *If the LO-mode schedule at the input of the algorithm was generated by a LO-mode list scheduling (or, equivalently, a fixed-priority policy) then the transformed list scheduling has the same algorithmic complexity as the non-transformed one, as was defined in Lemma 4.2:*

$$O(k(\log k + m) + E)$$

Proof. The 'start' and 'stop' LO-mode events have count $O(k)$ as they result from LO jobs getting ready, preempted, and terminated, whereas the number of preemptions in fixed-priority scheduling is $O(k)$. For the same reason, the number of the 'lag' events is also $O(k)$, so the number of main-loop iterations remains to be $O(k)$. The new operations added by transformations are also either $O(\log k)$ operations (priority-queue and set operations with $\mathbf{J}^{dis}$ and $\mathbf{J}^{LO\text{-}term}$) or $O(m)$ operations, such as finding the minimum-lag running job. Finally, the total number of all $termPredecessors$-update operations during the whole run of the algorithm is $O(E)$ like in the previous case. □

5 Experiments

In Appendix A we give a correctness result for the single-processor case without dependencies. To estimate the probability of getting a feasible solution in the multiprocessor case or with dependencies, we performed measurements for randomly generated problem instances and using an implementation of $MCPI$ and $\mathcal{T}(MCPI)$, described in Appendix B.

The random job generation algorithm was the same as in [6]. The instances were scaled to obtain a target '$Stress$' parameter [6], a processor resource utilization metric. For schedulability, the stress is bounded by m. For mixed-criticality problems, it is calculated separately for the LO and HI modes [6]. In our experiments the stress in the two modes was set to be equal to a target value S. For each instance, we first applied the MCPI algorithm [6]. If this did not produce a feasible schedule we canceled and restarted the experiment. In the case of a feasible schedule we applied the transformation algorithm to the result and checked if the experiment was a 'success', *i.e.,* whether a feasible STTM schedule was obtained. We performed two experiments, to test the effectiveness of the algorithm under different assumptions. In Table 5 the parameters used for each experiment are reported. It shows for each number of processors m the maximum allowed error δ on the Stress value and for each experiment the number of job and the number of precedence edges (*i.e.,* dependency arcs between jobs in the task graph). For each point, 1000 MCPI-schedulable instance were generated, in order to get a representative sample.

Experiment 1 shows the performance of the algorithm with no dependencies. Fig. 6 shows a plot of the success rate at different values of the normalized stress parameter S/m for 2, 4 and 8 processor problems. Also the theoretical success rate of 1 for single processor case is shown (confirmed by the experiments). We were not able to measure the success rate for S/m closely approaching to 1 due to the difficulty of finding high-stress problem instances that would be schedulable by MCPI.

The experiments show that the success rate is quite high, though it decreases with the stress and the processor count.

Fig. 7 shows the results of Experiment 2, where we added dependencies. From the graph we can see that there are no big difference in the case of task graph.

The curves are limited on the x axis because finding feasible schedule for randomly generated instances is computationally intractable for values of stress close to 1.

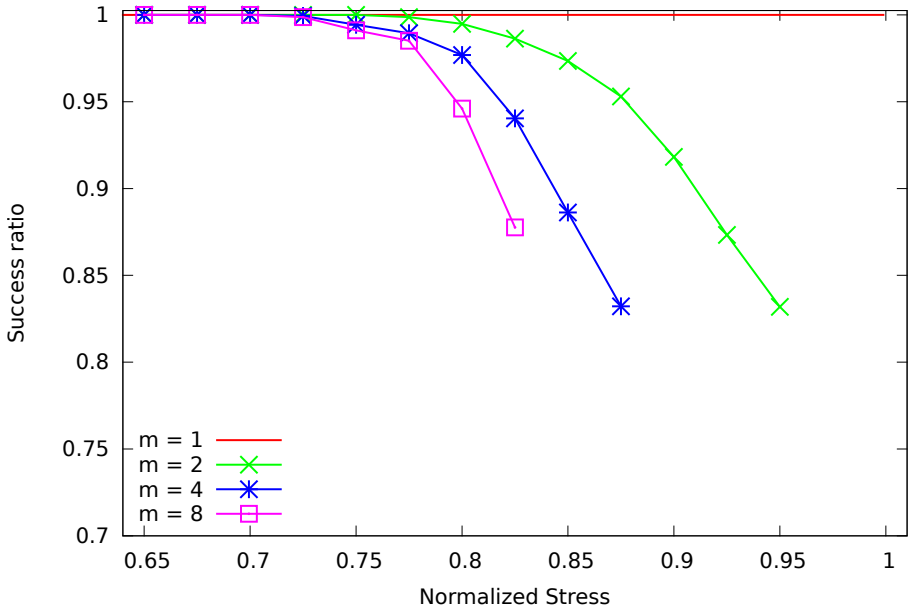


Figure 6: Experiment 1 - Without dependencies

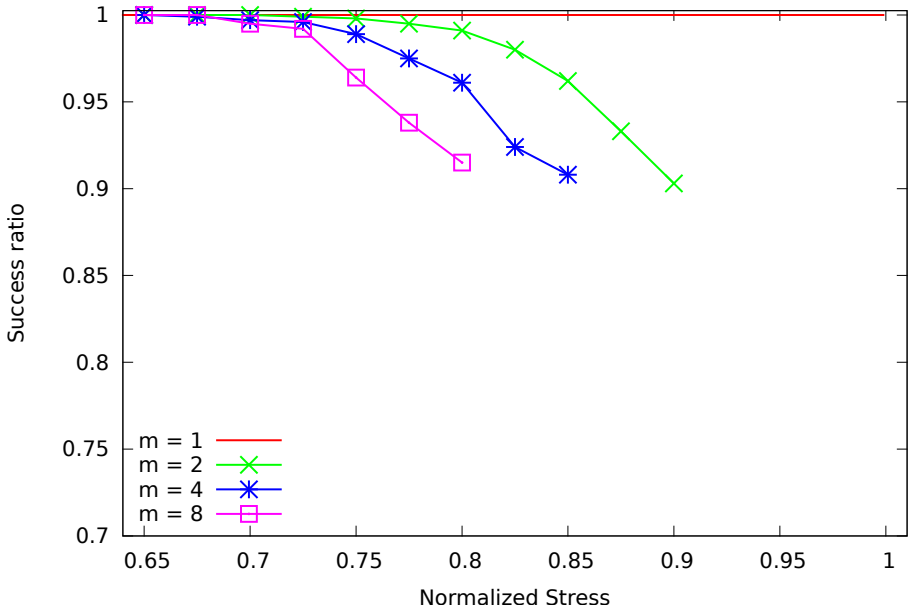


Figure 7: Experiment 2 - With dependencies

6 Conclusions and Future Work

In this paper we proposed a method to transform any memoryless scheduling policy into a time triggered (TT) one, which is not trivial in mixed-criticality problems if one wants to economize the processor resources, as this requires mode switching in order to give highly-critical jobs the exceptionally high level of resources required by certification only in the case of emergencies (*i.e.,* exceptional processor overloads).

For a single processor case our method was proven to work for any correct basis scheduling policy. For the multiprocessor case the same can be asserted in the grand majority of cases even the ones with large processor resource utilization, as confirmed by our experiments. In future, we plan to extend these results to more than two levels of criticality, bus and cache interference analysis, and pipelining.

References

[1] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Time-triggered mixed-critical scheduler on single and multi-processor platforms (invited paper)," in *Int. Conf. High Performance Computing and Communications*, HPCC'15, IEEE, 2015. 2

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

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

[4] S. Baruah and G. Fohler, "Certification-cognizant time-triggered scheduling of mixed-criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 3–12, 2011. 1, 2.4, 4.1

[5] P. Ekberg and W. Yi, "Bounding and shaping the demand of mixed-criticality sporadic tasks," in *ECRTS*, pp. 145–154, IEEE, 2012. 1

[6] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Multiprocessor scheduling of precedence-constrained mixed-critical jobs," in *ISORC 2015*, IEEE, 2015. 1, 2.3, 2.1, 5, B.4

[7] S. Baruah, "Semantics-preserving implementation of multirate mixed-criticality synchronous programs," in *RTNS*, pp. 11–19, ACM, 2012. 1, 4.1

[8] S. Baruah, "Implementing mixed-criticality synchronous reactive systems upon multiprocessor platforms." 1, 4.1

[9] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, 1973. 2.3

[10] R. Ha and J. W. S. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *Proc. Int. Conf. Distributed Computing Systems*, pp. 162–171, Jun 1994. 2.3

[11] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001. 4.3, B.2

[12] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Multiprocessor scheduling of precedence-constrained mixed-critical jobs," technical report TR-2014-11, Verimag, 2014. B, B.1, B.2, B.6, B.4

[13] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, Oct. 2011. B.1

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

[15] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Mixed critical earliest deadline first," technical report TR-2012-22, Verimag, 2012. B.1

[16] P. Poplavko, D. Socci, S. Paraskevas Bourgos, and M. B. Bensalem, "Models for deterministic execution of real-time multiprocessor applications," in *DATE'15*, 2015. B.4

[17] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Mixed critical earliest deadline first," in *Euromicro Conf. on Real-Time Systems*, ECRTS'13, pp. 93–102, IEEE, 2013. B.4

Appendices

A Proofs of Correctness for Single-processor Instances

In this section we always assume single-processor problem instances with preemption enabled and without task-graph dependencies. We give proofs for the theorems formulated in Section 3.

A.1 Direct Correctness

Below we recall the 'direct correctness result' for the transformation algorithm.

Theorem 3.2 (Transformation Correctness)**.** *For a given single-processor problem instance if the basis policy ALG is correct and reasonable then the policy $\mathcal{T}(ALG)$ is also correct.*

Let $TT_J^{HI(LO|HI\text{-}J')}$ be the termination time of J in **HI*** (respectively, **LO**, **HI-*J'***).

Theorem A.1. *Let J^{least} be the least priority HI job after a switch to HI mode. (Note that in the reasonable policy this is always a latest-deadline HI job). Then*

$$\exists J' \ : \ TT_{J^{least}}^{HI*} \leq TT_{J^{least}}^{HI\text{-}J'}$$

Let us first give some definitions and support lemmas.

Definition A.2. *A* busy interval *in some table (be it **LO**, **HI-J** or **HI*** table) is a maximal continuous interval of time where some jobs are enabled for execution.*

For table **HI*** we apply special rules defined earlier which can disable a job temporarily. When such rules are not applied, the busy intervals are obviously *open* intervals, because they are composed of union of (intersecting) open intervals between arrival and termination of different jobs. We state without proof that even with the extra rules we defined earlier for **HI***, the busy intervals remain to be open intervals.

For convenience, we use the term 'busy interval' also for the set of jobs that are enabled at least once inside the busy interval, and denote it BI, *e.g.,* BI^{HI*} for busy intervals in **HI***. Note that for this table, unlike the other tables, it is not always so that the total interval duration is exactly equal to the total work of jobs in BI, because there are rules that can temporarily disable a job after its arrival and before its termination. Therefore, the total work of jobs in BI^{HI*} can exceed the length of the busy interval. This also means that a job may belong to several busy intervals of **HI***.

In between BI, there are closed, sometimes single-point, *idle intervals*. For **HI***, we would like to distinguish an idle interval as a *hole* if inside this interval there are HI jobs that have arrived and not yet terminated, and are disabled because neither of the rules (1a), (1b), (1c) is true. The idle intervals that are not holes, are called *empty* intervals, *i.e.,* those where the job queue is empty.

For instance in Figure 1 in **HI*** there are two busy intervals: (0,8) and (8,11), thus we have a hole of size 0 at time 8. This happens because we have that immediately before time 8 J_1 is enabled by Rule (1a) while J_2 is disabled. On the other hand, at time 8 J_1 is disabled (because it terminates) while J_2 is enabled by Rule (1c).

The following proposition is well-known for fixed-priority policies, but needs to be re-established because we added the rules that can disable jobs.

Lemma A.3. *If J^{least} is the least priority (*i.e.*, the latest-deadline) HI job, then it terminates at the end of some busy interval BI^{HI*}.*

Proof. Let us assume by contradiction that J^{least} terminates inside a busy interval at time t. This means that at time t there is another enabled job (by definition of busy interval). If that is so, then J^{least}, having the least priority, should not be running at time t. □

Lemma A.4. *Let $BI^{HI*} = (a, b)$ be a busy interval in **HI***. At time a, the set of non-terminated HI jobs is the same in tables **LO** and **HI***, and for all of them holds that at time a the cumulative execution progress in **LO** is the same as in **HI***.*

Proof. Consider time a. The lemma thesis is obvious for any job that did not arrive yet, so in the sequel we consider only those jobs that have arrived.

If a job J is non-terminated in **LO** then it is non-terminated in **HI*** as well by Lemma 3.1. In addition, by the same lemma we have:

$$T_J^{HI*}(a) \leq T_J^{LO}(a) \tag{2}$$

On the other hand, if job J is non-terminated in **HI*** then the fact that it is not enabled at time a (by lemma condition) implies that Rule (1a) is false and hence the job is non-terminated in **LO** as well. Combined with the earlier observations, we conclude that the sets of non-terminated jobs at time a in these two tables are equal. In addition, also Rule (1b) is false, which means:

$$T_J^{HI*}(a) \geq T_J^{LO}(a) \tag{3}$$

Combining (2) and (3) we have the equality of the cumulative times. □

Corollary A.5. *Let $BI^{HI*} = (a, b)$ be a busy interval in which some job switches. Let J_s be the first such job, and let t_s be the time at which the switch occurs.*

*Then during the interval (a, t_s) tables **HI***, **HI-**J_s and **LO** are identical*

Proof. Notice that **HI-**J_s and **LO** are equal by construction in $(0, t_s)$ and hence in (a, t_s) as well. Let us compare **LO** and **HI***. At time a the set of non terminated jobs in these two tables are equal. In interval (a, t_s) no job switched yet, therefore all the jobs that run in **HI*** should satisfy Rule (1c), which is due to the fact that the other two rules require a switch to have occurred. As long as Rule (1c) holds, the **HI*** table replicates the **LO** table, and because it fills time interval (a, t_s) continuously, as $t_s \in BI^{HI*}$, we have proved our thesis.

□

Proof of Theorem A.1. Let $BI^{HI*} = (a, b)$ be the busy interval in which J^{least} terminates. By Lemma A.3, $TT_{J^{least}}^{HI*} = b$. By Lemma A.4, job J^{least} is not yet switched at start of this interval, and since this job terminates at the end of BI^{HI*}, we know also that it switches inside this interval as well, so Corollary A.5 applies for this interval.

Let us assume that $BI^{HI*} = (a, b)$ is followed by an empty interval, *i.e.,* an idle interval which appears due to termination of all HI jobs that have arrived so far. Because in this case all the jobs of BI^{HI*} have terminated by time b, we have:

$$b = a + \sum_{j \in BI^{HI*}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right)$$

Let J_s be the first job to switch in BI^{HI*}, at time t_s. By Lemma A.4 and Corollary A.5, we have that the same jobs, with the same remaining execution time as in **HI*** will run from time a in **HI-**J_s before the switch and, by construction after the switch as well. Therefore $BI^{HI*} = BI^{HI\text{-}J_s}$ and J^{least}, being the least-priority job, will terminate at time b in both tables.

Let us now examine the other case, in which $BI^{HI*} = (a, b)$, the busy interval where J^{least} terminates, is followed by a hole, *i.e.,* the idle interval which appears because at time b the rules for table **HI*** have disabled the non-terminated jobs. Also in this case J^{least} by our hypothesis and Lemma A.3 will terminate at time b, but in this case by construction not all jobs of BI^{HI*} terminate by time b:

$$b < a + \sum_{j \in BI^{HI*}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right) \tag{4}$$

Let J_s be the first job to switch in BI^{HI*}, at time t_s. Again by Lemma A.4 and Corollary A.5 we observe the same initial state and subsequent behavior in tables **HI*** and **HI-**J_s of all non-terminated HI jobs during the time interval $(a, t_s]$. So we conclude that all jobs of BI^{HI*} run in **HI-**J_s after time a continuously, at time a their total remaining work is equal to:

$$\sum_{j \in BI^{HI*}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right)$$

In line with equation (4), in order to complete this workload, table **HI-**J_s has to continue execution after time b. New jobs may arrive before the termination of the busy interval $BI^{HI\text{-}J_s}$. this busy interval executes all these jobs, J^{least} being the last one to terminate. So we have:

$$BI^{HI*} \subseteq BI^{HI\text{-}J_s}$$

and

$$TT^{HI\text{-}J_s}_{J^{least}} \geq a + \sum_{j \in BI^{HI*}} \big(C_j(\text{HI}) - T_j^{HI*}(a)\big) \tag{5}$$

Combining (4) and (5), and observing that $TT^{HI*}_{J^{least}} = b$, we have that also in this case in **HI-**J_s the least-priority job terminates no earlier than in **HI***. This completes the proof of Theorem A.1. □

Proof of Theorem 3.2. From Lemma 3.1 we know that in any possible scenario all the HI jobs will have enough processor resource to terminate. The termination time of J^{least} is guaranteed to meet the deadline due to the hypothesis that it meets deadline in the FPM policy and Theorem A.1. Now let us prove that also the HI jobs with higher priority in the EDF table PT_{HI} meet their deadlines. Let $J^{\overline{least}}$ be the next least priority HI job after J^{least} in the PT_{HI} table. Let **J** be the currently examined problem instance and let $\overline{\mathbf{J}}$ be the instance obtained from **J** by reducing the criticality of J^{least} to LO. It is easy to show that the HI-mode table $\overline{\mathbf{HI}*}$ obtained for this new instance coincides with **HI*** except that the intervals where J^{least} is running are idled. So, $J^{\overline{least}}$ will terminate in **HI*** at the same time as in $\overline{\mathbf{HI}*}$, where by Theorem A.1 applied to instance $\overline{\mathbf{J}}$ it will terminate no later than the latest termination under FPM policy. Obviously, also the latest termination of the FPM policy for job $J^{\overline{least}}$ is the same for both **J** and $\overline{\mathbf{J}}$. Because by our hypothesis this policy is feasible we conclude that $J^{\overline{least}}$ meets its deadline. Iterating this reasoning recursively, we argue that all HI jobs meet their deadline in **HI***, and thus we have our thesis. □

A.2 Reverse Correctness

In this section we prove the reverse correctness of transformation according to Theorem 3.3, *i.e.,* that for a reasonable basis policy ALG we have that $\mathcal{T}(ALG)$ can succeed only if the basis policy ALG succeeds. Similarly to the previous section, we first give some supplementary definitions and lemmas (in addition to those presented so far), and then we use them to establish a proof of the main theorem in the end of the section.

The *total remaining workload* when policy ALG executes basic scenario sc at time t is defined as:

$$WL^{sc}(t) = \sum_{j \in \mathbf{J}} \big(C_j(\chi_j^{sc}) - T_j^{sc}(t)\big)$$

where χ_j^{sc} is the criticality behavior shown by J_j in scenario sc. Similarly the total remaining *HI-job workload* is given as:

$$WL^{sc}_{\text{HI}}(t) = \sum_{j \in \mathbf{J}: \chi_j = \text{HI}} \big(C_j(\chi_j^{sc}) - T_j^{sc}(t)\big)$$

For table **HI*** we have:

$$WL^{HI*}(t) = WL^{HI*}_{\text{HI}}(t) = \sum_{j \in \mathbf{J}: \chi_j = \text{HI}} \big(C_j(\text{HI}) - T_j^{HI*}(t)\big)$$

Lemma A.6. *Given a reasonable basis policy, we have that:*

$$\forall sc, t \quad WL^{HI*}(t) \geq WL^{sc}_{HI}(t)$$

Proof. Before the mode switch, for any HI job j that did not terminate at time t in sc, we have that $C_j(\chi_j^{sc}) \leq C_j(\text{HI})$ by construction and $T_j^{sc}(t) \geq T_j^{HI*}(t)$ by Lemma 3.1. On the other hand, for a HI job that has terminated we have that $C_j(\chi_j^{sc}) - T_j^{sc}(t) = 0$. Thus we have $C_j(\text{HI}) - T_j^{HI*}(t) \geq C_j(\chi_j^{sc}) - T_j^{sc}(t)$ for all HI jobs j.

After the switch in sc, a reasonable policy will always execute a HI job when it can do so (*i.e.,* because the EDF policy is work-conserving and HI jobs have the highest priority). Thus, after the switch $WL^{HI*}(t)$ decreases at most as fast as $WL^{sc}_{\text{HI}}(t)$. □

Recall that a reasonable policy after the mode switch becomes priority-based and schedules HI jobs using the EDF priority table of HI jobs. Therefore, in this table we can identify the least priority job J_{least}.

Theorem A.7 (Worst Case Scenario)**.** *Let us consider a reasonable basis policy. Then, for all scenarios sc' and for the least priority job J_{least} we have:*

$$TT_{least}^{HI\text{-}J_s} \geq TT_{least}^{sc'}$$

*where J_s is the first job to switch in the busy interval in **HI*** where J_{least} terminates.*

In other words, $HI\text{-}J_s$ is the worst-case scenario for J_{least}.

Proof. In this proof we will use two **observations**:

1. after the switch we have $WL_{\text{HI}}^{sc} = WL^{sc}$.
2. consider to HI-job specific scenarios sc and sc' and some time instant t at or after the switching time of both scenarios; if at time t J_{least} did not yet terminate in neither of the two scenarios and $WL^{sc}(t) \geq WL^{sc'}(t)$, then $TT_{least}^{sc} \geq TT_{least}^{sc'}$; (this is so because after the switch a reasonable policy applies EDF, and for a fixed-priority policy the remaining workload has a monotonic impact on the termination time of the least priority job).

Let t_s be the time when J_s switches in **HI***. We know by Corollary A.5 that $WL^{HI*}(t_s) = WL_{\text{HI}}^{HI\text{-}J_s}(t_s)$. Then, by Lemma A.6:

$$\forall sc' \quad WL_{\text{HI}}^{HI\text{-}J_s}(t_s) \geq WL^{sc'}(t_s) \tag{6}$$

i.e., no scenario has more workload at time t_s than the scenario $HI\text{-}J_s$.

In the rest of the proof we assume that $t_{s'}$ is the switch time of another HI-job specific basic scenario $sc' = HI\text{-}J_{s'}$ and we compare that scenario to $sc = HI\text{-}J_s$.

For the scenarios where $t_{s'} \leq t_s$ the statement of the theorem is proved by the above stated Observation 2 and (6), as we have established the workload inequality for a time, t_s, that is at or later than the switch in the both scenarios.

Let us consider the other case, $t_{s'} > t_s$. Let us denote by t_{least} the time at which J_{least} terminates in the LO scenario (and, consequently, switches in the $HI\text{-}J_{least}$ scenario). Note that we can ignore the case $t_{least} < t_{s'}$, as in this case J_{least} would terminate in LO mode, and the LO basic scenario cannot be the unique worst case, as it is not worse than the scenario $HI\text{-}J_{least}$. So, we can assume $t_{s'} \leq t_{least}$. Due to this assumption, we also have: $t_{s'} \leq TT_{least}^{HI*}$ and $t_{s'} \leq TT_{least}^{HI\text{-}J_s}$. Therefore, $t_{s'}$ falls inside the busy interval where J_{least} terminates in the end, both for **HI*** and $HI\text{-}J_s$.

By construction, t_s and TT_{least}^{HI*} belong to the same busy interval BI^{HI*}, thus WL^{HI*} will constantly decrease in this interval. At time $t_{s'}$, we will have $WL^{HI*}(t_{s'}) = WL^{HI*}(t_s) - \mid (t_s, t_{s'}) \mid$. By a similar reasoning on the busy interval $BI^{HI\text{-}J_s}$, we have $WL^{HI\text{-}J_s}(t_{s'}) = WL^{HI\text{-}J_s}(t_s) - \mid (t_s, t_{s'}) \mid$.

Thus, using equality $WL^{HI*}(t_s) = WL_{\text{HI}}^{HI\text{-}J_s}(t_s)$, which we established earlier, we have:

$$\begin{aligned} WL^{HI\text{-}J_s}(t_{s'}) &= WL^{HI\text{-}J_s}(t_s) - \mid (t_s, t_{s'}) \mid \\ &= WL^{HI*}(t_s) - \mid (t_s, t_{s'}) \mid \\ &= WL^{HI*}(t_{s'}) \end{aligned}$$

Therefore, for time $t_{s'}$ we can repeat the same reasoning as we did for time t_s, which concludes the proof. □

Theorem 3.3. *For a given single-processor problem instance, under the assumption that the basis policy ALG is reasonable, we have that if the policy $\mathcal{T}(ALG)$ is correct then policy ALG is correct as well.*

Proof. Our thesis can be rewritten as:

$$(\forall j \quad TT_j^{HI*} \leq D_j) \;\Rightarrow\; (\forall sc, \forall i \quad TT_i^{sc} \leq D_i)$$

We prove the theorem for $J_i = J_{least}$ and then extend this argument from J_{least} to other jobs J_i by induction, in the same way as we did in the proof of Theorem 3.2 in the end of previous section.

Suppose by contradiction that J_{least} misses its deadline in ALG while all jobs meet their deadlines in $\mathcal{T}(ALG)$. We have:

$$TT_{least}^{HI*} \leq D_{least} < TT_{least}^{HI\text{-}J_s} \tag{7}$$

where $HI\text{-}J_s$ is the worst case scenario for J_{least} according to Theorem A.7. We distinguish two cases:

1. J_{least} **terminates before an "empty interval".**
   By the reasoning of the proof of Theorem A.1, we have:
   $$TT_{least}^{HI*} = TT_{least}^{HI\text{-}J_s}$$
   which contradicts (7).

2. J_{least} **terminates before a "hole".** Considering $BI^{HI*} = (a, b)$, as in the proof of Theorem A.1, and observing that, by Lemma A.4, $T_j^{HI\text{-}J_s}(a) = T_j^{HI*}(a)$ we have that:
   $$TT_{least}^{HI\text{-}J_s} = a + \sum_{j \in BI^{HI\text{-}J_s}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right) \tag{8}$$
   Let J_e be the last job to terminate in **HI***. For this job, by construction:
   $$TT_e^{HI*} \geq a + \sum_{j \in \mathbf{J}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right) \tag{9}$$
   The right side of Equation (9) is no less than the right side of Equation (8). Therefore, $TT_e^{HI*} \geq TT^{HI\text{-}J_s}$. Also, in EDF: $D_{least} \geq D_e$. From these observation and (7), we have:
   $$TT_e^{HI*} \geq TT_{least}^{HI\text{-}J_s} > D_{least} \geq D_e$$
   thus J_e will miss its deadline in **HI***, which contradicts the theorem assumptions.

□

B MCPI as Basis Algorithm

We describe in this appendix MCPI because by default it is used as the 'basis' algorithm ALG to serve as input for translation $\mathcal{T}(ALG)$ into time-triggered tables. MCPI stands for *Mixed Criticality Priority Improvement*. We give here only the information necessary to implement MCPI, whereas for a more complete description the reader is addressed to [12]. The presentation of the algorithm here is slightly adapted compared to [12]. It adds a few small generalisations necessary to extend MCPI for the support of non-preemptive scheduling, whereas by default the algorithm assumes preemptive scheduling.

B.1 MCPI Preliminaries

MCPI is an algorithm to compute job priorities offline. Consistently with the TT translation implementation described in Section 4, MCPI algorithm supports task-graph dependencies (precedence constraints) by building on top of classical *list scheduling*. The use of list scheduling makes it also more straightforward to present the extension of this algorithm to support non-preemptive scheduling. In fact, in the HI mode MCPI fully relies on classical list scheduling without any modification. Therefore, to obtain from MCPI algorithm the time-triggered variant $\mathcal{T}(\text{MCPI})$ we just use MCPI as is to generate the **LO** table and then derive from it the **HI** table by running the $\mathcal{T}(\text{LS-SC})$ described in Section 4. As the list scheduling algorithm itself, we allow *job migration* (*i.e.,* execution of different segments of a single job on different processors), but by construction it cannot occur in the non-preemption extension of the algorithm.

Fig. 8 shows an overview of MCPI. The algorithm takes as input the task graph $\mathbf{T}$, the number of processors m and an initial priority table PT. The latter may be generated by any known global fixed-priority algorithm for multiprocessors. We call this algorithm *support algorithm* and the initial table *support table*. By default we assume that the support algorithm is *EDF with modified deadlines and density threshold*, with two-step modification of deadlines for LO-mode table PT_{LO} [12]:

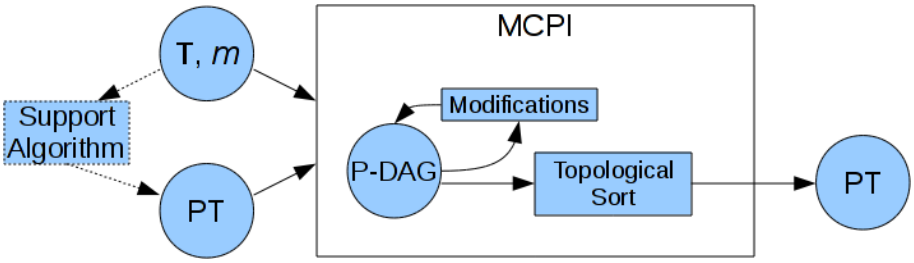


Figure 8: FPM algorithm MCPI (Basis algorithm ALG for TT translation $\mathcal{T}(ALG)$).

1. Subtract execution time uncertainty from the deadlines:

$$D_j^{mix} = D_j - (C_j(HI) - C_j(LO))$$

2. For task graph dependencies recursively propagate the deadlines from the graph sinks to sources:

$$D_j^*(LO) = \min_i (D_j^{mix}, D_i^*(LO) - C_i(LO) \mid J_i \text{ are LO-mode task-graph successors of } J_j)$$

whereas for the HI-mode table PT_{HI} we only apply the propagation of deadlines:

$$D_j^*(HI) = \min_i (D_j, D_i^*(HI) - C_i(HI) \mid J_i \text{ are HI-mode task-graph successors of } J_j)$$

In table PT_χ the priorities are assigned in the EDF way if the job density $\delta_j(\chi)$ is smaller than (experimentally determined) threshold $thr = 0.85$ and if $\delta(\chi) > thr$ then the jobs get the highest priority unconditionally. Here the job density is execution time - relative deadline ratio of the job: $\delta_j(\chi) = C(\chi)/(D_j^*(\chi) - A_j)$. Giving the highest priority to high-density jobs is a necessary technique to overcome the so-called Dhall effect that adversely impacts the EDF-based tables on multiple processors [13].

Deadlines D^{mix} are referred to as mix-mode deadlines because they are applied to the LO mode while taking into account the HI mode execution time. It is straightforward to show [14] that if a job misses in the LO mode its D^{mix} deadline then if it switches to the HI mode it will also miss the real deadline, therefore meeting this deadline is a necessary condition in the LO mode.

Our "priority improvement" algorithm MCPI tries to improve the priority table generated by the support algorithm so that the response times of HI jobs can be improved and thus the mixed-critical schedulability criteria can be met for a larger set of problem instances. The algorithm is based on the concept of *Priority Direct Acyclic Graph*(P-DAG), which defines a partial order on the jobs showing *sufficient* priority constraints needed to obtain a certain schedule. We build such a structure by adding, at each step, jobs from support priority table PT, starting from the one with the highest priority. Each time we add a HI (*i.e.,* safety-critical) job, we apply a modification to the priority order given by table PT, to increase the schedulability of safety-critical scenarios. The modification is done in a '*bubble-sort*' way, *i.e.,* we fist put the job at the least priority position and then try raise its priority by swapping it with the job at the previous position. We only swap a HI job with a LO job (never with another HI job) and we accept the swap only if LO job and the other jobs with less priority do not start missing their deadlines. Note that we do not do a usual 'bubble-sort' on a *total (linear) order* (the priority table), as such a naïve approach may encounter some artificial hazards [15]. Instead, we move the HI jobs along a *partial (tree-like) order* defined by the P-DAG. When all jobs have been added to the P-DAG (with an improvement attempt for each HI job), a priority table PT_{LO} is obtained by topological sort of the P-DAG.

The algorithm improves the support priority table only for the LO-mode table, whereas it keeps the HI mode table intact. Therefore, the main goal of the algorithm to compute and validate the PTLO, which we will simply denote as PT in the sequel. To construct the PT, MCPI takes the priority table generated by the support algorithm and tries to improve the HI scenarios schedulability by 'bubble-sorting' strategy mentioned above which increases the priorities of HI jobs as much as possible without undermining the LO-mode schedulability. When the table the ready, the algorithm also tests the schedulability in HI-mode scenarios.

Before specifying the algorithm itself, we give some definitions and propositions.

Definition B.1 (Blocking Relation '$\vdash$' between Jobs)**.** *Given two jobs J_1 and J_2 and priority table PT, we say that a higher-priority job J_1* blocks *a lower-priority job J_2 ($J_1 \vdash_{PT} J_2$) if there is a point in time t the list scheduler has to select a job to execute on one of m processors from a list of ready jobs where both J_1 and J_2 are present and it selects J_1 (due to its higher priority) whereas J_2 is not selected because J_1 and, possibly, other higher-priority jobs have occupied all the processors that become available at time t.*

Observation B.2 (Non-preemption Support and Blocking)**.** *It is mainly in this definition of the blocking relation (which is fundamental for MCPI) that we generalize MCPI to also support non-preemptive scheduling. For the preemptive case, this definition is equivalent to the one given in [12], it but it preserves all the useful properties of the blocking relation also for the non-preemptive case as their proofs from [12] remain valid.*

Note that by definition, we should have:

$$J_1 \vdash J_2 \implies J_1 \succ J_2 \tag{10}$$

It should be also noted that the blocking relation may only exist between two jobs that do not have a directed task graph path (*i.e.,* a dependency) between them, as task-graph predecessors and successors may never appear at the same time in the list of ready jobs of the list scheduling algorithm.

Definition B.3 (Potential Interference Relation)**.** *Given task graph $\mathbf{T}(\mathbf{J}, \rightarrow)$, number of processors m and a subset $\mathbf{J}' \subseteq \mathbf{J}$, we say that an equivalence relation $\overset{\mathbf{J}'}{\sim}$ on set $\mathbf{J}'$ is a 'potential interference' relation if it has the following property:*

$$\forall J_1, J_2 \in \mathbf{J}'.\ \exists PT : J_1 \vdash_{PT} J_2 \ \Rightarrow\ J_1 \overset{\mathbf{J}'}{\sim} J_2$$

whereby we consider LO-mode m-processor list schedules on maximal task subgraph with nodes $\mathbf{J}'$.

In general, there exist multiple potential interference relations, as joining two equivalence classes would lead to a new potential interference relation. Therefore, the (unique) maximal such relation is the total equivalence. The (unique) minimal potential interference relation can be obtained by union of blocking relations under all possible PT's, followed by transitive and reflexive closure, however it is a costly computation due to exponential number of PT's. Instead of computing this minimum, we over-approximate it by exploiting the following theorem (given without proof).

Theorem B.4 (Single-Processor Interference)**.** *In list scheduling (both the preemptive and non-preemptive one), a potential interference relation for a single processor is also a potential interference relation for m processors.*

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

As it turns out, calculating the minimal potential interference on a single processor can be done by a fast (almost linear) algorithm - the '*makespan*'. This algorithm simulates a single-processor list schedule of the task graph where instead of selecting the highest-priority ready job *arbitrary* ready job is selected to execute next. The goal is to obtain the list of *busy intervals* (see Definition A.2). MCPI assumes that $J_1 \overset{\mathbf{J}'}{\sim} J_2$ only if in the makespan simulation of job set $\mathbf{J}'$ the two jobs belong to the same busy interval.

B.2 MCPI Algorithm Specification

The pseudocode of the algorithm is given in Fig. 9. The algorithm takes as inputs the *support priority table* SPT and the task graph $\mathbf{T}$. We require the total order defined by SPT to be compliant with task-graph dependency partial order '$\rightarrow$':

$$J_1 \rightarrow^* J_2 \implies J_1 \succ J_2 \tag{11}$$

We require this property from SPT and ensure that it is preserved in the improved priority tables as well. Among other, this is needed to ensure that the jobs are handled by the algorithm in topological order: from

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

Figure 9: The MCPI algorithm

task-graph sources to task-graph sinks. To ensure the compliance to the task graph, the algorithm calls the $DependencyComplianceTransform$ algorithm, which produces a new SPT table by sorting the jobs such that, firstly, the requirement above is satisfied if there is a directed path between the jobs, and, secondly, the original SPT ordering is preserved otherwise.

We then check LO scenario schedulability, by running the list scheduler with priorities SPT in the LO mode. If the schedulability holds, it will be kept as an invariant during the execution, otherwise the algorithm terminates with a failure (not shown in the pseudocode). Subroutine $MCPI\_PDAG$ generates a (directed-forest shaped) P-DAG, based on the support priority table SPT and bubble-sort-like priority improvements for the HI jobs. Then we obtain a priority table from G by using the well-known $TopologicalSort$ procedure (see *e.g.,* [11]), which traverses the trees in G from the leafs to the roots while adding the visited nodes to PT. Finally, the subroutine $anyScenarioFailure$ checks the schedulability in any possible switch to the HI mode. The check is done by a simulation over the set of all job-specific scenarios $HI\text{-}J_h$ as explained in Section 2.[7]

In Fig. 10 subroutine *MCPI_PDAG* is shown. It takes as inputs the task graph $\mathbf{T}$, the support priority table SPT, and the graph G generated so far (that will be empty at the beginning). In every iteration, the algorithm handles J^{curr}, the highest-priority job of table SPT which is not yet in G and eventually adds that job to G. The algorithm terminates when all jobs have been added to G.

First, the current job is added to a priority table to a position inferior to all jobs handled in the previous iterations, using priority-table concatenation operator '$\frown$'. List-schedule simulation is carried out to discover which of the previous jobs would block the current job when that job has the least priority. We say that the blocking relation $\vdash$ is thus calculated. We also estimate the potential interference relation, for which we currently use the makespan algorithm to derive the single-processor busy intervals as explained earlier, though better approximations of potential interference are to be investigated in future work to take into account the number of available processors. After that:

if the current job criticality is LO we add an arc to J^{curr} from all the roots of the trees ST present in G where $\exists\ J' \colon J' \vdash J^{\text{curr}}$. This makes J^{curr} the new root of ST. This is needed to ensure that the priorities derived from G are compliant to Equation (10). In addition we do the same for the subtrees containing task-graph predecessors of J^{curr}, to ensure compliance to Equation (11).

if the current job criticality is HI we do similar actions as in the case of LO job, but instead of using the blocking relation we use the potential interference relation. The reason for this difference is that for HI jobs the final priority of J^{curr} is not known *apriori* as for such jobs 'bubble-sort' priority improvements are applied.

The priority improvements for the HI jobs are done by subroutine $PullUp$. This subroutine is the core of the algorithm. It modifies the P-DAG generated so far, trying to improve the HI schedulability of the

[7] Note that the list scheduling itself does not satisfy the definition of 'predictable policy', thus being unsuitable as an online policy and failing to satisfy the precondition of Theorem 2.2. Nevertheless, there exist other policies, which are on the one hand predictable and on the other hand are equivalent to the list scheduling for the basic scenarios, which justifies the applicability of MCPI in practice. However, in this work we do not need such policies as we use instead the STTM policy online.

Algorithm: $MCPI\_PDAG$
Input: **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
Input: **priority table** SPT
In/out: **forest P-DAG** $G(\mathbf{J}', \rhd)$

while $G.\mathbf{J}' \neq \mathbf{T}.\mathbf{J}$ **do**
  $J^{\text{curr}} \leftarrow SelectHighestPriorityJob(\mathbf{T}.\mathbf{J} \setminus G.\mathbf{J}', SPT)$
  $\mathbf{J}'' \leftarrow G.\mathbf{J}' \cup \{J^{\text{curr}}\}$
  $PT'' \leftarrow (TopologicalSort(G) \frown J^{\text{curr}})$
  $\mathbf{T}'' \leftarrow MaximalSubgraph(\mathbf{T}, \mathbf{J}'')$
  $\vdash \leftarrow SimulateListSchedule(\text{LO},\ \mathbf{T}'',\ PT'')$
  $\overset{\mathbf{J}''}{\sim} \leftarrow EstimateInterference(\text{LO},\ \mathbf{T}'')$
  $G.\mathbf{J}' \leftarrow \mathbf{J}''$
  for *all trees* $ST \in G$ **do**
    if $\chi(J^{\text{curr}}) = \text{LO}$ **then**
      if $\exists\ J' \in ST:\ J' \vdash J^{\text{curr}} \ \vee\ J' \rightarrow J^{\text{curr}}$ **then**
        $ConnectAsRoot(ST, J^{\text{curr}})$
      end if
    else
      if $\exists\ J' \in ST:\ J' \overset{\mathbf{J}'}{\sim} J^{\text{curr}} \ \vee\ J' \rightarrow J^{\text{curr}}$ **then**
        $ConnectAsRoot(ST, J^{\text{curr}})$
      end if
    end if
  end for
  if $\chi(J^{\text{curr}}) = \text{HI}$ **then** $PullUp(J^{\text{curr}}, G, \mathbf{T}, SPT)$
end while

Figure 10: The algorithm for computing priority tree in MCPI

initial priority order. Note that if this subroutine were not called, the algorithm would just generate a P-DAG that would correspond to the initial priority table SPT, thus not bringing any improvement on the results of the support algorithm.

Procedure $PullUp$ is described in the pseudocode in Fig. 11. The idea behind this subroutine is to try to improve the schedulability of HI scenarios by raising the priorities of HI jobs, "swapping" their position in the graph with LO jobs while keeping the LO scenario schedulability an invariant.

Procedure $LOpredecessors(J, G)$ returns for node J the set of its direct P-DAG predecessors[8] of LO criticality: $\{J_s \mid J_s \rhd J,\ \chi_s = \text{LO}\}$. At each step in Fig. 11 we pick the least priority predecessor from the working set $LOpredecessors(J, G) \setminus DONE$, where $DONE$ is a set that keeps track of the job we already tried to swap. Then subroutine $CanSwap$ checks if J and J' can swap priorities. If so, we apply the actual swapping transformation to graph G, otherwise the job J' will remain P-DAG predecessor of J but we will not try to swap J with that job again. The subroutine proceeds until we have tried to swap for all LO P-DAG predecessors of job J.

As shown in Figure 12, subroutine $CanSwap$ uses a private copy of graph G to perform a tentative swap modification and then evaluates its impact on the whole original job instance. To do so, it constructs a complete priority table by concatenating the one obtained from graph G with the trailer of SPT table that contains jobs that were not yet handled. Note that the latter jobs are identified by having less SPT-priority than the current HI job J, therefore we denote the 'trailer' part as $(SPT \mid\prec J)$. Note that thus we check the whole job set of the problem instance and not only the jobs whose priorities have been changed. This is required on a multi-processor because, unlike in single-processor case, changing the priorities of a pair of jobs may impact the schedulability of not only these jobs but of all jobs that have less priority. We accept the swapping only if it does not lead to a deadline miss for any job. This way, we maintain the schedulability in LO mode as an invariant of the algorithm. Note that $CanSwap$ immediately rejects to swap J and J' if $J' \rightarrow^* J$, to maintain the precedence compliance of priorities.

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

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

Algorithm: $PullUp$
Input: **job** J
In/out: **forest P-DAG** G
Input: **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
Input: **priority table** SPT
1: $DONE = \emptyset$
2: **while** $LOpredecessors(J, G) \neq DONE$ **do**
3: $J' \leftarrow SelectLeastPriorityJob(\ (\ LOpredecessors(J, G)\ \setminus\ DONE),\ SPT)$
4: $DONE \leftarrow DONE \cup \{J'\}$
5: **if** $CanSwap(J, J', G)$ **then**
6: $TreeSwap(J, J', G)$
7: $DONE \leftarrow\ DONE\ \cap\ LOpredecessors(J, G)$
8: **end if**
9: **end while**

Figure 11: The pull-up subroutine

Algorithm: $CanSwap$
Input: **HI job** J
Input: **LO job** J'
Input: **forest P-DAG** G
Input: **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
Input: **priority table** SPT
1: **if** $J' \rightarrow^* J$ **then**
2: **return** **False**
3: **end if**
4: $TreeSwap(J, J', G)$
5: $PT \leftarrow (TopologicalSort(G) \frown (SPT\ |\prec J))$
6: $allDeadlinesMet \leftarrow SimulateListSchedule(\text{LO},\ \mathbf{T},\ PT)$
7: **return** $allDeadlinesMet$

Figure 12: The subroutine for checking the feasibility of a priority swap

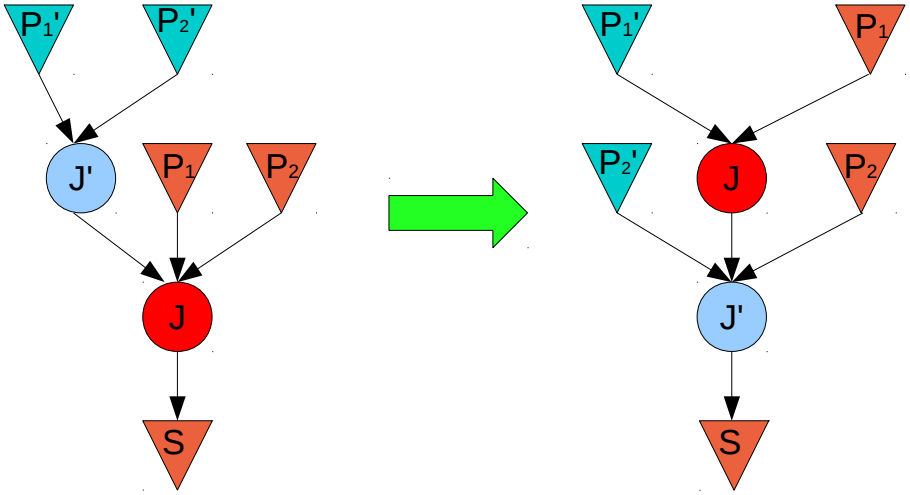


Figure 13: The effect of a Swap.

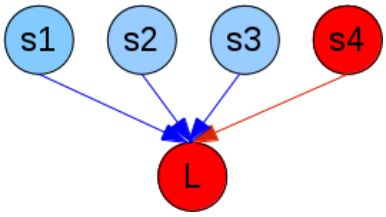


Figure 14: The graph of an airplane localization system illustrating LO→HI dependencies.

Definition B.5 (Swap)**.** *Let $G(\mathbf{J}', \rhd)$ be a forest P-DAG, let $J_{LO} \rhd J_{HI}$ and let $\mathbf{J}''$ represent the subset of jobs whose priorities can be potentially higher than or equal to J_{HI} after the swap is performed:*

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

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

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

The swap is illustrated in Fig. 13 for $J_{\text{HI}} = J$ and $J_{\text{LO}} = J'$. In this example the red triangle marked with S represent the successors of J, while the triangles marked with P_1, P_2 and P_1', P_2' are, respectively the predecessors of J and J'. More specifically, we assume in the figure for P_1 and P_1' are subtrees where the condition 'contains a job that is either potentially interferes with J or is a predecessor of J' is true, is true, while it is false for P_2 and P_2'.

When the swap is done, the $PullUp$ subroutine updates the set $DONE$ and reiterates.

B.3 MCPI Example

Let us first consider a realistic example and apply MCPI to it. This example also argues of practical rationale for supporting task-graph dependencies from LO jobs to HI jobs. The task graph is shown Fig. 14.

There we have a task graph of the 'localization system of an airplane', composed of four sensors (jobs s1-s4) and the job L, which computes the position. Data coming from sensor s4 is necessary and sufficient

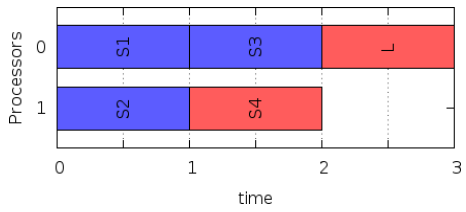


Figure 15: The LO-mode schedule from support table SPT.

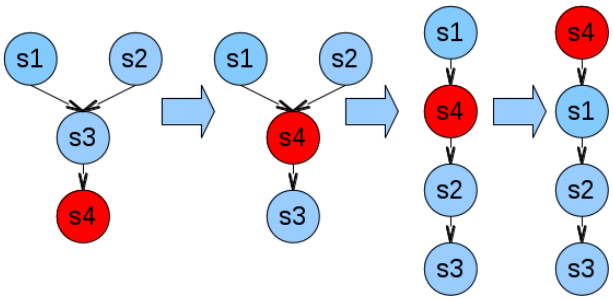


Figure 16: The effect of subroutine $PullUp$ on job $s4$.

to compute the plane position with a safe precision, thus only s4 and L are marked as HI-critical. On the other hand, data from s1, s2 and s3 may improve the precision of the computed position, thus granting the possibility of saving fuel by a better computation of the plane's route. So we do want job L to wait for all the sensors during normal execution, but when the systems switches to HI mode we only wait for data coming from s4.

Let us now assume that we have two processors ($m = 2$) and that the support table SPT is an EDF table:

$$PT = \{s1 \succ s2 \succ s3 \succ s4 \succ L\}$$

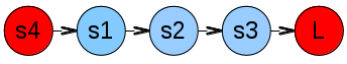
and that the nodes of this task graph are defined as follows:

Job	A	D	χ	C(LO)	C(HI)
s1	0	3	LO	1	1
s2	0	3	LO	1	1
s3	0	3	LO	1	1
s4	0	4	HI	1	3
L	0	6	HI	1	3

Let us apply MCPI to this example. The table SPT is already precedence compliant, so the dependency compliance subroutine will not modify it. Then we check LO schedulability, by simulation. The result of the simulation of the LO scenario is the Gantt chart of Fig. B.3, where it is easy to check that no jobs miss its deadline.

Then we apply subroutine *MCPI_PDAG*. In the first iteration we add $s1$ to G. It is not blocked by any other job, so we proceed with the second iteration. $s2$ is added to G, again we do not have any blocking. Next we add job $s3$, and we have the following blocking relations: $s1 \vdash s3$ and $s2 \vdash s3$. Thus we add the following edges to G: $s1 \rhd s3$ and $s2 \rhd s3$. Then we add $s4$. Since it is a HI job and $s4$, we add the edge $s3 \rhd s4$, since $s3$ is the root of the only tree of G.

Since $s4$ is a HI job, we run $PullUp$ on it. First we swap it with $s3$, after checking that after this operation the jobs will still meet their deadlines. Then we swap it also with $s1$ and $s2$. The result of *PullUp* subroutine is shown in Fig. 16. Finally we add job L to the graph and the edge $s3 \rhd L$. Since $s3 \rightarrow L$, we may not swap further, thus obtaining the following P-DAG:



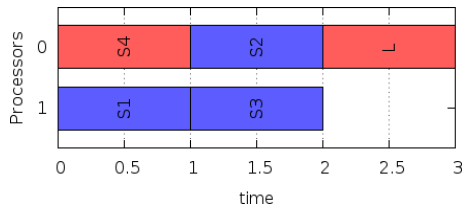


Figure 17: The schedule obtained by MCPI in the running example.

From topological sort we obtain the priority table $PT = \{s4 \succ s1 \succ s2 \succ s3 \succ L\}$. The priority table thus obtained leads to the schedule of Fig. B.3.

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

B.4 MCPI Overhead and Performance

Theorem B.6 (MCPI Algorithmic Complexity)**.** *[12] Let k be the number of jobs in '$\mathbf{J}$', E the number of precedence edges in '$\rightarrow$' and m the number of processors. The computational complexity of MCPI is*

$$O(Ek^2 + k^3(\log k + m))$$

Proof. (sketch) This result follows from the list scheduling complexity $O(E + k(\log k + m))$ and from the fact that the algorithm construct the priority table in a bubble-sort manner and runs a list scheduler at each swapping of two jobs in the table, whereas the number of bubble-sort swappings is $O(k^2)$. □

The complexity is dominated by the time do repetitive simulations in order to evaluate the blocking relation and the swapping. This time can be considerably optimized to be much smaller than the worst case by economizing in simulations *e.g.,* by not simulating the jobs that (according to busy intervals) cannot interfere with the jobs being tested in the given simulation. Currently our implementation for MCPI is not yet optimized with that respect and it took **2 minutes** handle the flight management system use case with **812 jobs** and almost **2000 task graph arcs** which we presented in [16]. On the other hand, the previous algorithm MCEDF (for single-processor scheduling) [17], whose implementation is much better optimized, can handle this example in less than a second. Therefore we believe that in the near future we can achieve better MCPI performance even for such examples. Note that such large examples are not uncommon if we obtain them from a hyperperiod of a multi-periodic system.

In [6, 12] we show 15%-30% improvement in schedulability of MCPI compared to its default support algorithm for the problem instances whose sum of loads in two modes (LO and HI) is larger than the maximal load the system could possibly have in a single mode.