Scheduling of certifiable mixed-criticality systems

Dario Socci

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Dario Socci

Thèse dirigée par **Saddek Bensalem**
et codirigée par **Petro Poplavko**

préparée au sein **du laboratoire VERIMAG**
et de **l' École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Scheduling of Certifiable Mixed-Criticality Systems

Thèse soutenue publiquement le **9 March 2016**,
devant le jury composé de :

Prof. Florence Maraninchi
Verimag, Président
Prof. Sanjoy Baruah
The University of North Carolina, Rapporteur
Prof. Alan Burns
University of York, Rapporteur
Dr. Luca Santinelli
ONERA, Examinateur
Dr. Madeleine Faugère
Thales Research & Technology, Examinateur

Acknowledgements

After four years of work in Verimag my PhD thesis is finally ready, and there are a lot of people that I have to thank for making this possible.

Above all of them there are my supervisors, professor Saddek Bensalem and doctor Petro Poplavko. Saddek put his trust on me, first by hiring me after a short Skype interview and later by giving me the freedom to develop my own ideas. He is an excellent director, being capable of motivating and directing my work without letting me feel under pressure. Then I would like to thank Petro for being always available when I needed help for my research, and his suggestions were always brilliant. Most of the results of this thesis would not have been achieved without the long time spent discussing together problems and new ideas. And also thanks to Marius, Jacques and all other members of the team.

Then I would like to thank all people from Verimag. The relaxed atmosphere you can breathe here, the skill and knowledge of professors and researchers and the kindness of the administrative staff makes it by far the best laboratory I know where to work on a research project in ideal condition. Working here as been both a pleasure and a privilege. A special thank goes to my friend and colleague Stefano for the hundreds of relaxing coffee breaks, after-work beers and philosophical conversations.

Also I would like to thank my family for the support they always gave me. In particular I would like to thank my mother, for being always helpful despite the distance and for the hundreds of culinary suggestions. Also a special thank goes to my father for the wine (which is quite an important matter).

Obtaining the PhD degree is a period of transition in life, one experience is over and new ones are coming. I don't know what waits for me in the future, but of one fact I am sure: whatever will happen next, I will face it with my girlfriend Irina beside me. For this reason, the most special thank goes to her: thanks for making the last year a very special one, meeting you was the most fortunate event.

Abstract

Modern real-time systems tend to be *mixed-critical*, in the sense that they integrate on the same computational platform applications at different levels of criticality (*e.g.,* safety critical and mission critical). Integration gives the advantages of reduced cost, weight and power consumption, which can be crucial for modern applications like Unmanned Aerial Vehicles (UAVs). On the other hand, this leads to major complications in system design. Moreover, such systems are subject to certification, and different criticality levels needs to be certified at different level of assurance.

Among other aspects, the real-time scheduling of certifiable mixed critical systems has been recognized to be a challenging problem. Traditional techniques require complete isolation between criticality levels or global certification to the highest level of assurance, which leads to resource waste, thus loosing the advantage of integration. This led to a novel wave of research in the real-time community, and many solutions were proposed. Among those, one of the most popular methods used to schedule such systems is Audsley approach. However this method has some limitations, which we discuss in this thesis. These limitations are more pronounced in the case of multiprocessor scheduling. In this case priority-based scheduling looses some important properties. For this reason scheduling algorithms for multiprocessor mixed-critical systems are not as numerous in literature as the single processor ones, and usually are built on restrictive assumptions. This is particularly problematic since industrial real-time systems strive to migrate from single-core to multi-core and many-core platforms.

Therefore we motivate and study a different approach that can overcome these problems. For this reason we assume a fixed set of jobs as workload model. This model can represent a hyperperiod of synchronous periodic tasks or servers. These removes some fundamental difficulties of non-synchronous systems (at risk to increase costs in some cases), thus leaving us more space to focus on limitations of Audsley approach and on overcoming the schedulability complications brought about by multiprocessor systems. Fixed job sets permit us to manipulate their priorities by using the novel concept of priority graph (P-DAG), which defines minimal relation between priorities in a schedule. Based on this formalism we propose two priority based algorithms. The first algorithm, Mixed Criticality Earliest Deadline First (MCEDF), is a single processor algorithm that dominates state-of-the-art Audsley approach based algorithm *Own Criticality Based Priority* (OCBP). The second one is a multiprocessor algorithm, Mixed Criticality Priority Improvement (MCPI), that, given a global fixed-priority assignment for jobs, can modify it in order to iteratively improve its schedulability for mixed-criticality setting. Our experiments show an increase of schedulable instances up to a maximum of 30% if compared to classical solutions for this category of scheduling problems.

A restriction of practical usability of many mixed-critical and multiprocessor scheduling algorithms is assumption that jobs are independent. In reality they often have precedence

constraints. In the thesis we show the mixed-critical variant of the problem formulation and extend the system load metrics to the case of precedence-constraint task graphs. We also show that our proposed methodology and scheduling algorithm MCPI can be extended to the case of dependent jobs without major modification and showing similar performance with respect to the independent jobs case.

Another topic we treated in this thesis is time-triggered scheduling. 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 analyses 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.*.

The trivial method of obtaining a time-triggered schedule is simulation of the worst-case scenario in event-triggered algorithm. However, when applied directly, this method is not efficient for mixed-critical systems, as instead of one worst-case scenario they have multiple corner-case scenarios. For this reason, it was proposed in the literature to treat all scenarios into just a few tables, one per criticality mode. We call this scheduling approach Single Time Table per Mode (STTM) and propose a contribution in this context. In fact we introduce a method that transforms practically *any scheduling algorithm* into an STTM one, which is again based on a simulation, but it is adapted to ensure safe switching between the scenarios. It works optimally on single core and shows good experimental results for multi-cores. In addition we show that applying it to list scheduling leads to support of task graph (precedence) dependencies, for which our method also shows good experimental results.

Finally we studied the problem of the practical realization of mixed critical systems. This is a challenging task due to a semantic gap between real-time scheduling policies and the various models of computation proposed for programming timing-critical concurrent systems. To overcome this difficulty, we represented both the models of computation and the scheduling policies by timed automata. We believe that using the same formal language for the model of computation and the scheduling techniques is an important step to close the gap between them. Our effort in this direction is a design flow that we propose for multicore mixed critical systems. In this design flow, as the model of computation we propose a network of deterministic multi-periodic synchronous processes. Our approach is demonstrated using a publicly available toolset, an industrial application use case and a multi-core platform. An ongoing work is integration of the proposed design flow with time-triggered variant of list scheduling.

Rèsumè

Les systèmes temps-réels modernes ont tendance à obtenir la criticité mixte, dans le sens où ils intègrent sur une même plateforme de calcul plusieurs applications avec différents niveaux de criticités. D'un côté, cette intégration permet de réduire le coût, le poids et la consommation d'énergie. Ces exigences sont importantes pour des systèmes modernes comme par exemple les drones (UAV). De l'autre, elle conduit à des complications majeures lors de leur conception. Ces systèmes doivent être certifiés en prenant en compte ces différents niveaux de criticités. L'ordonnancement temps réel des systèmes avec différents niveaux de criticités est connu comme étant lun des plus grand défi dans le domaine. Les techniques traditionnelles nécessitent une isolation complète entre les niveaux de criticité ou bien une certification globale au plus haut niveau. Une telle solution conduit à un gaspillage des ressources, et à la perte de lavantage de cette intégration. Ce problème a suscité une nouvelle vague de recherche dans la communauté du temps réel, et de nombreuses solutions ont été proposées. Parmi elles, l'une des méthodes la plus utilisée pour ordonnancer de tels systèmes est celle d'Audsley. Malheureusement, elle a un certain nombre de limitations, dont nous parlerons dans cette thèse. Ces limitations sont encore beaucoup plus accentuées dans le cas de l'ordonnancement multiprocesseur. Dans ce cas précis, l'ordonnancement basé sur la priorité perd des propriétés importantes. Cest la raison pour laquelle, les algorithmes d'ordonnancement avec différents niveaux de criticités pour des architectures multiprocesseurs ne sont que très peu étudiés et ceux quon trouve dans la littérature sont généralement construits sur des hypothèses restrictives. Cela est particulièrement problématique car les systèmes industriels temps réel cherchent à migrer vers plates-formes multi-cœurs. Dans ce travail nous proposons une approche différente pour résoudre ces problèmes.

Contents

Chapter 1

Introduction

1.1 Motivation

Safety-critical embedded systems are facing issues related to stringent non-functional requirements as cost, size, weight, heat and power consumption. This caused two leading trends in the design of such systems.

First, modern systems tend to be "mixed-critical", in the sense that they integrate components that must guarantee correctness under different levels of assurance. For example avionic standard DO-178B defines five levels of criticality, called Design Assurance Levels (DALs), from DAL-A to DAL-E. The classification is based on the potential effect that a failure of a component may have. For example a failure of a DAL-A component may have catastrophic effects (loss of human lives), while a failure of a DAL-E component has no effect at all on safety. Several other standards are used in industry, like for example, the IEC 61508, DO-254 and ISO 26262 standards. Traditional safety-critical systems tend to isolate the levels of criticality, to avoid that a failure of a least critical component may propagate to a high critical one. By letting components at different levels of criticality execute on the same platform, mixed criticality systems introduce several design challenges. Barhorst *et al.* give a thorough discussion of the mixed-criticality problem in [BBB+09], they identify the main challenges of the problem and identify mixed-critical systems as a distinct and new area of research. One of the main discussed topics on mixed-criticality systems is scheduling, since it was shown that classical techniques are not applicable in this case. The latter topic generated a new wave of research in the scientific community, that produced hundreds of paper in less than a decade [BD13].

The other leading trend in safety-critical system design is the migration from single- to multi- and many-core platforms. High performance single-core processors suffer of the well known problems of power consumption and overheating. In addition, advanced single-core machines, the so-called *superscalar* processors, are designed to optimize the average performance, not the worst case. They, in fact, have problematic architectural features, like pipelining flush, that can increase the worst-case execution time and complicate its computation. Thus they cannot be used in time-critical applications. Multi-cores, on the other hand, may increase performances by increasing the number of cores, thus keeping the cores unit simple and heating and power consumption under acceptable levels. However there are several challenges to be solved in multi-core platforms, related to the usage of shared resources between cores. The problem is still subject of research, and it is suggested ([Eur11, Fed15]) to not use processors with more than two cores in avionic systems.

Level	Effect	FR
A	Catastrophic	$10^{-9}/h$
B	Hazardous	$10^{-7}/h$
C	Major	$10^{-5}/h$
D	Minor	$10^{-3}/h$
E	No Effect	n/a

Table 1.1: Design Assurance Levels in DO178b

Multi-core mixed-criticality systems are a promising evolution of safety-critical systems. However there are a lot of unsolved issues and there is a strong need to provide theory, tools and techniques for the design of such systems. This thesis is an effort to dig a way towards solid design techniques for mixed-critical multi-core systems that can yield efficient resource usage and guarantee timing constraints to be met. Our main focus is on scheduling, since this is probably the most challenging problem in mixed-critical system design, but we also propose a model of computation and a design flow for mixed-critical systems design. This chapter gives an overview of the problems of mixed-criticality multi-core systems, and it is organized as follows. Section 1.2 explains the challenges of multi-core mixed-criticality systems design. Then an overview of the state of the art is given in Section 1.3. Finally Section 1.4 closes the chapter by summarizing the thesis' contributions.

1.2 Challenges

Avionic standard DO-178B defines five levels of criticality, called Design Assurance Levels (DALs). Each DAL is labeled with a letter, as shown in Table 1.1. The classification is based on the effect that a failure of a component may have, as shown in the second column of Table 1.1. Also each criticality levels has to satisfy several *objectives*, defined by the standard. Among those, there is the need to bound a maximum *Failure Rate* (FR). This is measured in maximum number of failures per hour, and it is reported on the third column of Table 1.1.

1.2.1 Scheduling

Scheduling is one of the most challenging problems in the context of mixed critical systems. One of the first works on MCS scheduling was done by Vestal [Ves07]. In his seminal paper a formalization of the problem was introduced. This formalization is used in most of the works proposed in literature and it is known as the "Vestal Model". The majority of the papers proposed in literature adopt the Vestal model or some variations and/or extensions, with few exception. A formal definition of this model is presented in Chapter 2.

Each criticality level needs to guarantee the correctness at different level of assurance. This means that, in Vestal model, that for each criticality level in the system we compute a *Worst Case Execution Time* (WCET) estimation that is safe according to the respective level of assurance. Worst Case Execution Times (WCETs) are hard to compute. The number of possible execution paths of a computer program grows exponentially with the complexity of the code. Analyzing all the possible executions of a piece of code is computationally unfeasible even for relatively simple programs. The analysis becomes even more complex on modern architecture, since the use of two or three level caches and complex pipelining

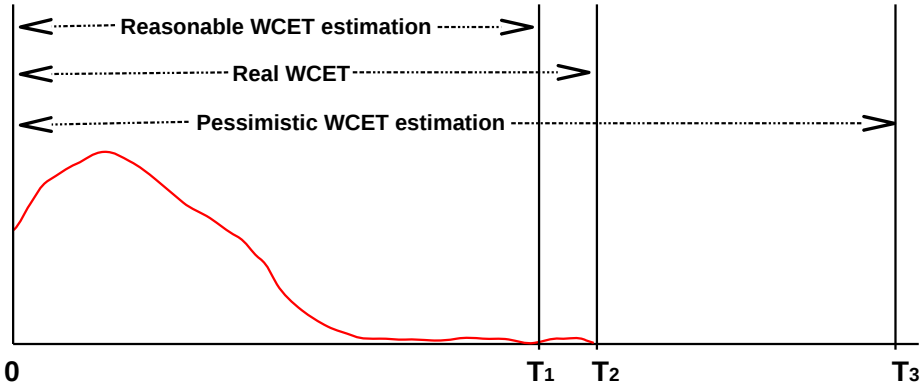


Figure 1.1: WCET and its estimations

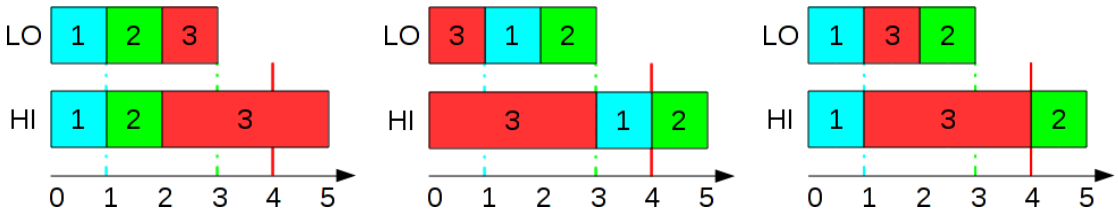
requires that phenomena like cache miss and pipeline flush needs to be taken into account. In addition, when migrating to multi- and many-cores one has to consider also job interference due to access to shared resources.

This issue has been analyzed for decades by the scientific community, and a deep dissertation on this topic is beyond the scope of this thesis. It is, however, necessary to be aware of the fact that there are two big categories of approaches to the WCET computation. The first one considers computing the WCET by extensive measurements or by analyzing only paths of the code that are more likely to be executed (probabilistic worst-case execution time analysis) or a combination of both. What this approaches have in common is that they are usually very restrictive or not *safe*, in the sense that the WCET computed using those techniques is not necessarily an over-estimation of the real WCET. The second group of techniques considers reducing the problem size by using pessimistic simplifications. This allows to compute safe estimation of the WCET, but it usually happens that such estimations are too pessimistic,*i.e.,* the estimated value is much bigger than the real one. This, of course, causes the system to be over-sized, with a consequential waste of computational resources.

This problem is depicted in Figure 1.1. Here we have an hypothetical probability density function of the termination time of an imaginary piece of code. It is clear from the picture that the real worst case execution time, is T_2. Using a non-safe WCET computation technique, we can end up by computing a WCET of T_1. If we design our system based on this estimation, we may observe a timing failure from time to time. A hypothetical safe tool would compute the value T_3, which is safe, but far from the real value. Designing a system based on such values will ensure that there will be no timing failures, but the system will be over-sized.

Vestal proposed to solve this issue by labeling each task with 5 different worst case execution time estimations, one for each criticality level. The idea is that each estimation is safe with a confidence level that corresponds to the maximum failure rate allowed by the standard for that particular criticality level (see Table 1.1). This allows at the same time, a good resource utilization and the safety of the highly critical levels.

However, Baruah [Bar09] showed that the problem of scheduling mixed criticality systems is highly intractable, in the sense that it is NP-hard in the strong sense to determine whether it is possible to successfully schedule a given system specified in the Vestal model upon a fully preemptive uniprocessor platform. To give an intuitive idea of why this problem is hard to solve, we give a problem instance example.



(a) EDF (b) Criticality Monotonic (c) Correct priorities

Figure 1.2: Different scheduling solutions for the instance of Example 1.2.1

Example 1.2.1. *Consider the problem of scheduling three jobs, that we will indicate as job* 1*, job 2 and job 3. Consider two criticality levels. Under normal circumstances (the non-critical scenario) , all jobs have to execute for 1 time unit, and their respective deadlines are at time 1, 3 and 4. In the case there is a fault in the system (critical scenario), only job 3 has to meet its deadline, but it will execute for 3 time units in this case.*

If we apply the well-known Earliest Deadline First (EDF) scheduling, all jobs will meet their timing constraints, as shown in the upper chart of Figure 1.2(a). However, in that case, job 3 will miss its deadline if it has to execute for three time units.

In the case where we apply a criticality monotonic strategy, i.e., *we give higher priority to the critical task, job 3 will meet its deadline in the critical scenario, but job 1 will miss its deadline in the non critical one, as shown in Figure 1.2(b).*

A solution that satisfies all timing requirements schedules the jobs in the following order: 1, 3, 2. As shown in Figure 1.2(c), this allows all jobs to meet their deadline in the non-critical scenario, and job 3 to meet its deadline in the critical one.

From the above example it is easy to see that meeting the constraints of the mixed critical scheduling in the non-critical scenario and in the critical one individually are solvable using classical techniques. However, meeting at the same time constraints from the critical and non-critical scenario is not trivial.

Scheduling mixed-critical system on multiprocessor platforms adds further complexity to the problem. One of the most successful techniques in mixed-critical scheduling is the so-called Audsley approach [Aud93], which proved to be very effective on single processors. This approach selects the priority of jobs by starting from the lowest priority one. It is based on the assumption that the termination time of a job *does not depend on the relative priority of higher priority jobs.* Thus, the termination time of the least priority job can be computed exactly before making any other priority assignment. However the base assumption of Audsley approach does not hold on multiprocessor platforms, as shown in the following:

Example 1.2.2. *Consider a set of four jobs, labeled with the firsts four integer numbers. Among those, job 1, job 3 and job 4 execute for 1 time unit, while job 2 executes for 2 time units. Scheduling them with the priority order* $1-2-3-4$ *will cause job 4 to terminate after 3 time units, as shown in the Gantt chart on the left of Figure 1.3. Changing the priority order* $1-3-2-4$ *will cause job 4 to terminate after 2 time units, as shown in the Gantt chart on the right of Figure 1.3. Thus the termination time of the job with the lowest priority (job 4) depends on the relative priority of job 2 and 3.*

In Section 3.1.1 we give a more formal definition of Audsley approach and we discuss its limitations.

Figure 1.3: The Gantt Chart of Example 1.2.2

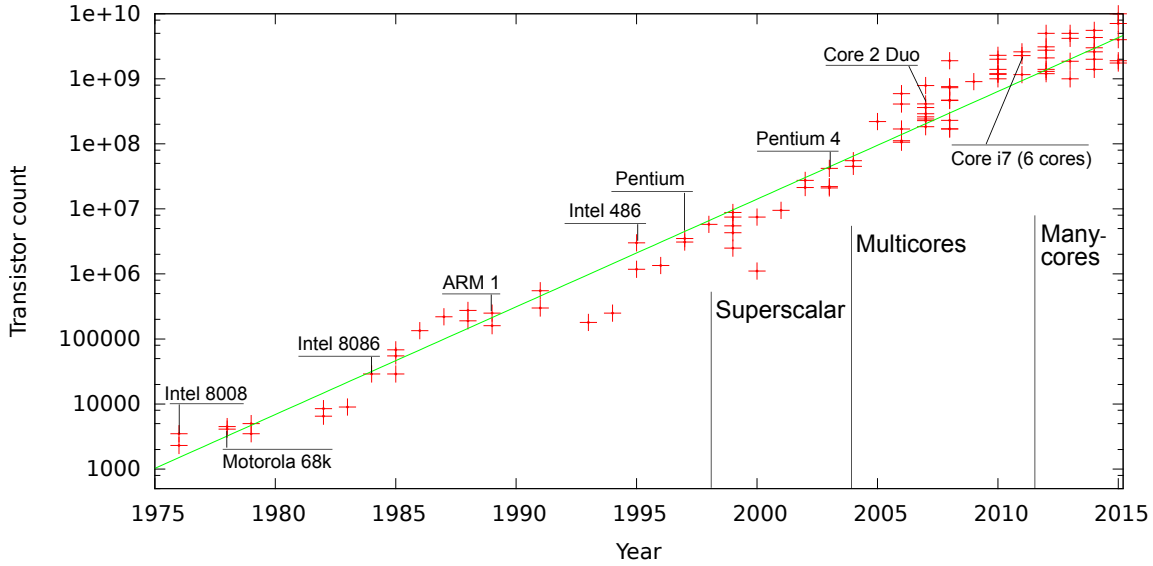


Figure 1.4: Moore's Law

1.2.2 Multi-core

In 1965 Gordon Moore [Moo65] formulated the well-know empiric law on the size of integrated circuits, known as *Moore's Law*. This empirical law states that *the number of transistors that can fit inside a chip doubles every 18 months*. Moore's law correctly estimated the evolution of integrated circuits as shown in Figure 1.4.

Dennard *et al.* [DRBL74] described the evolution of digital circuits using the so-called *Dennard Scaling*. The power consumed by an integrated circuit is given by:

$$P = NfCV^2 + VI_l \tag{1.1}$$

where N is the number of transistors in the chip, f is the clock frequency, C is the effective capacitance, V the voltage and I_l is the leak current. At the time Dennard *et al.* wrote their law, the leak current was small compared to the other quantity in the equation, so the latter term could be neglected. According to Moore Law, each 18 months the area of transistors reduces of a factor 2, *i.e.,* the linear dimension of transistor l reduces of a factor $\sqrt{2}$. The quantities in equation (1.1) depend on the linear dimension l according to the following relations:

$$N \propto l^2 \qquad C \propto 1/l \qquad V \propto 1/l \tag{1.2}$$

It follows from (1.1) and (1.2) that each time l is reduced by a scaling factor of $\sqrt{2}$, it is possible to double the number of transistor and increment the frequency of the circuit by a factor $\sqrt{2}$ while keeping the power consumption and the area of the chip constants.

Dennard scaling described accurately the evolution of processors for more than four decades, in which processors with higher frequency and more complex architecture were produced at each technological step. This law, however, no longer holds on modern systems. The main problem is that the second term on the right hand side of equation (1.1) is no longer neglectable, due to the high level of miniaturization. This causes higher power consumption with consequential overheating. Another problems is that, together with the number of components, the complexity of the processors' architecture grew to the point that it increased the cost of design and caused some processors to be affected by hardware bugs [Col97]. The latter issue is particularly problematic for timing-critical systems. Firstly, because complex architectures make the worst case execution time analysis time harder, and secondly, because a possible hardware bug may cause a system failure.

Multi-core architecture were proposed to solve the above-discussed problems. Putting more cores into a single chip, in fact, allows to:

1. Improving the computational power by exploiting the parallelism and not by increasing the clock frequency, thus preventing the power consumption to grow and limiting overheating.

2. Using multiple simpler cores, instead of a single complex one, which, in turn, allows to to avoid design errors.

Migration to multi-cores, and subsequently to many-cores, is the only way we can implement computationally complex systems. This process is, however, not easy. Concurrent access to shared resources complicates the analysis of the system and introduces *non-determinism*. This is particularly problematic for timing critical architectures, since modern platforms are designed for average performances, not for worst case. For instance, one of the most problematic issue to solve is the presence in such architectures of two- or even three-level caches, that at the same time decrease the worst case execution time and make its computation harder, making such architectures hardly *predictable*. The solution would be to use *timing predictable architectures*, with simple pipelines and scratchpads memories instead of caches. Even if such architecture are studied in academy [Sch09], using them in real system is not nowadays possible. The reason is that microelectronic industry is characterized by high fixed costs and low variable costs, making the price of a chip strongly dependent on the number of units sold. Therefore if such a timing-predictable chip were produced, its cost would be much higher than the one of its general purpose competitor, due to its relatively narrow applicability, thus making the latter a preferable choice for economical reasons.

To guarantee *predictability* and *determinism* is the big challenge of multi- and many-core time-critical system design.

1.3 State of the Art

In this section we will give a quick overview of the state of the art. This dissertation is not meant to be complete or detailed, our objective is to give an idea of the directions that are followed by the scientific community in the field of mixed-critical systems. The works of particular relevance with respect to this thesis are discussed at a higher level of detail in the related work section of the proper chapter.

1.3.1 Scheduling

One of the first paper on mixed critical scheduling was written by Vestal [Ves07]. In this work Vestal model was introduced, thus pointing the way to the following research. Even if the approach proposed by Vestal is the most used, different models exists, as explained in [GB13].

Baruah and Vestal give schedulability analysis in [BV08]. In this work Vestal model is generalized for sporadic tasks. This paper also shows that Earliest Deadline First (EDF) is not optimal,*e.g.,* it does not dominate fixed priority policy when criticality levels are introduced, and that there are feasible systems that cannot be scheduled by EDF.

Job Scheduling

The simplest model for scheduling is the job model. This model only considers a finite set of tasks instances (jobs). The main drawback of this approach is that to be applied to periodic tasks systems one has to impose synchronous tasks and to consider all the jobs present in a hyperperiod, *i.e.,* the least common multiplier of all the periods, which can grow exponentially. On the other hand, this model is simpler, thus allowing easier analysis and better performances.

One of the main results for this model is an algorithm called *Own Criticality-Based Priorities* (OCBP) [BLS10], proposed by Baruah *et al.* in 2010. In [LB10b] they also introduce load based schedulability analysis for OCBP in two criticality systems. This analysis is based into two distinct load values, LO and HI, one per criticality. This metric has been widely used and modified by other works. In [LB10a] Li and Baruah extends OCBP to sporadic systems, by computing priorities at run-time. The latter work has been improved by Gu *et al.* [GGDY13], by reducing the complexity of the run-time scheduler from quadratic to linear.

Baruah and Guo [BZ13] consider the problem of scheduling finite sets of jobs upon unreliable processors. They propose a modified version of Vestal model where the processor can run at two different speeds: *normal* and *degraded*. When the processor switches to degraded speed, only high-critical jobs are obliged to terminate before the deadline.

Task Model

Task model is more versatile than the previously discussed job model. One of the most popular techniques used in task scheduling is *Response Time Analysis* (RTA). Baruah *et al.* [BBD11] propose an algorithm that maximizes the priority of high critical tasks in an optimal way. A RTA is provided to ensure schedulability. Several extensions and improvement [ZGZ13, BD$^+$14, Fle13, HGL14] have been done starting from the above-discussed paper of Baruah *et al.* [BBD11].

Baruah and Chattopadhyay [BC13] propose an RTA based algorithm using an alternative model where high critical tasks execute with shortest period, instead of longest execution time, in case of a mode-switch.

An alternative approach to RTA is slack scheduling. This kind of scheduling is based on the idea to let the low-critical tasks run in the slack generated by high-critical one. See for example the work of De Niz and Phan [dNP14].

Multiprocessor Scheduling

One of the first work is made by Mollison *et al.* [MEA+10]. This work is mainly based on temporal isolation techniques, which provide good isolation between criticality levels, but worse performances if compared to solutions that allows jobs at different criticality to run at the same time. Herman *et al.* [HKM+12] extend this approach to take OS overhead in account. Li and Baruah [LB12] proposed a global multiprocessor algorithm, fpEDF, with a complete theoretical analysis. However in a later work [BCLS14] they also propose a partitioned scheduling, and they show that this second solution gives better performances. Other partitioned solutions can be found in [LDNRM10, LDNRM10, KAZ11, GRP14].

Baruah [Bar13], propose a multiprocessor list scheduling algorithm for synchronous reactive systems. This papers introduces the problems of job dependencies, however, it is restricted to jobs that all have the same arrival time and deadline.

Time Triggered

Time-triggered scheduling, being static, allows simpler analysis, and thus is popular in time-critical applications. Baruah and Fohler [BF11] extend the time-triggered scheduling policy, to mixed-critical systems by considering one scheduling table per criticality mode. In the work of Baruah [Bar13] the STTM scheduling was extended from single to multiple processors, in the context of synchronous reactive systems.

Theis *et al.* [TFB13] build such tables via search tree, but their approach does not allow high utilization. Jan *et al.* [JZLP14] propose a linear programming based solution.

Probabilistic Scheduling

An interesting area of research is the one of probabilistic scheduling. In this theory WCET are modeled as probability density functions (pWCET). Work on this directions were proposed by Alahmad *et al.* [AGSCG11] and Guo *et al.* [GSY15].

1.3.2 Formal Languages

Only a few work on formal languages have proposed for mixed critical systems. The main problems of formal languages is the state-space explosion problem, which is particularly problematic for mixed-critical systems, since they allow different modes of executions.

Amey *et al.* [ACW05] proposed static code analysis techniques to guarantee isolation in mixed criticality software written in SPARK [Bar03], a safety-critical dialect of Ada language.

Lindgren *et al.* in [LEL+14] discuss an approach based on static semantic analysis performed directly on the system specification in the experimental language RTFM-lang. This allows at compile time to discriminate in between critical and non-critical functions, and assign these appropriate access rights.

1.3.3 Managing Accesses to Shared Resources

Sharing resources within criticality levels is one of the key issues of mixed-critical systems. Burns [Bur13] extends to mixed-criticality system the classical priority ceiling protocol [SRL90] by adding criticality-specific blocking terms into the response time-analysis. This allows lower criticality tasks to transfer budgets of the resource usage to higher criticality ones.

Zhao *et al.* [ZGZ13] extend the Stack Resource Protocol (SRP) to Mixed-Criticality Systems and provide schedulability analysis.

Brandenburg [Bra14] proposes MC-IPC protocol, that enables temporal and logical isolation among tasks of different criticality.

1.4 Contributions

In this thesis we mainly address the problem of scheduling finite job sets in a dual criticality systems. Considering only two levels of criticality is a simplification often used in literature, to limit the complexity of the problem. Such systems are, however, of practical interest, since there are standards in domain like *Unmanned Air Vehicles* (UAVs) that define two levels of applications: *safety critical* and *mission critical*. Considering a finite set of job is also a simplification of the problem *w.r.t.* the more general task model. Job model allows potentially better processor utilization and simplifies the analysis of the system from a theoretical point of view. Its main drawback is that one has to consider an entire hyperperiod, *i.e.,* the least common multiplier of all the jobs' periods, to check the schedulability of such systems. This can make the analysis computationally complex. However, we are interested in static scheduling, in order to simplify the schedulability analysis. In static scheduling the problem of analyzing the whole hyperperiod is unavoidable. Also in a real-life system, one would expect the periods to have big common divisors, that would prevent the hyperperiod to grow too much.

In Chapter 2 we present the scheduling model and and characterization of problem instances. The model used is the so-called Vestal model [Ves07] and we extend it to take into account data dependencies between jobs. We also extend the well-known load characterization in one that is more suitable to data dependencies and to better account for multiprocessors, the latter variant of load extension is referred to as *Stress*.

Our contributions in priority based scheduling are presented in Chapter 3. Here we discuss the limitations of Audsley approach. To overcome those limitations we propose a theoretical tool, the *Priority Direct Acyclic Graphs* (P-DAGs) that models how the jobs interact with each other. Based on that, we propose two scheduling algorithms. The first is *Mixed Criticality Earliest Deadline First* (MCEDF), a priority based algorithm for single processor. We formally prove that MCEDF dominates the state of the art algorithm OCBP despite the latter being an optimal fixed-priority algorithm. The second algorithm is *Mixed Criticality Priority Improvement* (MCPI). This algorithm can be applied to multiprocessor instances with dependency constraints. To the best of our knowledge no other algorithm to solve this kind of problem has been proposed in literature, if not under quite restrictive constraints. We show equivalence and some optimality properties of both algorithms and conclude the chapter by showing experiment that confirms the good performances of MCEDF and MCPI.

In Chapter 4 we address the problem of static scheduling. We propose an algorithm that can generate static scheduling tables from a non-static scheduling policy (for example a priority based one). Our algorithm is optimal for single processor case, in the sense that we successfully find a feasible scheduling table if and only if the original policy generates a feasible schedule as well. Hence we show that a static schedule can be used to check correctness of non-static policies, which can potentially be done at less computational costs. Experiments provided at the end of the chapter show satisfying results for the multiprocessor case.

Finally in Chapter 5 we propose a formal method based design flow for mixed critical systems. It is based on a novel Model of Computation (MoC), *Fixed Priority Process Network* (FPPN), that shows good potential to prove suitable for describing both reactive-control and data-flow applications and has the advantage of generating deterministic executions. This model has relation to synchronous languages, and the latter have been studied in the related work as potential candidates to represent tasks communicating via buffers deterministically. In our flow both the MoC and the scheduling policy are described in the timed-automata based language BIP (Behaviour Interaction Priorities). The BIP code is generated automatically from high level description of MoC and policy.. The effectiveness of the approach is shown using a real-life example from avionic industry, a *Flight Management System*.

Chapter 2

Scheduling Model

In this thesis we will take into consideration a model of scheduling that is known as *Vestal Model* [Ves07]. We consider *dual-criticality* systems, having two levels of criticality, the high level, denoted as 'HI', and the low (normal) level, denoted as 'LO'. Every highly critical job gets a pair of WCET values: the LO WCET and the HI WCET. The former one is for normal safety assurance, used to assess the sharing of processor with the LO jobs, and the other one, a higher value, is used to ensure certification. Dual-criticality systems are of practical interest, since certain safety-critical application domains can be classified as such. For instance in the *unmanned aerials vehicle* (UAV's) domain, functionalities are divided into *mission-critical* functionalities and *flight-critical* functionalities, and only the latter undergo certification [BLS10]. One important remark is that both HI and LO jobs are hard real-time, so both *must* complete their executions before the deadlines. But only HI jobs undergo certification. This means that the designer is confident that the jobs will never exceed their LO WCET. However, it is required to prove to the certification authorities that the HI jobs will meet the deadlines even under the unlikely event that some jobs would execute at their HI WCET, calculated by very pessimistic certification tools. This necessity thus comes from certification needs (*i.e.,* legal constraints) and not from engineering considerations. For this reason, upon the hypothetical event in which some jobs violate their LO WCET, the scheduling policy tolerates that the LO jobs may miss their deadlines or even drops them altogether, in order to certify the HI jobs while requiring as little as possible processor resources. This approach for mixed-critical RT systems is called *certification-cognizant* [BBD+12b, BF11]. Note that other approaches exist such as *asymmetric isolation*, which instead of specifying different WCETs, ensures that the highly critical jobs get a higher protection from missing the deadline when a job violates its normal WCET [NLR09], but they are beyond the scope of this thesis.

2.1 Independent Jobs

The use of computing systems in life-critical applications such as avionics or automotive usually requires very high reliability and responsiveness. Most tasks in these applications have timing constraints, *i.e.,* deadlines, to be satisfied. Generally, real-time tasks are categorized as *periodic* and *aperiodic*. A single instance of a task execution is called job. Periodic tasks are activated repeatedly in a fixed time interval, called period. Such tasks are usually used to poll sensors and for control-loop subroutines. The activation of aperiodic tasks can occur at any time and it is usually triggered by special conditions or operator command. Since, by default, the number of jobs that can be generated by aperiodic tasks is unbounded, it

is impossible to guarantee the schedulability of such tasks. For this reason such tasks are usually refined to the *sporadic* task model. Similarly to periodic ones, sporadic tasks can be triggered at any time, but once a job is triggered a fixed length minimum inter-arrival time must pass before a new execution of the task may be triggered. A more general definition of sporadic tasks defines a maximum number of jobs activation in any time interval of given length.

However in this thesis we will mainly focus on finite set of jobs. This is motivated by the fact that first, Mixed Critical Scheduling is a new problem in research, and thus even under this simplifying assumption there there is fertile ground for new research. Moreover, in the following chapters we will target time triggered scheduling, which is by its nature easily modeled with a finite set of jobs. Finally a set of periodic tasks can be easily modeled as a finite set of jobs, considering only the jobs appearing in one hyperperiod, *i.e.,* the least common multiple of the tasks' period. This can, with limitations, be extended to sporadic tasks, as shown in Chapter 5.

2.1.1 Problem Definition

In this section we introduce a formalization of the Vestal model for Mixed-Critical System (*MCS*) for the dual-criticality case. In Vestal model, 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.

The index j is technically necessary to distinguish between the jobs with the same parameters. We assume that [BBD+12b]:

$$C_j(\text{LO}) \leq C_j(\text{HI})$$

The latter makes sense, since $C_j(\text{HI})$ is a more pessimistic estimation of the WCET than $C_j(\text{LO})$. We also assume that the LO jobs are forced to complete after $C_j(\text{LO})$ time units of execution, so:

$$(\chi_j = \text{LO}) \Rightarrow C_j(\text{LO}) = C_j(\text{HI})$$

The interval $[A_j, D_j]$ is the *time window* of job J_j.

An *instance* of the scheduling problem is a set of jobs $\mathbf{J}$. A *scenario* of an instance $\mathbf{J}$ is a vector of execution times of all jobs: $(c_1, c_2, \ldots, c_K)$. If at least one c_j exceeds $C_j(\text{HI})$, the scenario is called *erroneous*. The *criticality of scenario* $(c_1, c_2, \ldots, c_K)$ is LO if $c_j \leq C_j(\text{LO}), \quad \forall j \in [1, K]$, is HI otherwise. A scenario is *basic* if:

$$\forall j = 1, \ldots, K \quad c_j = C_j(\text{LO}) \vee c_j = C_j(\text{HI})$$

A *schedule* $\mathcal{S}$ of a given scenario is the mapping:

$$\mathcal{S} : T \mapsto \mathbf{J}_\epsilon \times \mathbf{J}_\epsilon \times \ldots \times \mathbf{J}_\epsilon = \mathbf{J}_\epsilon^m$$

where T is the physical time and $\mathbf{J}_\epsilon = \mathbf{J} \cup \{\epsilon\}$, where ϵ denotes no job and m the number of processors available. Every job should start at time A_j or later and run for no more than

c_j time units. A schedule is *preemptive* if a job run can be interrupted and resumed later. A job may be assigned to only one processor at time t, but *migration* from one processor to another may be possible.

A job J is said to be *ready* at time t iff:

1. it is already arrived at time t
2. it is not yet completed at time t

The online state of a run-time scheduler at every time instance consists of the set of completed jobs, the set of *ready jobs*, the progress of ready jobs, *i.e.,* for how much each of them has executed so far, and the current *criticality mode*, χ_{mode}, initialized as $\chi_{mode} = \text{LO}$ and switched to 'HI' as soon as a HI job exceeds $C_j(\text{LO})$. A schedule is *feasible* if the following conditions are met:

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

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

An instance **J** is *clairvoyantly schedulable* if for each non-erroneous scenario, when it is known in advance (hence *clairvoyantly*), one can specify a feasible schedule. This property is of purely theoretical interest, as in reality the execution time of every job J_j is only discovered when J_j signals its termination. Hence, whether the LO job time termination is required is not known as long as no HI job has shown an execution time exceeding its $C_j(\text{LO})$.

Based on the online state, a *scheduling policy* deterministically decides which ready jobs are scheduled at every time instant on m processors. A scheduling policy is *correct* for the instance **J** if for each non-erroneous scenario it generates a feasible schedule. A *mode-switched* scheduling policy uses χ_{mode} in the scheduling decisions, *e.g.,* to drop the LO jobs, otherwise it is *mode-ignorant*. A policy is said to be *work-conserving* if it never idles the processor if there is pending workload.

An instance **J** is *MC-schedulable* if there exists a correct scheduling policy for it.

2.1.2 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 a change of the mode in which job 'B' terminates, for example 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 it suffices 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 $\mathcal{S}^{LO}$ 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 until their maximal execution time – the HI WCET. Note that in multiprocessor scheduling multiple jobs may also terminate *exactly* at time t_h in $\mathcal{S}^{LO}$, and they are conservatively 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 2.1.1 (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 $\mathcal{S}^{HI\text{-}J_h}$.*

Note that $\mathcal{S}^{HI\text{-}J_h}$ coincides with $\mathcal{S}^{LO}$ 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.

Theorem 2.1.2 (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 [BBD+12b].

2.1.3 Fixed Priority and Fixed Priority per Mode

A *fixed-priority* scheduling policy is a mode-ignorant policy that can be defined by a priority table PT, which is a K-sized vector specifying all jobs (or, optionally, their indexes) in a certain order. The position of a job in PT is its *priority*, the earlier a job is to occur in PT the higher the priority it has. Among all ready jobs, the fixed-priority scheduling policy always schedules the m highest-priority jobs in PT. Fixed priority is a *work-conserving* policy. 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 it is clear from the context to which priority table we are referring to.

We introduce *fixed priority per mode* (FPM), a natural extension of Fixed-priority for MCS. FPM is mode-switched policy with two tables: PT_{LO} and PT_{HI}. The former includes all jobs. The latter only HI jobs. As long as the current criticality mode χ_{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}.

The following [HL94] states a very useful property, for which we formulate a corollary:

Lemma 2.1.3. *Fixed-priority policy (without precedences), is predictable.*

Corollary 2.1.4. *FPM is predictable per-mode.*

If a scheduling policy cannot be defined by a static priority table, it is called *dynamic*. Dynamics policy are, in general, more powerful then fixed priorities, but they are generally more complex and they usually require heavier run-time computation. Also they are not necessarily predictable. Fixed-priority scheduling are very popular thanks to the fact that they are predictable and easier to implement. Also, they are natively supported by many operating systems and libraries for programming real-time systems.

2.1.4 Characterization of Problem Instance

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

To characterize the performance of scheduling algorithms one uses the utilization and the related demand-capacity ratio metrics, *i.e.,* the maximal ratio between demand and capacity of the system. For a job set $\mathbf{J} = \{J_i\}$ and an assignment of execution times c_i the appropriate metric is load [Liu00]:

$$\ell oad(\mathbf{J}, c) = \max_{0 \le t_1 < t_2} \frac{\sum\limits_{J_i \in \mathbf{J}:\ t_1 \le A_i \wedge D_i \le t_2} c_i}{t_2 - t_1}$$

For a multiprocessor system there does not exist a necessary and sufficient schedulability bound on load, whereas it exists for *uniprocessor systems*:

$$\ell oad \le 1$$

For m-processor system the corresponding bound is only *necessary, but not sufficient* [BF05]:

$$\ell oad \le m$$

Baruah *et al.* [LB10b] defined the load metrics for mixed-critical scheduling problems and applied them to analyze fixed-priority algorithms. The authors define the in LO and in HI mode as shown below:

$$Load_{\mathrm{LO}}(\mathbf{T}) = \max_{0 \le t_1 < t_2} \frac{\sum\limits_{J_i:\ t_1 \le A_i \wedge D_i \le t_2} C_i(\mathrm{LO})}{t_2 - t_1}$$

$$Load_{\mathrm{HI}}(\mathbf{T}) = \max_{0 \le t_1 < t_2} \frac{\sum\limits_{J_i:\ \chi_i = \mathrm{HI}\ \wedge\ t_1 \le A_i \wedge D_i \le t_2} C_i(\mathrm{HI})}{t_2 - t_1}$$

An instance can only be schedulable if the processor is not overloaded. Hence, a *necessary* condition for MC schedulability is [LB10b]:

$$Load_{\mathrm{LO}}(\mathbf{T}) \le m \wedge Load_{\mathrm{HI}}(\mathbf{T}) \le m \tag{2.1}$$

This is also a sufficient condition for clairvoyant scheduling on single processor, but not for practically realizable policies [LB10b].

The characterization above proved useful for the fixed-priority policy. However, we would like to stress that a shortcoming of $Load_{\text{LO}}$ and $Load_{\text{HI}}$ is that they ignore a phenomenon which we call the *WCET uncertainty*. This phenomenon makes a practically realizable policy inferior to a clairvoyant scheduler. The latter 'knows for certain' whether and when a mode switch will occur at runtime, whereas an ordinary policy is 'uncertain' about this. By definition, this knowledge can be exploited online only by mode-switched policies. The job WCET uncertainty can be measured as $\Delta C_j = C_j(\text{HI}) - C_j(\text{LO})$ (strictly positive only for the HI jobs). In [PK11] it is proposed to consider a new set of job deadlines for the LO scenario: $D'_j = D_j - \Delta C_j$. It was noticed in [PK11] that in the LO scenario the jobs should meet deadlines D'_j, otherwise deadlines D_j are missed in a HI scenario. A new metric, $Load_{\text{MIX}}$ is thus defined as the one equal to $Load_{\text{LO}}$ after substituting D'_j into D_j [PK11]:

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

By the reasoning above, the *necessary* condition (2.1) can be refined to the following:

Lemma 2.1.5 (Necessary condition for schedulability)**.** *Mixed-critical problem instance* $\mathbf{T}$ *is schedulable only if*

$$Load_{MIX}(\mathbf{T}) \le m \wedge Load_{HI}(\mathbf{T}) \le m \tag{2.2}$$

for all jobs it must hold:

$$A_i + C_i(LO) \le D'_i$$

in addition HI jobs must respect the following condition:

$$A_i + C_i(HI) \le D_i$$

$Load_{\text{MIX}}$ is a better indicator of schedulability than $Load_{\text{LO}}$. This is shown in Section 3.3.4, where we introduce *splitting*, a transformation that modifies an instance into another with equal $Load_{\text{LO}}$, but lower $Load_{\text{MIX}}$. The instance thus generated is more likely to be schedulable by FPM policies.

The *Load* metric is very powerful to profile single processor problems. On multiprocessor, however, its effectiveness reduces. This is due to the fact that in multiprocessor it is not always possible to use all the available resources for the available workload, due to the fact that we may not parallelize a single job's execution. For example, if we consider an instance composed of only one job J_1 such that $C_1 = D_1 - A_1$, then we will have $Load = 1$. If we decrease the speed of the processor by a factor ϵ it will not be possible to schedule this instance, not even increasing the number of processors. This shows the weakness of *Load* metric, since if we have, for example, 4 processors, the *Load* metric will tell us that we have only a little bit more then 1/4 of our resources busy. To compensate this issue, we introduce the *Stress* metric:

$$stress_{\text{LO}}(\mathbf{J}) = \max_{0 \le t_1 < t_2} \left\{ \frac{m}{\min\{|\mathbf{J}'|, m\}} \cdot \frac{\sum_{J_i \in \mathbf{J}'} C_i(\text{LO})}{t_2 - t_1} \right\}$$

where $\mathbf{J}' = \{J_i \mid t_1 \leq A_i \wedge D_i \leq t_2\}$

$$stress_{\text{HI}}(\mathbf{J}) = \max_{0 \leq t_1 < t_2} \left\{ \frac{m}{\min\{|\mathbf{J}'|, m\}} \cdot \frac{\sum_{J_i \in \mathbf{J}'} C_i(\text{HI})}{t_2 - t_1} \right\}$$

where $\mathbf{J}' = \{J_i \mid \chi_i = \text{HI} \ \wedge \ t_1 \leq A_i \wedge D_i \leq t_2\}$

$$stress_{\text{MIX}}(\mathbf{J}) = \max_{0 \leq t_1 < t_2} \left\{ \frac{m}{\min\{|\mathbf{J}'|, m\}} \cdot \frac{\sum_{J_i \in \mathbf{J}'} C_i(\text{LO})}{t_2 - t_1} \right\}$$

where $\mathbf{J}' = \{J_i \mid t_1 \leq A_i \wedge D_i' \leq t_2\}$.

The $m/|\min\{|\mathbf{J}'|, m\}|$ scale factor is used to consider the fact that if there are $j < m$ ready jobs then only j processors can be used to schedule them. In the example given above, in fact, we will have that $m/|\mathbf{J}'| = 4$, thus giving $stress > 4$, coherently with the non-schedulability of the instance.

In the context of Lemma 2.1.5, one can rewrite the necessary conditions (2.1) and (2.2) using stress, but this would not make the lemma stronger due to other conditions formulated there. Nevertheless in general, we have $stress \geq \ell oad$, therefore we use it as a more 'realistic' metric of 'complexity' of the scheduling problem, as for the problem instances of growing complexity it approaches the critical bound m faster than the load.

2.2 Precedence Constraints

2.2.1 Problem Definition

In this section we will extend the Vestal model for the case of precedence constrained jobs.

A *task graph* $\mathbf{T}$ of the MC-scheduling problem is a pair $(\mathbf{J}, \rightarrow)$ of a set $\mathbf{J}$ of K jobs with indexes $1 \ldots K$ and a functional precedence relation:

$$\rightarrow \subset \mathbf{J} \times \mathbf{J}$$

We use the notation $J_a \nrightarrow J_b$ to indicate that there is no precedence relation $J_a \rightarrow J_b$. The precedence relation is well defined iff:

$$J_a \rightarrow^* J_b \Rightarrow J_b \nrightarrow J_a$$

where $\rightarrow^*$ is the transitive closure of $\rightarrow$. The above condition imposes absence of cycles in the precedence relation.

The criticality of a precedence constraint $J_a \rightarrow J_b$ is HI if $\chi(a) = \chi(b) = \text{HI}$. It is LO otherwise. For each precedence constraint $J_a \rightarrow J_b$, job J_b may not run until J_a completes. Thus when scheduling a task graph a job is not *ready* until all its predecessors have signalled their termination.

The reader may have noticed that we implied the possibility of having precedence relation from LO jobs to HI jobs. This may be unusual and may be considered a bad practice.

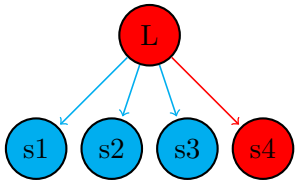


Figure 2.1: The graph of an airplane localization system illustrating LO→HI dependencies.

Nevertheless we allow this, because it is necessary in some practical situation, like the one of the of Figure 2.1. There we have a task graph of the localization system of an airplane, composed of four sensors (jobs s1-s4) and the job L, that computes the position. Data coming from sensor s4 is necessary and sufficient 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 switch to HI mode we only wait for data coming from s4.

In MCS, a schedule, to be feasible, needs to 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.*

2.2.2 Extending Fixed Priority to Precedence Constrains

We will now show how to generalize the fixed-priority and FPM policy for precedence constrained instances. In the previous section we redefined the concept of ready job, by adding the condition that all the predecessors must have completed. The use of fixed-priority in combination with the adopted precedence aware definition of ready job is called in literature *List Scheduling*. For a detailed implementation of list scheduling, see Appendix A. The generalization of fixed priority is then straightforward, but it is important to stress the following:

Lemma 2.2.1. *List Scheduling is* not *predictable on multiprocessor.*

This is shown in the following:

Example 2.2.2. *Consider a task-graph* $\mathbf{T} = (\mathbf{J}, \rightarrow)$ *where* $\mathbf{J}$ *is defined by the following table*

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

on which the followings precedence constraints are defined: $J_1 \rightarrow J_2$ *and* $J_1 \rightarrow J_3$*. Consider also the following priority table:*

$$PT = J_1 \succ J_2 \succ J_3 \succ J_4$$

Figure 2.2: The Gantt Chart of Example 2.2.2.

Then it is easy to see that the schedule generated by PT for the basic scenario on a system with 2 processors is the one shown in the Gantt chart on the left of Figure 2.2. In at time 0 *only* J_1 *and* J_4 *are ready* (J_2 *and* J_3 *are waiting* J_1 *to terminate). At time* 2, J_1 *and* J_4 *terminates and* J_2 *and* J_3 *become ready. They will execute for* 2 *time units and then they will terminate.*

Now consider the scenario where $c_1 = 1$. *This is shown in the right of Figure 2.2. In this case* J_1 *will terminate at time* 1, *thus making* J_2 *and* J_3 *ready before the termination of* J_4. *This will cause the preemption of* J_4, *that will be allowed to execute only when* J_2 *and* J_3 *terminate (time* 3*). This will cause* J_4 *to terminate at time* 4, 2 *times unit after its termination in the basic scenario. Notice that in this case* J_4 *also misses its deadline.*

Predictability is required to reason in terms of basic scenarios. Thus, when using list scheduling offline, some modifications to online scheduling policy that ensure schedulability must be applied. In Chapter 4 we show how to do this by using Time-Triggered paradigm.

Usually a priority table PT is required to be *precedence compliant*, *i.e.,* the following property must hold:

$$J \rightarrow J' \Rightarrow J \succ_{PT} J' \tag{2.3}$$

The above requirement is reasonable, since we may not schedule a job before its predecessors complete. It is indeed possible to schedule using non precedence compliant tables, since the online policy will just ignore high-priority jobs until all their low-priority predecessors will signals their termination, but as will be clear in the next chapters, precedence compliance can be useful in some cases.

In Figure 2.3 an algorithm that can transform a priority table to a precedence compliant one is shown. This algorithm preserves $J_1 \prec J_2$ unless $J_1 \rightarrow^* J_2$. The transformation is done as follows. We repeatedly scan the priority table, from the highest to the least priority. For each job J that has higher priority than some of its predecessors in $\mathbf{T}$, we raise the priority of those predecessors moving them immediately before J, keeping their relative order.

Example 2.2.3. *Consider the task graph* $\mathbf{T}$ *and the priority table shown in Figure 2.4. At the first iteration we will have* $J^{curr} = PT[1] = J_1$. *Since this job has no predecessors, no change in PT will be done inside the* while *loop, and thus we will increment* i. *At the second iteration we will have* $J^{curr} = PT[2] = J_2$. *At the first iteration of the* while *loop we will have that* $J_1 = J^{curr}$. J_1 *has no predecessors, so we will do no actions for it. At the following iteration* $J^{curr} = J_2$. *Since* $J_3 \rightarrow J_2$, *we will swap the position of* J_2 *and* J_3. J_2 *has no other predecessors, so we will not further modify PT. Since we did a modification of the table, we will not update* i, *thus at the third iteration we will have* $J^{curr} = PT[2] = J_3$. *At this iteration we will move back* J_5 *and* J_6, *as shown in Figure 2.4. Then until the end of the algorithm the table will not further modify table PT. It is trivial to check that the resulting table is indeed precedence compliant.*

Algorithm: *DependencyComplianceTransform*
Input: **task graph T**
Input: **priority table** PT
Output: **priority table** PT

```
1: i = 1
2: while i ≤ K do
3:     J^curr ← PT[i]
4:     for j = i + 1 to K do
5:         moved ← false
6:         J^p ← PT[j]
7:         if J^p → J^curr then
8:             MoveToFront(J^p, J^curr, PT)
9:             moved = true
10:        end if
11:    end for
12:    if moved = false then
13:        i ← i + 1
14:    end if
15: end while
```

Figure 2.3: The DependencyComplianceTransform algorithm

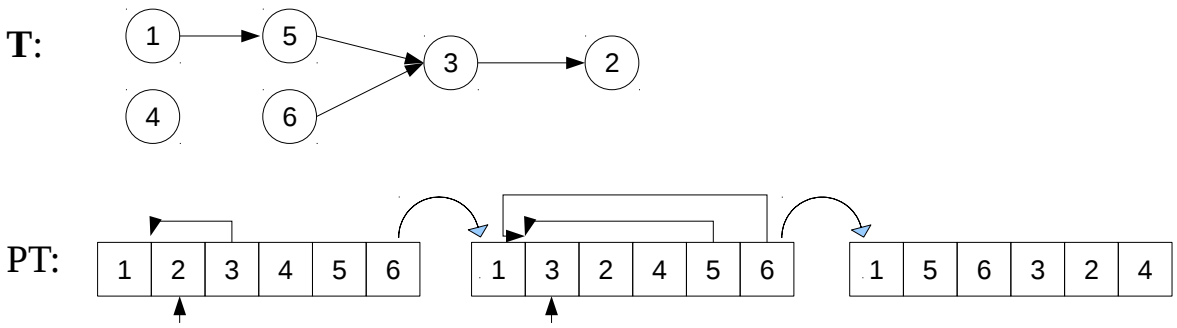


Figure 2.4: The initial PT transformation

2.2.3 Characterization of Problem Instances

In this work we will show how to adapt the metrics introduced in Section 2.1.4 when we deal with precedence constrained jobs.

First, we give some preliminaries. From the problem instance $\mathbf{T}(\mathbf{J}, \rightarrow)$ it is convenient to derive the following graphs:

HI-criticality graph $\mathbf{T}_{\text{HI}}(\mathbf{J}_{\text{HI}}, \rightarrow_{\text{HI}})$, where the $\mathbf{J}_{\text{HI}}$ is the subset of $\mathbf{J}$ that contains only HI jobs with HI WCET, and $\rightarrow_{\text{HI}}$ is the subset of $\rightarrow$ that contains only the precedences of HI criticality level.

LO-criticality graph $\mathbf{T}_{\text{LO}}(\mathbf{J}_{\text{LO}}, \rightarrow)$, where the jobs in $\mathbf{J}_{\text{LO}}$ are obtained from the original set of jobs $\mathbf{J}$ by considering LO WCET.

MIX-criticality graph $\mathbf{T}_{\text{MIX}}(\mathbf{J}_{\text{MIX}}, \rightarrow)$, where the jobs in $\mathbf{J}_{\text{MIX}}$ are obtained from the original set of jobs $\mathbf{J}$ by considering LO WCET and modifying job deadlines such that: $D'_i = D_i - (C_i(\text{HI}) - C_i(\text{LO}))$.

We define ASAP arrival times and ALAP deadlines, known in the task graph theory [KA99], but so far mainly used to derive priority tables rather than to compute the load[1].

For a task graph with execution times c, ASAP arrival time A^* is the earliest time when a job can possibly start:

$$A^*_j = \max_i \left(A_j, A^*_i + c_i \;\mid\; J_i \text{ are predecessors of } J_j\right)$$

Dually, ALAP deadline D^* is the latest time when a job is allowed to complete:

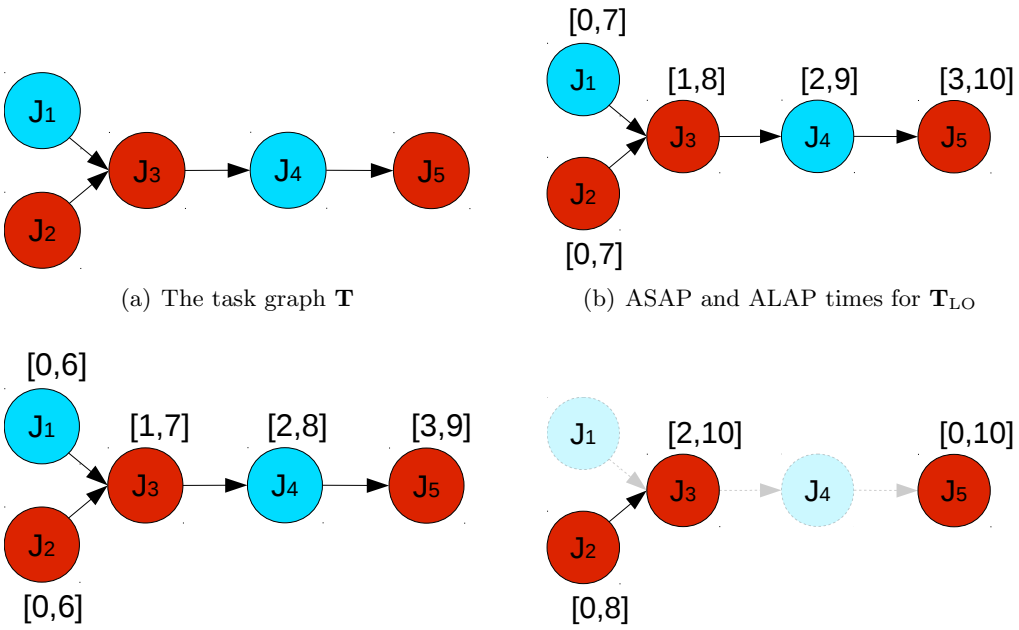
$$D^*_j = \min_i \left(D_j, D^*_i - c_i \;\mid\; J_i \text{ are successors of } J_j\right)$$

By analogy to static timing analysis in digital circuits, ASAP and ALAP values are obtained by *propagation* of the arrival and deadline times through the graph, *i.e.,* by longest-path algorithm that solves the equations above by visiting the nodes in topological or reverse topological order.

It is trivial that substituting ASAP arrival time and ALAP deadline to the job parameters does not change the schedulability of the task graph, so the necessary conditions in Lemma 2.1.5 remain valid, whereas the lemma becomes, in general, stronger. It should be noted that, by definition, to compute $Load_{\text{MIX}}$ one should do the ASAP/ALAP calculation in MIX-criticality graph $\mathbf{T}_{\text{MIX}}$ using $C(LO)$, whereas for $Load_{\text{HI}}$ it should be done in graph $\mathbf{T}_{\text{HI}}$ using $C(HI)$. These two graphs have different precedence constraints, and, by definition, we use the execution times c of two different modes for them: $C(LO)$ for MIX-graph and $C(HI)$ for HI-graph. Therefore ASAP arrival and ALAP deadlines for the same job are *mode-dependent*, and one should use these mode-dependent values also for the second part of Lemma 2.1.5, where we check the properties of individual jobs. ALAP and ASAP modified arrivals and deadlines give a better information on the amount of work present in the system.

An example of ASAP and ALAP times is given in Figure 2.5. Figure 2.5(a) shows the topology of the task graph. For this graph consider that all jobs have $A = 0$, $D = 10$,

[1] In literature the word ALAP is usually used for latest arrival



(a) The task graph $\mathbf{T}$

(b) ASAP and ALAP times for $\mathbf{T}_{\text{LO}}$

(c) ASAP and ALAP times for $\mathbf{T}_{\text{MIX}}$

(d) ASAP and ALAP times for $\mathbf{T}_{\text{HI}}$

Figure 2.5: Example of the various task graphs

$C(\text{LO}) = 1$. For the HI jobs (colored in red in the figures) we have $C(\text{HI}) = 2$ (and thus, $D' = D - (C(\text{HI}) - C(\text{LO})) = 9$). Figure 2.5(b) shows ASAP and ALAP times for graph $\mathbf{T}$. The numbers shown in the parenthesis give for each node, respectively, ASAP arrival time and ALAP deadline. In the same way, Figures 2.5(c) and 2.5(d) show, respectively, ASAP and ALAP times for $\mathbf{T}_{\text{MIX}}$ and $\mathbf{T}_{\text{HI}}$.

It has to be noticed that, ASAP and ALAP time are mode dependent, since they directly depend on mode-specific precedence constraints and execution times, as shown in the previous example.

In the sequel, unless mentioned otherwise, we assume in the algorithms and analysis that the load and stress values are computed using ASAP and ALAP values, using the respective mode.

Chapter 3

Priority Based Algorithms

In this chapter we will present some algorithms to compute priorities to schedule mixed criticality systems using fixed priority or fixed priority per mode paradigm. We will start in Section 3.2 by introducing the Priority-DAG. This is a theoretical tool that is useful to understand how jobs interact with each other, and on which all the algorithm presented in this chapter are based on. In Section 3.3 we will present MCEDF, a single processor algorithm for scheduling MCS. Later in Section 3.4, we will show MCPI, a multiprocessor algorithm that can be considered, in some extent, a generalization of MCEDF. Finally Section 3.6 will conclude this chapter by showing experimental evaluation of the scheduling algorithms.

3.1 Related Work

Although our scope is finite set of jobs, most of the literature concerns with instances that have an infinite set of jobs, generated by periodic or sporadic tasks. Periodic tasks are said to be synchronous if the offsets between the first arrival of different tasks are statically known. The deadlines can be implicit (*i.e.,* equal to the period), constrained (*i.e.,* less or equal to the period) or arbitrary (*i.e.,* larger than period).

Our work can be applied for scheduling the hyperperiod of *periodic synchronous non-pipelined*(*i.e.,* implicit or constrained-deadline) tasks with *precedence constraints*. However, we still consider general real-time policies, even if not originally designed for such systems, as they can be reused as *starting point for our priority-based algorithms*. We are particularly interested in the policies tailored for multiprocessor systems, assuming *global fixed priority* for jobs.

3.1.1 Audsley approach and its limitations

Most of the mixed-critical scheduling work present in literature use the so-called Audsley approach [Aud93]. The pseudocode of a generic implementation Audsley priority assignment technique is shown in Figure 3.1. At each step of the external loop the algorithm selects a Job to be put in the last position of priority table PT. The job is selected in the internal loop. For each job it computes the termination time in case the job is selected for the least priority, in case such a termination time is less then its deadline the job is selected and added in table PT. Then we remove the job from the current set of jobs $\mathbf{J}'$ and we run a new iteration. If no job can be selected the algorithm fails in finding a solution.

Audsley approach is based on the following assumptions:

Algorithm: *AudsleyPriorityAssignment*
Input: **job set J**
Output: **priority table** PT

```
1: J' ← J
2: PT ← ∅
3: while J' ≠ ∅ do
4:     i ← 1
5:     JobFound = false
6:     while i ≤ J'.size() ∨ ¬JobFound do
7:         TT ← GetTerminationTime(J', i)
8:         if TT ≤ J'[i].D then
9:             JobFound ← true
10:            PT ← J[i] ⌢ PT
11:            J' ← J' \ {J[i]}
12:        end if
13:    end while
14:    if ¬JobFound then
15:        return (FAIL)
16:    end if
17: end while
```

Figure 3.1: The Audsley algorithm

1. The termination time of any job does not depend on the jobs of lower priority
2. The termination time of any job does not depend on the relative priority of higher-priority jobs

If both assumptions are true, then Audsley approach is optimal. Unfortunately this is not always true in mixed-critical scheduling. Assumption 1 usually holds for preemptive scheduling, while cannot be guaranteed in the case of list scheduling of non-preemptive systems. Assumption 2 holds for single criticality scheduling on single processor. In [BLS10] it is shown that using the fixed priority policy this assumption holds on single processor, thus the proposed Audsley approach based algorithm OCBP is optimal. However, as previously discussed, the FP policy is too restrictive for MCS problems, where FPM is preferred. In the latter case, the assumption does not hold, since the order of execution may determine if a LO job is executed or not in case of a switch. Also Assumption 2 does not hold in multiprocessor systems.

Audsley approach can be used also in the case that one or both the assumptions do not hold. In that case the function *AudsleyPriorityAssignment* may not compute the exact termination time, and a safe worst case estimation must be made instead. In this case, however, the estimation is usually too pessimistic and/or computationally intractable. In the rest of this subsection we study how to safely estimate the termination time in the case of multiprocessor platforms. An example of such an estimation is present in the work of [Pat12], where a schedulability analysis is given for sporadic tasks set. Using a similar technique we derive a formula to estimate the termination time in the case of finite job sets and use it to implement the *GetTerminationTime* function of Figure 3.1.

Consider two jobs J_k and J_i such that $J_k \succ J_i$. The time interval in which J_i may

preempt J_k is given by:

$$\iota_{i,k} = [A_k, TT_k(\chi)] \cap [A_i, D_i]$$

where $TT_k(\chi)$ is a pessimistic estimation of the termination time of J_k in a scenario of criticality χ. Then the *interference* of J_i on J_k, *i.e.,* the cumulative length of the intervals in which J_i is executing and J_k is ready but not executing, is at most:

$$I_{i,k}(\chi) = \min\{|\iota_{i,k}|, C_i(\chi)\}$$

thus $TT_k(\chi)$ can be estimated by the following

$$TT_k(\chi) = A_k + C_k(\chi) + \left\lfloor \frac{\sum_{i \neq k} I_{i,k}(\chi)}{m} \right\rfloor \tag{3.1}$$

We realized the algorithm of Figure 3.1 implementing *GetTerminationTime* function using the fixed point of Equation (3.1) to estimate the termination time and we performed some experiments. We randomly generated 181 450 instances of 30 jobs, at different values of $Stress_{\text{LO}}$ and $Stress_{\text{HI}}$, similarly to Section 3.6.2. The values of *Stress* ranged uniformly from 0 to 2. We tried to schedule each instance first using FPM policy with EDF priority and then the above described implementation of Audsley approach based on Equation (3.1). Only 4.3% (7804) could be scheduled using Audsley approach, while 56.6% (102690) could be scheduled by EDF.

The weakness of this approach is shown in the following:

Example 3.1.1. *Consider the following instance* **J***:*

Job	*A*	*D*	χ	*C(LO)*	*C(HI)*
1	0	8	*LO*	3	3
2	0	8	*LO*	5	5
3	2	10	*LO*	5	5

It is easy to check that the followings are fixed points for equation (3.1) in the case $m = 2$*:*

$$TT_1(LO) = 9, \quad TT_2(LO) = 9, \quad TT_3(LO) = 11$$

hence no job can be selected for the lowest priority. Notice that this instance has a low load for a 2 *processor instance, in fact* $Load_{LO}(\mathbf{J}) = 1.3$*. Also notice that* any *priority assignment will lead to a correct schedule.*

3.1.2 Multiprocessor Scheduling

Whereas for uniprocessor scheduling a fixed-job-priority algorithm (EDF) is optimal, for multiprocessor case, dynamic job priorities are essential for optimality[DB11]. Moreover, the EDF heuristic can be very inefficient for multiprocessors. In seminal work of Dhall and Liu [DL78] it was shown that the *best*, *i.e.,* maximal, at which we can be sure to have a schedulable job set for EDF on multiprocessors is no better than for EDF on uniprocessor. For arbitrarily small $\epsilon > 0$ one can find a feasible job instance with load $1 + \epsilon$ that is not schedulable by EDF. For this, let us consider m small-deadline jobs with utilization ϵ/m each and one job with utilization 1 and a large deadline. If the last job, which has a large utilization, was given the highest priority then the schedule would be feasible.

In [Bar04] it was shown that in general implicit-deadline periodic task sets under global *fixed priority* for jobs have the following best guaranteed utilization: $(m + 1)/2$. Roughly speaking, the fixed priority scheduling can be guaranteed to find a multiprocessor schedule if the system is loaded by no more than *one half*, and even this is only possible if job priorities are well calculated, *e.g.,* the plain EDF cannot provide this guarantee, as explained earlier. Therefore, EDF modifications have been proposed to provide this guarantee. The main idea of several such algorithms is so-called 'separation' of jobs, *i.e.,* separating those that have low and high contribution to load. One of such algorithms is fpEDF, formulated for periodic tasks [Bar04], and later on generalized to sporadic tasks under name *EDF-DS*, where DS stands for *density separation* (see [DB11] for references). In our notation, this algorithm computes job density as $\delta_i = C_i/(D_i - A_i)$ and it differs from EDF by always giving the jobs with $\delta_i > th$ the highest priority, for a certain threshold th. Ties are broken arbitrarily. For the other jobs, the priority is the default EDF. Obviously, this strategy resolves the Dhall-effect counterexample mentioned earlier. However this approach does not give any schedulability assurance in the case of finite sets of jobs. Experiments shows that one can even loose in schedulability using a threshold $th = 1/2$. For finite job sets, experiments suggest to use a higher threshold to improve schedulability.

3.1.3 Precedence-constrained Scheduling

The *list scheduling* (see Appendix A) can be seen as generalization of fixed-priority scheduling by handling precedence constraints using *synchronization* between dependent jobs, *i.e.,* including wait for predecessor termination into the condition of job 'ready' status. Synchronization is essential for multiprocessors, whereas for single processor systems it may be sufficient to require precedence compliance of the priority [F$^+$10, Bar12]. For job priorities, it is generally recognized that the definition of EDF heuristics should be adjusted by using *ALAP deadlines* D^* instead of the nominal deadlines for priority assignment. For example, the list scheduling literature knows so-called 'ALAP' and b-level heuristics [KA99]. Single-processor scheduling uses this approach for priority assignment with adjusted deadlines [F$^+$10]. Sometimes the ALAP-adjusted EDF is a part of an optimal strategy, see [KA99] for further references.

3.1.4 Mixed-critical Scheduling

Single Processor MC scheduling

One of the most notable result in MCS scheduling is OCBP algorithm, which is Fixed Priority based. MCPI and MCEDF use the flexibility of fixed-priority per mode policies to be dominant over fixed priority algorithms. To the best of our knowledge, in the previous work no FPM algorithm [GESY11, BBD$^+$12a, EY12] has been proven to be theoretically dominant over OCBP. The priority assignment of [GESY11] applies OCBP to compute PT_{LO}, thus having equivalent schedulability. [BBD$^+$12a] proposes an efficient online algorithm with the optimal scaling factor and [EY12] presents a highly efficient priority computation method that dominates OCBP and several other algorithms empirically. Note, however, that [BBD$^+$12a, EY12] are not directly applicable to the problem studied in this paper as they are designed for a periodic job model with unknown arrival times.

The FPM policy provides better results than fixed priority, but in general dynamic-priority policies are necessary for optimality.

OCBP

OCBP is based on Audsley algorithm presented in Section 3.1.1. It selects the least-priority job J_i using the following criterion: when having the least priority in the working set, job J_i still meets its deadline in the scenario $c_j = C_j(\chi_i), \forall j$, *i.e.,* the basic scenario with the WCET at the criticality level χ_i, which is '*own*' for J_i. This can be checked by simulating[1] the scheduling with *any* priorities for the other jobs in the working set provided that they are higher than J_i. The correctness of this check is due to the following lemma [BBD+12b]:

Lemma 3.1.2. *The execution time available for a job J_i in a fixed priority scheduling algorithm depends on the arrival and execution times of jobs J_j with a priority higher than J_i, but not on their relative priority assignment.*

This is given by the fact that it the lowest priority jobs terminates at the end of its busy interval, and this can be computed using the (3.8). Thus at each step we can compute the exact worst case response time for the job to which we assign the lowest deadline, and this is where the optimality of OCBP comes from. Note that Lemma 3.1.2 confirms Assumption 2 of Audsley algorithm. Note also that this assumption holds because OCBP does not drop LO jobs even if a HI job exceeds its LO WCET.

The following example shows how OCBP works:

Example 3.1.3. *Let* **J** *be described by the following table:*

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

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

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

[1] [BLS10] uses a more efficient procedure - *makespan* (see Section 3.2.4)

Multiprocessor MC Scheduling

Some of the first works made on multiprocessor MC scheduling are based on temporal isolation techniques (Mollison*et al.* [MEA+10], Herman *et al.* [HKM+12]). This approach provides good isolation between criticality levels, but it gives worse performances compared to solutions that allows jobs at different criticality to run at the same time. Li and Baruah [LB12] proposed a global multiprocessor algorithm, fpEDF, with a complete theoretical analysis. This approach, however does not provide good utilization for high number of processors, and it was shown [BCLS14] that partitioned solutions are preferable when using the task model.

There are only a few works on precedence-constrained mixed-criticality scheduling. For single processor. In [Bar13], multiprocessor list scheduling algorithm was proposed. However, it is restricted to jobs that all have the same arrival and deadline times. Finally, [YKRB14] consider pipelined scheduling for task graphs. However, they implicitly assume that the deadlines are large enough, such that they can be ignored during the problem solving, as only period (throughput) constraints were considered and not deadline (latency) ones.

3.2 Priority DAG

In this section we will introduce the idea of Priority DAG (P-DAG). Informally it is a graph that defines a partial order on the jobs showing *sufficient* priority constraints needed to obtain a certain schedule. This structure makes it easier to reason on priorities than a priority table, since the latter is a total order and thus contains also *unnecessary* priority constraints. We will imply for the rest of this section that we are using preemptive list scheduling and that the jobs execute by default in the basic LO scenario. Recall that a priority table PT defines a total order on the set of jobs $\mathbf{J}$. A priority table PT defines one and only one schedule $\mathcal{S}$ when applying list scheduling on m processors, we indicate it with the following notation: $PT \vDash_m \mathcal{S}$.

3.2.1 Motivation

Before defining the concept of P-DAG, we will show in this section the reason why such a structure is needed. Figure 3.2 shows the pseudocode of an algorithm that computes priorities for mixed critical scheduling problem. The idea is to start with a good mixed criticality-unaware priority order (in this case EDF), and then to improve the priorities by raising the priorities of HI-critical jobs. All the priority based algorithm proposed in this work are based on this template. In terms of schedulability, this procedure is constrained by meeting the LO scenario deadlines, postponing the HI scenario checks until the final check.

A simple implementation of the HI job priority improvement is shown in Figure 3.3. This procedure increases the priorities of the HI jobs *w.r.t.* the LO jobs, while the relative priorities between the jobs of the same criticality level, LO or HI, remain deadline-monotonic. This is done in a manner similar to a bubble-sort in the PT array. We visit the HI jobs in decreasing priority order, and try to raise each HI job ('raising a bubble') by repeatedly swapping priority with the adjacent priority LO job. Subroutine $CanSwap(i, i-1, \ldots)$ simulates the fixed priority schedule PT with entries i and $i-1$ swapped and returns whether all deadlines are met. Subroutine *Swap* performs the actual swapping.

This procedure is illustrated in the following examples:

Algorithm: *MC-ALGO*
Input: **job instance J**
Output: **priority table** PT

1: $PT \leftarrow JobsOrderedByEDF(\mathbf{J})$;
2: **if** $LOscenarioFailure(PT, \mathbf{J})$ **then**
3: **return** (FAILURE-NON-SCHEDULABLE-INSTANCE)
4: **end if**
5: $PT \leftarrow ImproveHIJobs(PT, \mathbf{J})$
6: **if** $anyHIscenarioFailure(PT, \mathbf{J})$ **then**
7: **return** (FAILURE-MCEDF-SCHEDULABILITY)
8: **end if**

Figure 3.2: The algorithm for computing priorities

Algorithm: *MonotonicImproveHIJobs*
In/out: **priority vector** PT (deadline monotonic)
Input: **job instance J**

1: $i \leftarrow 2$
2: **while** $i \leq K$ **do**
3: $Swapped \leftarrow$ **False**
4: **if** $i \geq 2 \wedge \chi_{PT[i]} = \text{HI} \wedge \chi_{PT[i-1]} = \text{LO}$ **then**
5: **if** $CanSwap(i, i-1, \ PT, \mathbf{J})$ **then**
6: $PT \leftarrow Swap(i, i-1, \ PT)$;
7: $Swapped \leftarrow$ **True**
8: **end if**
9: **end if**
10: **if** $Swapped$ **then**
11: $i \leftarrow i - 1$
12: **else**
13: $i \leftarrow i + 1$
14: **end if**
15: **end while**

Figure 3.3: Improvement procedure, keeping the deadline-monotonic order between same-criticality jobs

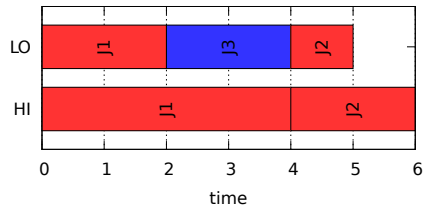


Figure 3.4: The Gantt chart of Example 3.2.1

Example 3.2.1. *Let* **T** *be the instance defined by the following table:*

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

The algorithm of Figure 3.2 will first give EDF priorities to the jobs, thus generating the following priority table:

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

Then the algorithm of Figure 3.3 will be called to improve this priority table. First, it will check condition of line 4 on J_1 and J_3. Since the condition holds, it will check if the swap between them is possible by checking if, after being moved to the second PT position, J_3 will still met its deadline in the LO scenario. In this case J_1 will execute for 2 time units, thus terminating at time 2, and then J_3 will execute for other 2 time units, thus terminating at 4. Since there is no deadline miss, we will accept the swap, thus obtaining:

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

Since there are no other possible swap, the algorithm terminates. Figure 3.4 shows that using this priority order all deadlines are met in all the possible scenarios of the instance.

However, this procedure may easily fail, as show in the next example:

Example 3.2.2. *Let* **T** *be the instance defined by the following table:*

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

The algorithm of Figure 3.2 will first give EDF priorities to the jobs, thus generating the following priority table:

$$PT = (J_1, J_3, J_4, J_2) \tag{3.2}$$

The only possible swap here is between jobs J_4 and J_3, but it will lead to a deadline miss of job J_3. The algorithm will then leave the priority table unchanged, not having improved any HI-job priority. Thus the algorithm will fail, since, as the reader may check, the priority

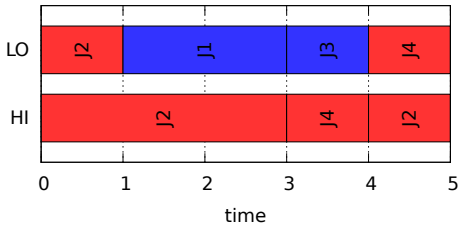


Figure 3.5: The Gantt chart of Example 3.2.2

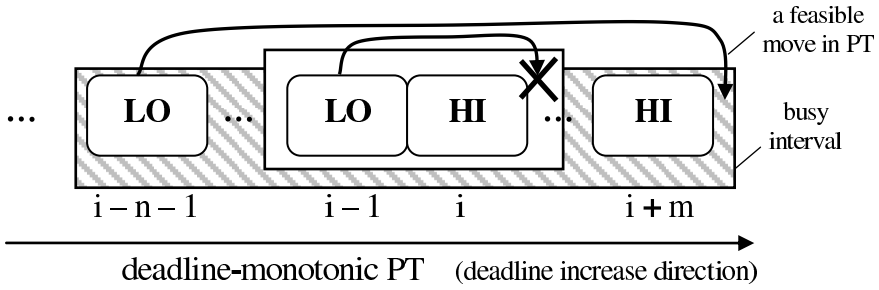


Figure 3.6: The blocking of the deadline-monotonic improvement

table of (3.2) will lead to a non-feasible schedule in scenario $HI - J_2$*. In this case, a correct priority table is the following:*

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

as shown in Figure 3.5.

The problem of the previous example, as sketched in Figure 3.6 is the fact that a LO job PT_{i-1} cannot be moved behind the HI job PT_i in the priority table still does not always exclude the possibility that a tighter-deadline LO job PT_{i-1-n} can be moved behind a looser-deadline HI job PT_{i+m}, whose completion time would thereby improve, which can be crucial for the HI-scenario schedulability. In the linear-array swapping procedure, the job pair (PT_{i-1}, PT_i) would *block* this possibility.

To avoid this kind of problems we introduce the concept of *Priority DAG* (P-DAG), that is intuitively a structure that represent how job really interfere with each other and allows us to "swap" job priorities in a DAG structure instead of linear chain structure.

3.2.2 P-DAG Definition and Properties

Consider a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$, a number of processors m and the graph $G = (\mathbf{J}, \rhd)$, where $\rhd$ is a partial order relation defined on $\mathbf{J}$. Though by default in this thesis we assume preemptive scheduling, in this section we also speculate on non-preemptive variant of priority-based scheduling algorithms.

Definition 3.2.3 (P-DAG)**.** *We call* $\mathbf{PT}(G)$ *the set of all priority tables that can be obtained by a topological sort of* $G(\mathbf{J}, \rhd)$*. In other words* $J_1 \rhd J_2 \Rightarrow J_1 \succ_{PT} J_2$ *for all* $PT \in \mathbf{PT}(G)$*. We also say that edges '*$\rhd$*' define relative priority constraints between jobs.* G *is a P-DAG on* m *processors for schedule* $\mathcal{S}$ *iff:*

$$\forall PT,\ PT \in \mathbf{PT}(G) \Rightarrow PT \vDash_m \mathcal{S} \tag{3.3}$$

We indicate the schedule generated by a P-DAG G as $\mathcal{S}(G)$. Two P-DAGs giving the same schedule are called *equivalent*. Formally:

Definition 3.2.4 (equivalent P-DAGs)**.** *Two P-DAGs G and G' are equivalent on m processors iff:*

$$\mathcal{S}(G) = \mathcal{S}(G')$$

Lemma 3.2.5. *A necessary condition for the* non *equivalence between two P-DAG $G(\mathbf{J}, \rhd)$ and $G'(\mathbf{J}, \rhd')$ is that*

$$\exists J_1, J_2 \mid J_1 \rhd J_2 \wedge J_2 \rhd' J_1 \tag{3.4}$$

The above comes from the consideration that, to obtain a different schedule, at a certain time t the scheduler must make a different choice in the two cases. This may only happen if the (3.4) is true.

Also, the following is trivial:

Lemma 3.2.6. *If adding an edge to a P-DAG G does not introduce a cycle, the resulting graph G' is still a P-DAG and it is equivalent to G. Also $\mathbf{PT}(G') \subseteq \mathbf{PT}(G)$.*

Definition 3.2.7 (Canonical P-DAG)**.** *A* Canonical *P-DAG for a schedule $\mathcal{S}$ is a P-DAG G:*

$$\forall PT,\ PT \in \mathbf{PT}(G) \Leftrightarrow PT \vDash_m \mathcal{S} \tag{3.5}$$

Definition 3.2.8 (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 where 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 whereas J_2 is not selected.*

It's trivial that:

$$J_1 \vdash_{\mathcal{S}} J_2 \Rightarrow J_1 \succ_{PT} J_2 \tag{3.6}$$

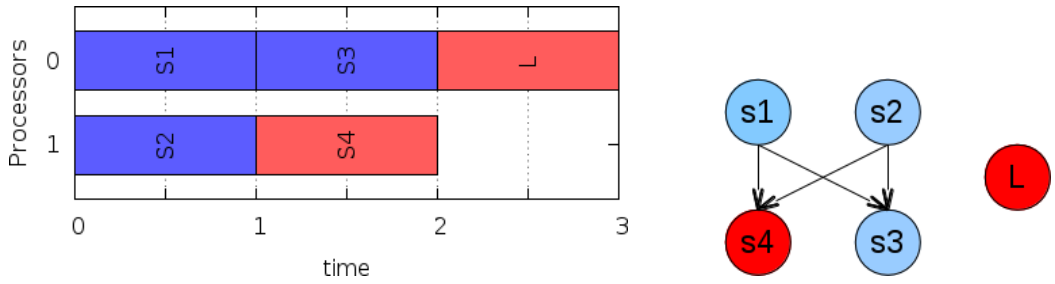
Note that the above definition depends on when the scheduler is allowed to take decisions. In a preemptive scheduler, decisions can be taken at any time, while in a non-preemptive one decisions are limited to time instant where at least one processor is idle. This can generate different blocking relation in the two cases, as shown in the following:

Example 3.2.9. *Consider the instance of Example 3.2.1 and the following priority order:*

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

The schedule generated in the LO scenario is the one shown in Figure 3.4 both in preemptive and non-preemptive case. However, in the preemptive case we will have the relation $J_3 \vdash J_2$, since when J_2 arrives (at time 3) the schedule may decide to preempt J_3 and schedule J_2, but does not so, because $J_3 \succ J_2$. In the non preemptive case, when J_2 arrives the scheduler may not take any decisions until J_3 terminates at time 4. At this time J_2 is the only ready job, thus $J_3 \nvdash J_2$.

Lemma 3.2.10. *Given a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$, a table PT and a number of processors m. Consider the* blocking *relation $\vdash_{\mathcal{S}}$, where $\mathcal{S}$ is such that $PT \vDash_m \mathcal{S}$. Then $G = (\mathbf{J}, \vdash_{\mathcal{S}})$ is a canonical P-DAG for $\mathcal{S}$.*



(a) Schedule (b) P-DAG

Figure 3.7: The figures of Example 3.2.11.

Proof. We need to prove that (3.5) holds. Let us first prove that G is actually a P-DAG (*i.e.,* (3.3) holds). This trivially comes from the observation that during the execution of the schedule $\mathcal{S}$, we only need to compare job priorities when a job blocks another. So the relative priority constraints defined by relation $\vdash_{\mathcal{S}}$ are *sufficient* to generate $\mathcal{S}$.

To prove that the priority constraints defined by $\vdash_{\mathcal{S}}$ are also *necessary*, let us suppose by contradiction that there exist a table PT' such that $PT' \vDash_m \mathcal{S}$ and $PT' \notin \mathbf{PT}(G)$. The latter means that $\exists\ J_1, J_2$ so that $J_1 \vdash_{\mathcal{S}} J_2$ and $J_1 \not\succ_{PT'} J_2$. By the first statement and by (3.6), we have $J_1 \succ_{PT'} J_2$ that contradicts the second statement. □

Example 3.2.11. *Let us consider the task graph of Fig 2.1, where* $\mathbf{J}$ *is 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*

consider the priority table $PT = \{s1 \succ s2 \succ s3 \succ s4 \succ L\}$. *On two processors* PT *produces the schedule* $\mathcal{S}$ *shown in Fig. 3.7(a). From the figure is easy to derive the blocking relation* $\vdash_{\mathcal{S}}$. *We have:* $s_1 \vdash s_3$, $s_2 \vdash s_3$, $s_1 \vdash s_4$, $s_2 \vdash s_4$. *Notice that* L *is never blocked, because, due to precedence constraints, it is never ready until time 2, when all its predecessors complete. From the blocking relation* $\vdash_{\mathcal{S}}$, *we can derive the canonical P-DAG* $G = (\mathbf{J}, \vdash_{\mathcal{S}})$, *shown in Fig. 3.7(b).*

The following trivially follows from Lemmas 3.2.10 and 3.2.6:

Lemma 3.2.12. *Consider a task graph* $\mathbf{T} = (\mathbf{J}, \rightarrow)$ *and a graph* $G = (\mathbf{J}, \rhd)$. *Let* $\rhd^*$ *be the transitive closure of* $\rhd$ *and* $\mathcal{S}$ *be a schedule generated by a priority table* $PT \in \mathbf{PT}(G)$. *Then* G *is a P-DAG iff:*

$$J' \vdash_{\mathcal{S}} J'' \Rightarrow J' \rhd^* J'',\ \forall J', J'' \in \mathbf{J} \tag{3.7}$$

Definition 3.2.13 (Redundant edges)**.** *An edge* $J_1 \rhd J_2$ *of a P-DAG* G *is called* redundant *iff there exists another path in* G *from* J_1 *to* J_2.

Definition 3.2.14 (Ineffective edges)**.** *An edge* $J_1 \rhd J_2$ *of a P-DAG* G *is called* ineffective *iff* J_1 *and* J_2 *belong to two different connected components of* $\vdash_{\mathcal{S}(G)}$.

Algorithm: *Forest_PDAG*
Input: **task graph** $\mathbf{T}(\mathbf{J}, \rightarrow)$
Input: **priority table** PT
Output: **P-DAG** $G(\mathbf{J}', \rhd)$

```
1:  G = (∅, ∅)
2:  while PT ≠ ∅ do
3:      J^Curr ← PopHighestPriority(PT)
4:      PT' ← TopologicalSort(G) ⌢ J^Curr
5:      G.J' ← G.J' ∪ {J^Curr}
6:      T' ← MaximalSubgraph(T, G.J')
7:      SimulateListSchedule(LO, T', PT')
8:      for all trees ST ∈ G do
9:          if ∃ J' ∈ ST : J' ⊢ J^Curr then
10:             G. ▷← G. ▷ ∪{(root(J'), J^Curr)}
11:         end if
12:     end for
13: end while
14: return G
```

Figure 3.8: The forest P-DAG generation algorithm

It is trivial that removing redundant and ineffective edges from a P-DAG G will not have any effect on $\mathcal{S}(G)$[2].

3.2.3 Forest-shaped P-DAG generation

A P-DAG can, in general, be any kind of DAG. We are interested in generating P-DAGs that are shaped like forests (*i.e.,* a collection of unconnected trees) directed towards the roots. Please note that in this thesis by "tree" we mean a directed tree, in the sense of a connected DAG such that all edges are directed towards a single edge, called root. Also note that some authors uses a different terminology by defining a directed tree simply as a DAG whose underlying undirected graph is a tree, and use the term "arborescence" to indicate the DAG that we defined as "tree". The reason why we want such a structure will be clear in the following sections, where we use the properties of forest to prove some properties of our algorithm.

We propose in this section an algorithm that generates a forest-shaped P-DAG. We will first explain the algorithm and then prove its correctness. The algorithm is shown in Fig. 3.8, it takes a task graph and a precedence compliant priority table as input and proceeds as follows. The highest priority job J^{Curr} is removed from the table PT and added to the graph G. Then we simulate a run of the jobs included so far in G and their precedences, using as priority table a topological sort of G (with J^{Curr} at the least position). During this simulation we keep note of the jobs that block J^{Curr} and add an edge to J^{Curr} from the root of all the subtrees of G that include a job that blocks J^{Curr}.

Example 3.2.15. *Consider the task graph and the priority table of Example 3.2.11. We will apply Forest_PDAG algorithm to them. In the first step the algorithm picks up* s_1*, the*

[2]Removing redundant edges, will not affect $\mathbf{PT}(G)$ as well, which is a stronger statement.

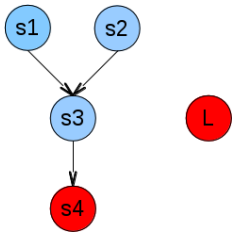


Figure 3.9: Forest P-DAG

*highest priority jobs from PT, and will add it to the graph. In the second iteration, we pick up s_2, since it is not blocked by any job, we continue without adding any arc. Then we pick up s_3, that is blocked by both s_1 and s_2, so we add the arcs (s_1, s_3) and (s_2, s_3). At the next iteration we pick up job s_4, that is also blocked by both s_1 and s_2, so we add an arc from the root of the tree that contains the blocking jobs (*i.e., *s_3*) to s_4. In the final iteration we pick up job L, that is not blocked by any job, thus we add it to the graph without inserting any arcs from it. The resulting graph is shown in Fig. 3.9.*

Theorem 3.2.16. *Let G be the graph generated by the* Forest_PDAG *algorithm. Then G is a P-DAG and a forest.*

Proof. We will prove both statements by induction, by showing that at the n-th step, the statement is true for the partial graph G_n obtained at the end of n-th iteration of the loop and whose nodes are thus the first n elements at the original PT provided to the algorithm. We denote by PT_n the table composed of the firsts n elements of that table and show at each step that G_n is a P-DAG for the schedule obtained from priority table PT_n, *i.e.,* $PT_n \vDash \mathcal{S}(G_n)$.

Basic step. The basic step is trivial. We have a priority table $PT_1 = \{J_1\}$ with one element and a graph $G_1 = (\{J_1\}, \emptyset)$. A graph of one element is a forest and the only possible topological sort of G_1 gives PT_1.

Inductive step. We know by inductive hypothesis that G_{n-1} is a P-DAG and $PT_{n-1} \vDash \mathcal{S}(G_{n-1})$. Also G_{n-1} is a forest. We only add edges to J_n from the root of unrelated subtrees, this operation may only generate another tree, thus G_n is a forest.

G_n is a P-DAG, by construction, since the loop from line 8 to 11 (see Fig. 3.8) ensures property (3.7) of Lemma 3.2.12. Also, since J_n has no successors in G, during a topological sort of G_n we can give J_n the n-th position in the priority table, same position it has in PT_n. For the other jobs, the partial graph that we have to explore is exactly G_{n-1}, so we can generate PT_{n-1} from it. Since by construction up to the $(n-1)$-th element PT_n and PT_{n-1} are equal, we can generate PT_n by topological sort of G_n. Thus $PT_n \vDash \mathcal{S}(G_n)$.

□

3.2.4 P-DAGs and Single-Processor Busy Intervals

P-DAGs assume a special meaning in single processor system, since they are strictly related to the concept of *busy interval*.

Definition 3.2.17 (Busy Interval)**.** *Consider work-conserving policy and an instance* **J**. *A* busy interval *is an open time interval (τ_1, τ_2) in $\mathcal{S}$ that is a maximal time interval where the set of ready jobs is never empty (assuming, for convenience, that the interval where a job is ready is also open).*

Note that the set of busy interval *does not depend* on the selected work-conserving policy, but only on $\mathbf{J}$. Also, the interval is half-open because the jobs arriving at time t count ready only for the time instances strictly later than t. It is obvious that neither the start time τ_1 nor the length of a busy interval $\tau_2 - \tau_1$ depend on the exact priority assignment. In fact this is so because the former is given by:

$$\tau_1 = \min_{J_i \in \mathbf{J}_{BI}} \{A_i\}$$

and for the latter we have:

$$\tau_2 - \tau_1 = \sum_{J_i \in \mathbf{J}_{BI}} c_i \tag{3.8}$$

where $\mathbf{J}_{BI} \subseteq \mathbf{J}$ is the set of the jobs running in the busy interval.

By abuse of terminology, we apply the term 'busy interval' also to $\mathbf{J}_{BI}$, and denote it BI. In general, a job set $\mathbf{J}$ can be partitioned into multiple busy intervals, because some jobs in $\mathbf{J}$ may arrive at or later than the end of a busy interval of some other jobs in $\mathbf{J}$. In single processor system the following holds:

Lemma 3.2.18 (Single processor P-DAG and BI)**.** *Let $G = (\mathbf{J}, \rhd)$ be a P-DAG without any ineffective edge. Then jobs belonging to the same connected component of G belongs to the same busy interval BI of $\mathbf{J}$ as well. Formally:*

$$J_1(\rhd \cup \rhd^{-1})^* J_2 \Rightarrow J_1, J_2 \in \mathbf{J}_{BI}$$

Proof. It is trivial that jobs belonging to the same connected components of $\vdash_{\mathcal{S}(G)}$ belong to the same busy interval as well, since jobs from different busy intervals may not block each other. For the hypothesis of the absence of ineffective edges we have that $J_1(\rhd \cup \rhd^{-1})^* J_2$ implies that J_1 and J_2 belong to the same connected components of $\vdash_{\mathcal{S}(G)}$ (see Definition 3.2.14). □

To compute the busy intervals one can use the *makespan* procedure [BLS10], which, for a given mode χ, works as follows: Let $\mathbf{J}_\chi$ denote the jobs in $\mathbf{J}$ with criticality level χ or higher: $\mathbf{J}_\chi = \{J_i \in \mathbf{J} \mid \chi_i \geq \chi\}$. Also let $J_1, J_2, \ldots, J_{n\chi}$ denote all the jobs in $\mathbf{J}_\chi$ ordered by non-decreasing arrival times. Consider the sequence $f_1, f_2, \ldots, f_i$ of numbers defined according to the following recurrence:

$$\begin{aligned} f_1 &= A_1 + C_1(\chi) \\ f_i &= \max(f_{i-1}, A_i) + C_i(\chi) \qquad i > 1 \end{aligned} \tag{3.9}$$

Then the latest termination time (makespan) of a preemptive work-conserving schedule at criticality χ is given by $F_\chi = f_{n\chi}$. To take note of all the busy interval, we have to consider that whenever in equation (3.9) we have that $f_{i-1} < A_i$ the previous busy interval ends at time f_{i-1} and the new one starts at time A_i.

The following holds [BLS10]:

Lemma 3.2.19. *Makespan computation has linear complexity when applied to jobs pre-sorted by arrival times.*

3.2.5 P-DAGs and Potential Interference on Multiprocessor

In this section we extend the concept of busy intervals for the multiprocessor case, introducing the *potential interference relation.*

Definition 3.2.20 (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:*

$$\forall J_1, J_2 \in \mathbf{J}' : (J_1 = J_2 \ \vee \ \exists PT : J_1 \vdash_{PT} J_2) \quad \Rightarrow \quad J_1 \overset{\mathbf{J}'}{\sim} J_2$$

whereby we consider LO-mode m*-processor list schedules with priority table* PT *applied to maximal task subgraph with nodes* $\mathbf{J}'$*.*

We can extend Lemma 3.2.18 for the multiprocessor case:

Lemma 3.2.21 (P-DAGs and Potential Interference)**.** *Let* $G = (\mathbf{J}, \rhd)$ *be a P-DAG without any ineffective edge. Then jobs belonging to the same connected component of* G *belongs to the same equivalence class of potential interference relation* $\overset{\mathbf{J}}{\sim}$ *as well. Formally:*

$$J_1(\rhd \cup \rhd^{-1})^* J_2 \Rightarrow J_1 \overset{\mathbf{J}}{\sim} J_2$$

The proof is similar to the one of lemma 3.2.18.

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 3.2.22 (Single-Processor Interference)**.** *In list scheduling 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.

Calculating the minimal potential interference on a single processor can be done in linear time using the makespan procedure, as explained in Section 3.2.4.

The following lemma is easy to prove:

Lemma 3.2.23 (Least priority in a busy interval)**.** *Given a job set* $\mathbf{J}'$ *and any of its busy interval* BI *of instance* $\mathbf{J}'$ *with time interval* (τ_1, τ_2)*. In fixed-priority scheduling for job set* $\mathbf{J}'$*, the least-priority job running in this* BI *terminates at time* τ_2 *and is blocked by at least one other job in* BI *(if there are any).*

The following theorem can be easily derived from the above lemma:

Theorem 3.2.24 (Minimal Potential Interference in Busy Intervals)**.** *Given a job set* $\mathbf{J}'$ *without precedences, then the set of busy intervals* BI *are the equivalence classes of the corresponding minimal potential interference relation* $\overset{\mathbf{J}'}{\sim}$ *on single processor.*

Algorithm: *MCEDF*
Input: **job instance J**
Output: **priority table** *PT*

1: **if** *LOscenarioFailure*(**J**) **then**
2: **return** (FAIL-NON-SCHEDULABLE)
3: **end if**
4: $G \leftarrow MCEDF\_PDAG(\mathbf{J}, \emptyset, \emptyset)$
5: $PT \leftarrow TopologicalSort(G)$
6: **if** *anyHIscenarioFailure*$(PT, \mathbf{J})$ **then**
7: **return** (FAIL-NON-SCHEDULABLE-BY-MCEDF)
8: **end if**

Figure 3.10: The MCEDF algorithm for computing priorities

Corollary 3.2.25 (Potential interference with precedences)**.** *Given a task graph* $\mathbf{T}$ *and a subset of jobs* $\mathbf{J}'$*. Let* $\mathbf{T}'$ *be the maximal subgraph of* $\mathbf{T}$ *with nodes* $\mathbf{J}'$*. The busy intervals* BI *of task* $\mathbf{T}'$ *correspond to equivalence classes of some (not necessarily minimal) potential interference relation* $\overset{\mathbf{J}'}{\sim}$ *on single processor.*

We cannot claim minimality in the second case above because currently we are not sure about it.

3.3 Independent Jobs Single Processor Scheduling – MCEDF

3.3.1 Mixed Critical Earliest Deadline First

Our proposed *Mixed-Critical Earliest Deadline First* (MCEDF) algorithm uses FPM policy, thus when χ_{mode} = HI, the scheduling problem is a standard non MC problem, for which EDF is optimal in single processor case. The problem is then reduced to compute PT_{LO}, which we will call just PT for the rest of this chapter. The algorithm is formulated here for *independent jobs*, but it can be easily extended to support task graphs.

The algorithm to compute PT is shown in Figure 3.10. Initially, we compute the schedulability of LO scenario in subroutine *LOscenarioFailure*, by running EDF. By optimality of EDF for single criticality level, if a job misses the deadline, then the instance is not schedulable. Thus the algorithm establishes that MC schedulability Condition 1 (see Section 2.1.1) is satisfiable, and this remains invariant of the algorithm which ensures that the final priority table PT computed by the algorithm also satisfies this condition. While satisfying Condition 1, the algorithm applies a best-effort heuristic to ensure Condition 2, *i.e.,* that the deadlines of all HI jobs are met in any basic HI scenario, by trying to ensure that the priorities of the HI jobs are as high as possible, under the constraint that all jobs meet their deadlines in the LO scenario (*i.e.,* Condition 1 is still satisfied).

To compute the final priority table, MCEDF first calls subroutine *MCEDF_PDAG*, which generates a P-DAG with HI job priorities improved *w.r.t.* the original EDF table. Subroutine *TopologicalSort* employs the well-known topological sort algorithm to generate the priority table PT. Finally, the subroutine *anyHIscenarioFailure* evaluates whether Condition 2 is met. In this case the algorithm succeeds. The check is done by a simulation over the set of job specific HI scenarios HI-J_n in line with Theorem 2.1.2. In Chapter 4 we show a more

Algorithm: *MCEDF_PDAG*
Input: **job instance** $\mathbf{J}'$
Input: **node** J^{parent}
In/out: **P-DAG** G
1: **BI** $\leftarrow$ *PartitionIntoBIs*($\mathbf{J}'$);
2: **for** all $BI \in$ **BI do**
3: $J^{\text{least}} \leftarrow$ *SelectLeastPriorityJob*(BI)
4: $G.\mathbf{J}' \leftarrow G.\mathbf{J}' \cup \{J^{least}\}$
5: **if** $J^{parent} \neq \emptyset$ **then**
6: $G. \rhd \leftarrow G. \rhd \cup \{(J^{least}, J^{parent})\}$
7: **end if**
8: $\mathbf{J}'' \leftarrow BI \setminus \{J^{\text{least}}\}$
9: $MCEDF\_PDAG(\mathbf{J}'', J^{least}, G)$
10: **end for**

Figure 3.11: The MCEDF algorithm for computing P-DAG

efficient implementation of the schedulability check that does not use exhaustive simulation.

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

The subroutine *MCEDF_PDAG* is defined in Figure 3.11. The algorithm is based on the concept of busy interval, as defined in Section 3.2.4. The P-DAG construction algorithm splits some subinstance $\mathbf{J}'$, $\mathbf{J}' \subseteq \mathbf{J}$ into BI's and selects the least priority job in each BI (see Figure 3.11, line 3). Observe that in a busy interval (τ_1, τ_2), the selected job will terminate at time τ_2, which can be computed by equality (3.8). Let $J^{\text{low}}_{\text{LO}}$ and $J^{\text{low}}_{\text{HI}}$ be the latest deadline [3] job among the LO and the HI jobs of BI respectively. Subroutine *SelectLeastPriorityJob* selects the least priority job according to the following rule.

- **if** $\exists J_j \in BI : \chi_j = \text{LO} \ \wedge \ J^{\text{low}}_{\text{LO}}.D \geq \tau_2$
- **then** $J^{\text{least}} \leftarrow J^{\text{low}}_{\text{LO}}$
- **else** $J^{\text{least}} \leftarrow J^{\text{low}}_{\text{HI}}$

This rule prefers to assign the least priority to $J^{\text{low}}_{\text{LO}}$ if BI has LO jobs and if the latest-deadline one would not miss its deadline. Otherwise the algorithm has no other choice but to select a HI job. Thus, the algorithm greedily avoids assigning a HI job the least priority, and does so only if otherwise it would break Condition 1. Let us now show that in a feasible problem instance this rule makes a choice that is feasible for the LO scenario. In uniprocessor scheduling the choice of the least-priority job affects the schedulability of that job only. Thus, we only need to ensure that this job meets the deadline. The job selected by the described rule can only miss its deadline if the latest-deadline job among all jobs in BI would also miss its deadline, which is only possible in an unfeasible instance.

The P-DAG G has multiple subtrees that correspond to the BI's of the complete problem instance $\mathbf{J}$. Subroutine *PartitionIntoBIs* in Figure 3.11 splits the currently examined instance into BI's. Then the subroutine *MCEDF_PDAG* examines every busy interval BI to select

[3] for equal-deadline jobs we break the ties by selecting the job with minimal $C_j(\text{HI}) - C_j(\text{LO})$. This choice is explained in Section 3.3.2.

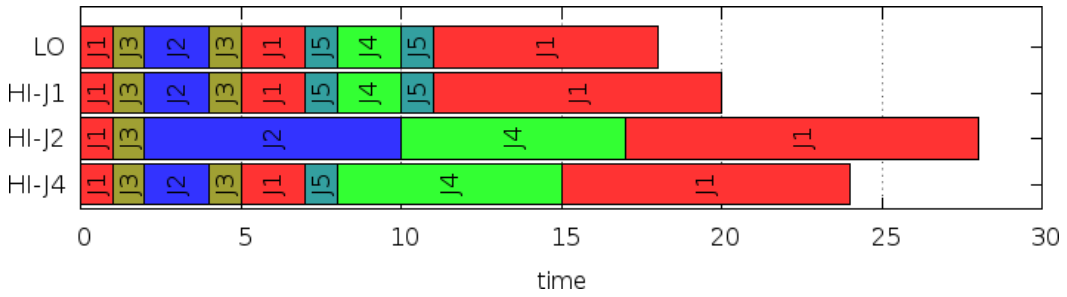


Figure 3.12: The Gantt charts for Example 3.3.1 with $PT = (2, 4, 3, 5, 1)$

the least-priority job in that interval. Afterwards the algorithm continues recursively with sub-instances $\mathbf{J}'' = BI \setminus \{J^{\text{least}}\}$. Removing a job from a BI reveals further fragmentation into busy intervals, which become direct children of J^{least} in the P-DAG. In those new BI's the same algorithm is used to find the least-priority job and to construct the subtree further from the roots to the leafs.

Example 3.3.1. *Let the instance* $\mathbf{J}$ *be defined by:*

Job	A	D	χ	$C(LO)$	$C(HI)$
1	0	30	*HI*	10	12
2	2	10	*HI*	2	8
3	1	8	*LO*	2	2
4	8	17	*HI*	2	7
5	7	11	*LO*	2	2

MCEDF computes the following solution $PT_{MC} = (2, 4, 3, 5, 1)$. *Let us demonstrate MCEDF computations step-by-step. MCEDF starts by checking whether the instance is schedulable in the 'LO' scenario by a simulation with* $PT_{EDF} = (3, 2, 5, 4, 1)$. *Instead, Figure 3.12 row 'LO' shows a simulation for the* PT_{MC}*; no deadline is missed there, and hence the same should hold for* PT_{EDF} *as well.*

Then MCEDF generates the P-DAG, see Figure 3.13. Instance $\mathbf{J}$ *has one busy interval* BI. *We can see this by the LO-scenario simulation in Figure 3.12, where job* J_1 *remains ready in interval* $(0, 18)$, *which implies that the processor is continuously busy until all jobs finish. Thus* $BI = \mathbf{J}$ *corresponds to a single tree in the P-DAG, for which we should select the overall least-priority job as the root. For the considered* BI, $J^{\text{low}}_{LO} = J_5$ *and* $J^{\text{low}}_{HI} = J_1$. *Since* $D_5 = 11 < 18$, *we cannot select* J_5, *so we select* J_1 *as* J^{least} *for the P-DAG root. Now we split the subinstance* $\mathbf{J} \setminus \{J_1\}$ *into* BI*'s, obtaining* $BI' = \{J_3, J_2\}$, *running in* $(1, 5)$ *and* $BI'' = \{J_5, J_4\}$, *running in* $(7, 11)$, *selecting, respectively,* J_3 *(since* $D_3 \geq 5$*) and* J_5 *(since* $D_5 \geq 11$*). The remaining subinstances have only one job, so the final P-DAG generation steps are trivial (see Figure 3.13). Priority table* PT_{MC} *satisfies the partial order of the resulting P-DAG. Finally the algorithm runs the simulations for the HI scenarios, which deviate from the basic LO scenario by switching to the HI mode at some HI job* J_j, *as illustrated in rows 'HI-*J_j*' in Figure 3.12. Because, as the reader can verify, the deadlines are met, the algorithm succeeds.*

Lemma 3.3.2. *The graph* G *generated by MCEDF_PDAG is a P-DAG and a forest.*

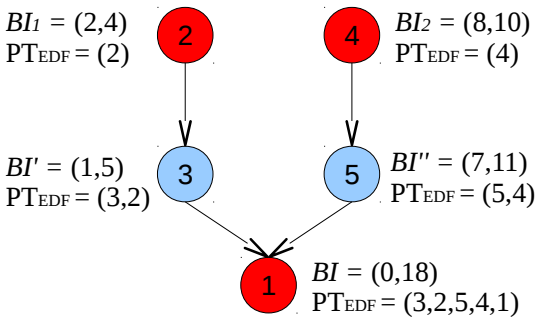


Figure 3.13: The P-DAG for Example 3.3.1; each node is annotated by the selected job index.

Proof. To prove that G is a P-DAG, consider two jobs J_1, J_2 such that $J_1 \vdash J_2$. This implies that $\exists BI : J_1, J_2 \in BI$. By construction, this implies that J_1 and J_2 have a closest common ancestor J^a in G. It may not be that $J^a = J_2$, because then we will have that $J_2 \triangleright J_1$, which contradicts $J_1 \vdash J_2$. Suppose that $J^a \neq J_1 \wedge J^a \neq J_2$. Then let ST^a be the subtree of G rooted in J^a and $\mathbf{J}^a = ST^a \setminus \{J^a\}$. Since J^a is the closest ancestor of J_1 and J_2, they will not have any ancestor in $\mathbf{J}^a$, thus $J_1 \overset{\mathbf{J}^a}{\nsim} J_2$, which also contradicts $J_1 \vdash J_2$. Thus it may only be that $J^a = J_1$, we thus we have $J_1 \vdash J_2 \Rightarrow J_1 \rightarrow J_2$ and hence the theorem is true by Lemma 3.2.12. □

Lemma 3.3.3 (MCEDF Complexity)**.** *MCEDF has an implementation with complexity* $O(k^2)$ *where* $k = |\mathbf{J}|$.

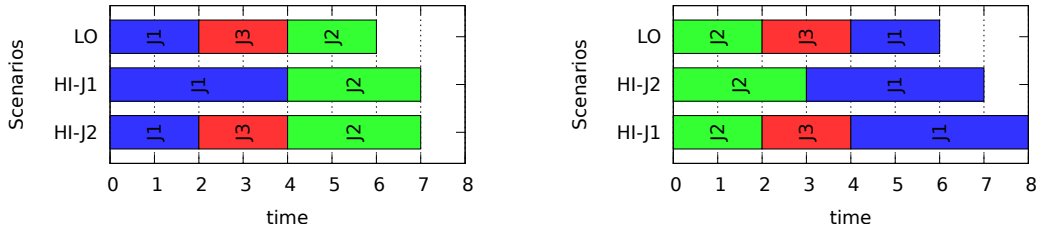
Proof. First of all, let us agree that we represent each subinstance $\mathbf{J}'$ by a list that is initially presorted (in time $O(k \log k)$) by arrival times. Note that when slitting subinstances into busy intervals to obtain new subinstances they can stay sorted by arrival times without any additional sorting as they are obtained by simple decomposition of pre-sorted list into more lists. In *LOscenarioFailure* we perform one fixed-priority schedule simulation. By Lemma A.2 in Appendix A, taking into account $m = 1$ and $E = 0$, the total cost of one simulation is $O(k \log k)$.

Graph G, being a collection of trees, has k nodes and at most $k-1$ edges. The complexity of *TopologicalSort* for such graphs is $O(k)$ [CSRL01].

We now analyze the complexity of *MCEDF_PDAG*. The most time-costly procedure at each node is the partitioning of current subinstance $\mathbf{J}'_i = ST^i - \{J_i\}$ into busy intervals. Because the subinstance is previously sorted by the arrival times, this can be done in a time linear in $|\mathbf{J}'_i|$ by the linear complexity makespan procedure (see Lemma 3.2.19). Next to splitting into BI's, the other basic procedure at each P-DAG node is the selection of the least-priority job J_i in $\mathbf{J}'_i$, which is also linear in $|\mathbf{J}'_i|$ as a selection of the maximal-deadline elements in the list.

Now observe that all $\mathbf{J}'_i$ together at each tree level contain at most k jobs, with exactly k jobs at the root level and removing some of them when going from the roots to the leafs. So, the tree generation cost is $O(k^2)$ per level. Because there are at most k tree levels, the total P-DAG generation complexity is $O(k^2)$.

Finally *anyHIscenarioFailure* can be done in $O(k \log k)$ time, as shown in Chapter 4. □



(a) $PT = J_1 \succ J_3 \succ J_2$ (b) $PT' = J_2 \succ J_3 \succ J_1$

Figure 3.14: The Gantt charts of Example 3.3.4

3.3.2 Support Priority Table

As the candidate for least priority job, J_{LO}^{low} and J_{HI}^{low}, the MCEDF assigns the latest-deadline job at the given criticality level. MCEDF, however, does not prescribe anything specific to break the tie in case multiple jobs have the same deadline. Certain properties of MCEDF do not depend on how ties are broken. However, for better schedulability and for certain other properties we have to specify this. In this case MCEDF we assumes that the user provides so-called "support" priority table, denoted SPT. This table must be EDF-compliant, that is:

$$J_1.D < J_2.D \Rightarrow J_1 \succ J_2$$

J_{LO}^{low} and J_{HI}^{low} are now unambiguously identified by MCEDF as least-priority jobs according to SPT. To disambiguate equal-deadline jobs in SPT, the user may chose his preferable heuristic criteria. In our MCEDF implementation we use the following heuristic. In case of equal-deadline jobs we break the ties by selecting for less priority the job with minimal uncertainty $\Delta C_j = C_j(\text{HI}) - C_j(\text{LO})$. Intuitively the reason why we do so stands in the observation that jobs with high uncertainty are 'more critical', since the quantity of additional computation they add in the case of a switch is higher. We will show the advantage of this choice with the following:

Example 3.3.4. *Consider the instance* **J** *defined by the following table:*

Job	*A*	*D*	χ	*C(LO)*	*C(HI)*
1	0	7	*HI*	2	4
2	0	7	*HI*	2	3
3	0	3	*LO*	1	1

In case we break ties using the minimal uncertainty, MCEDF will generate the following priority table:

$$PT = J_1 \succ J_3 \succ J_2$$

Using this priority the jobs will meet the deadline in all scenarios, as shown in Figure 3.14(a). In case we would select the job with the highest uncertainty, MCEDF will generate the following priority table:

$$PT' = J_2 \succ J_3 \succ J_1$$

This table is not schedulable, since J_1 *misses the deadline in scenario HI-*J_1*, as show in Figure 3.14(b).*

3.3.3 Dominance over OCBP

In this subsection we provide a theoretical evidence that MCEDF dominates OCBP. Example 3.3.1 gives an MCEDF-schedulable instance which is not OCBP-schedulable. The latter can be shown as follows. Suppose one can select the least OCBP-priority job in this instance. It cannot be a LO job, because, as shown earlier (see Figure 3.3.1), instance $\mathbf{J}$ consists of a single BI that finishes at time 18, when any LO job would miss its deadline. If we could select a HI job, then OCBP would evaluate its completion time by effectively extending the aforementioned LO-scenario BI into a longer HI-scenario BI where all HI jobs take $C_j(\text{HI}) - C_j(\text{LO})$ extra time. Summing up these differences, this adds 13 time units to completion time 18. But the completion time 31 is beyond the latest HI job deadline, $D_1 = 30$.

Thus, the dominance is given by the following:

Theorem 3.3.5. *If an instance is OCBP schedulable, then it is schedulable by the MCEDF algorithm as well.*

Proof. Recall that, by P-DAG definition, the preference for one particular topological order of the P-DAG does not impact the MCEDF schedulability. Similarly, when OCBP has multiple choices for the selection of the least priority job then preferring a particular choice does not matter for the OCBP schedulability [BLS10]. So, we will show that if one follows certain rules in making a choice in the MCEDF and OCBP, then both algorithms will construct the same priority table PT for any OCBP-schedulable instance $\mathbf{J}$.

Let us first examine in detail how MCEDF constructs PT. At any step of the topological sort, there is a 'ready set' (RS), *i.e.,* the set of busy intervals $\{BI_i^{\text{RS}}\}$ of the P-DAG nodes v_i that are not yet selected but whose parent node has already been selected. Implicitly, there is a sub-instance $\mathbf{J}'$ of which BI_i^{RS} are the busy intervals. MCEDF can choose to pick any BI to provide its J^{least} as the least-priority job in sub-instance $\mathbf{J}'$. What we have to show is that if $\mathbf{J}'$ is OCBP-schedulable then at least one BI will provide a job J^{least} that can be selected for the least OCBP priority as well.

- **Case 1: There is a BI_i^{RS} whose J^{least} is a LO job.**
  In this case, OCBP can select the J^{least} of any such busy interval. This is because when evaluating whether a LO job can be assigned the least priority OCBP simulates the basic LO scenario, effectively doing the same check as MCEDF.
- **Case 2: The J^{least} in every busy interval is a HI job**
  In this case, the MCEDF rule to select the J^{least} in a BI implies that the end time of every BI_i^{RS} is later than the deadline of any LO job contained in it. Consequently, no LO job can be selected by OCBP, because in an OCBP simulation a least-priority LO job will complete at a time equal to the end time of its BI_i^{RS}, thus missing its deadline.

  Therefore, because instance $\mathbf{J}'$ is OCBP-schedulable, OCBP should be able to select a HI job. Let us denote this job J' and let J'^{least} be the HI job selected by MCEDF for the busy interval BI_i^{RS} where J' is located. Because MCEDF selects the latest-deadline HI job, we have: $J'^{\text{least}}.D \geq J'.D$.

  The HI jobs are evaluated by OCBP using the HI scenario where no LO jobs are dropped and the jobs have $C_j(\text{HI})$ execution times. Because these execution times are larger or equal than the execution times in the basic LO scenario and no LO jobs are

dropped we conclude that J' and J'^{least} must be located in the same busy interval not only in the same scenario, but also in the HI scenario. The fact that J' can be selected by OCBP means that if it completes at the end of this HI busy interval then it still meets its deadline. But because the deadline of J'^{least} is not less than that of J', it is eligible to let J'^{least} complete at the end of that HI busy interval as well, and hence it can also be selected by OCBP.

Thus, for an OCBP-schedulable instance, both algorithms can construct the same PT. MCEDF uses this priority table before the mode switch, thus having exactly the same behavior as OCBP under these conditions. After the mode switch OCBP meets the HI job deadlines without dropping the LO jobs, and MCEDF will surely be able to do the same because it drops the LO jobs and employs EDF, an optimal strategy. □

To the best of our knowledge, so far MCEDF is the only FPM scheduler that exploits the freedom to drop the LO jobs (or to reduce their priority) to perform, *in theory*, strictly better than OCBP, the optimal fixed-priority scheduler.

3.3.4 MCEDF and Splitting

In this section we will introduce *splitting*, a theoretical transformation[4] of a job instance into a new instance where a HI job is equally divided into a certain number (called *split factor*) of equal smaller jobs, whose total execution times $C_j(LO)$ and $C_j(HI)$ add up to that of the original job. Obviously, the splitting does not impact $Load_{\text{LO}}$ and $Load_{\text{HI}}$, but it reduces the uncertainty and $Load_{\text{MIX}}$. Therefore, for mode-switched policies, such as MCEDF, the splitting can translate an unschedulable instance into a schedulable one. An infinitely large splitting of all HI jobs can bring the optimality of a mode-switched policy infinitely closer to that of the clairvoyant scheduling. For some instances, a finite splitting is enough to equate the clairvoyant scheduling. Mode-ignorant policies, such as OCBP, cannot take any advantage of splitting by construction. These observations are confirmed in our experiments in Section 4.4.2.

The following example demonstrates the effect of splitting. It has $Load_{\text{MIX}} = 1.166\ldots$:

Job	A	D	χ	C(LO)	C(HI)
1	0	6	LO	5	5
2	0	12	HI	2	12

This instance is not schedulable because the necessary condition (2.2) is broken and due to uncertainty of the execution time. If J_1 executes first then J_2 starts at time 5. In the LO scenario there would be no problem, but J_2 misses its deadline should it 'decide' to execute in the HI scenario, for 12 time units. Otherwise, if J_2 starts first then even in the HI scenario it meets its deadline (whereby the LO job J_1 can be dropped), but there is a problem in the LO scenario, as J_2 would delay J_1 by two time units, leading to a missed deadline. The clairvoyant scheduler would know the scenario in advance and make the proper choice accordingly.

It is easy to check that after splitting J_2 into two jobs, the instance becomes MCEDF-schedulable.

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

Job	A	D	χ	C(LO)	C(HI)
1	0	6	LO	5	5
21	0	12	HI	1	6
22	0	12	HI	1	6

The scheduler can execute J_{22} until completion, effectively getting from it the online knowledge of the execution scenario that was missing in the previous case. If job J_{22} has executed in the LO scenario, J_1 can follow, starting at time 1, and then J_{21} can run from time 6 even until time 12 in the HI scenario. If job J_{22} has executed in the HI scenario, J_1 will be skipped, and J_{22} together with J_{21} meet the deadline. Compared to the instance before the split, $Load_{\text{MIX}}$ reduces from $1.166\ldots$ to 1, whereas $Load_{\text{LO}} = 0.833\ldots$ and $Load_{\text{HI}} = 1$ stay constant, not showing any advantage of the split instance *w.r.t.* the original one.

Note that the splitting, even being a theoretical transformation, may have some practical significance. This depends on the WCET tools, in particular, by what extent the sum of WCETs may change by the splitting of code into blocks. Note that despite the fact that the arrival times of all subjobs are equal, they are not restricted to be data-independent of one another. This is due to the fixed-priority per job scheduling policy, which has the property that the jobs with equal arrival times never preempt each other but instead execute in a sequential priority-driven order and the sequential blocks of the job code can be assigned to the subjobs in the same order.

3.4 Multi Processor Scheduling – MCPI

We define here the *Mixed Criticality Priority Improvement* (MCPI) algorithm. It is basically an algorithm to compute job priorities under list scheduling offline, while online for predictable response time we use precedence-unaware global fixed priority with adapted arrival times to account for precedences.

We first discuss the offline computation of priorities and then we describe the online policy.

3.4.1 Preliminaries

As previously discussed, our aim is to overcome the limitation of Audsley approach in multiprocessor systems, by assigning priorities starting from the highest. This allows us to compute the exact job termination times. The drawback of this approach is that, unlike Audsley approach (see Section 3.1.1), just picking up a job that meets the deadline is not enough for optimality, as, unlike *e.g.,* OCBP, the choice made out of different alternatives has effect on the final outcome. Thus we lose the property of Audsley approach that ensures optimality of the priority table just by selecting a schedulable job at each step.

Experiments (see Section 3.6) show that if ideal Audsley approach could find tight upper-bounds on termination times it would constitute a serious competitor to MCPI on multiple processors. Nevertheless, we also show in Section 3.5 that on single processor MCPI is equivalent to MCEDF, thus dominating Audsley algorithm (OCBP) in this case.

Figure 3.15 shows an overview of MCPI. The algorithm takes as input the task graph $\mathbf{T}$, the number of processors m and a priority table SPT. The latter may be generated by any known multiprocessor algorithm. We call this algorithm *support algorithm* and the input priority table *Support Priority Table*, by analogy to MCEDF.

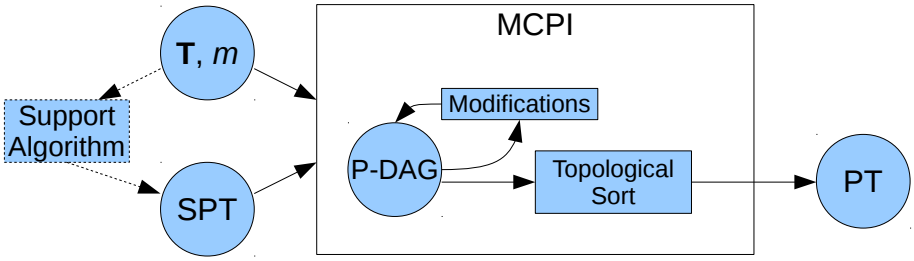


Figure 3.15: Proposed algortirhm MCPI. **T** stands for task graph and *SPT* for support priority table.

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. Similarly to MCEDF, the algorithm is based on the concept of *Priority Direct Acyclic Graph*, (P-DAG), but unlike MCEDF, we construct the P-DAG by adding at each step a job with the highest priority (according to SPT) and not the least one. 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, as shown in Section 3.2.1. 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 support table intact. Therefore, the main goal of the algorithm to compute and validate the PT_{LO}, 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.

MCPI uses potential interference relation to determine the set of LO jobs that may interfere with a HI job. The algorithm tries to improve the priority of a HI job over such LO jobs.

We use Theorem 3.2.22 to compute the potential interference relation, *i.e.,* 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 (see Section 3.2.5).

First we describe the MCPI algorithm itself, then we describe our support algorithm for it and we finish Section 3.4 by describing the online scheduling policy.

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 3.16: The MCPI algorithm

3.4.2 MCPI Algorithm Specification

The pseudocode of MCPI is given in Figure 3.16. The algorithm takes as inputs the support priority table SPT and the task graph $\mathbf{T}$. We require SPT to satisfy precedence compliance property (2.3), 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 task-graph sources to task-graph sinks. To ensure the compliance to the task graph, the algorithm calls the *DependencyComplianceTransform* algorithm, introduced in Section 2.2.2, 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, similarly to MCEDF, we obtain a priority table from G by using the well-known *TopologicalSort* procedure (see *e.g.,* [CSRL01]), 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 scenarios $HI\text{-}J_h$, in line with Theorem 2.1.2.

In Figure 3.17 subroutine *MCPI_PDAG* is shown. This subroutine is very similar to the algorithm of Figure 3.8. 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

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

1: **while** $G.\mathbf{J}' \neq \mathbf{T}.\mathbf{J}$ **do**
2: $J^{\text{curr}} \leftarrow SelectHighestPriorityJob(\mathbf{T}.\mathbf{J} \setminus G.\mathbf{J}', SPT)$
3: $\mathbf{J}'' \leftarrow G.\mathbf{J}' \cup \{J^{\text{curr}}\}$
4: $PT'' \leftarrow (TopologicalSort(G) \frown J^{\text{curr}})$
5: $\mathbf{T}'' \leftarrow MaximalSubgraph(\mathbf{T}, \mathbf{J}'')$
6: $\vdash \leftarrow SimulateListSchedule(\text{LO},\ \mathbf{T}'',\ PT'')$
7: $\overset{\mathbf{J}''}{\sim} \leftarrow EstimateInterference(\text{LO},\ \mathbf{T}'')$
8: $G.\mathbf{J}' \leftarrow \mathbf{J}''$
9: **for** *all trees* $ST \in G$ **do**
10: **if** $\chi(J^{\text{curr}}) = \text{LO}$ **then**
11: **if** $\exists\ J' \in ST\colon\ J' \vdash J^{\text{curr}}\ \vee\ J' \rightarrow J^{\text{curr}}$ **then**
12: $ConnectAsRoot(ST, J^{\text{curr}})$
13: **end if**
14: **else**
15: **if** $\exists\ J' \in ST\colon\ J' \overset{\mathbf{J}'}{\sim} J^{\text{curr}}\ \vee\ J' \rightarrow J^{\text{curr}}$ **then**
16: $ConnectAsRoot(ST, J^{\text{curr}})$
17: **end if**
18: **end if**
19: **end for**
20: **if** $\chi(J^{\text{curr}}) = \text{HI}$ **then** $PullUp(J^{\text{curr}}, G, \mathbf{T}, SPT)$
21: **end while**

Figure 3.17: The algorithm for computing priority tree in MCPI

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 3.18: The pull-up subroutine

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 (3.6). In addition we do the same for the subtrees containing task-graph predecessors of J^{curr}, to ensure precedence compliance, as defined by Equation (2.3).

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.

Relation $\vdash$ is evaluated by simulation. In this simulation we assume that the selected job has the lowest priority and the other priorities are determined by P-DAG G. Recall that notation $PT \frown J$ means concatenation of job J in the lowest-priority (*i.e.,* the last) position in the priority table PT. Relation $\overset{\mathbf{J}'}{\sim}$ is evaluated using the single-processor busy interval obtained from makespan algorithm (see Theorem 3.2.22), whereas other more accurate estimation procedures are possible.

The reason why instead of blocking we use a larger relation in the case of HI jobs is to ensure safety of further modifications of G. These modifications 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 initial priority order. Notice that if this subroutine were not called, the algorithm would just generate a P-DAG of the initial priority table SPT.

Procedure *PullUp* is described in pseudocode in Figure 3.18. 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.

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: $PT \leftarrow (TopologicalSort(G) \frown (SPT \mid\prec J))$
6: $allDeadlinesMet \leftarrow SimulateListSchedule(\text{LO},\ \mathbf{T},\ PT)$
7: **return** *allDeadlinesMet*

Figure 3.19: The subroutine for checking the feasibility of a priority swap

Procedure $LOpredecessors(J, G)$ returns for node J the set of its direct P-DAG predecessors[5] of LO criticality: $\{J_s \mid J_s \rhd J,\ \chi_s = \text{LO}\}$. At each step in Figure 3.18 we pick the least priority P-DAG predecessor from the working set $LOpredecessors(J, G) \setminus DONE$, where $DONE$ is a set that keeps track of the jobs 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 and 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 3.19, 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 as all those that have less SPT-priority than the current HI job J, therefore we denote the 'trailer' part of SPT 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 'swap' transformation on graph G, defined as follows:

Definition 3.4.1 (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 possibly 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:

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

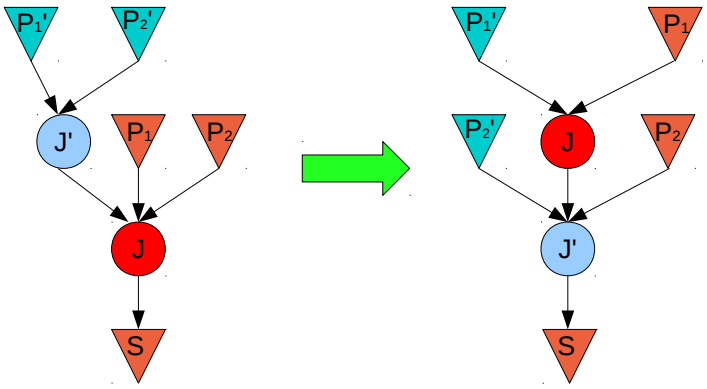


Figure 3.20: The effect of a Swap.

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 Figure 3.20 for $J_{\text{HI}} = J$ and $J_{\text{LO}} = J'$. In the original P-DAG the red triangle marked with S represents the P-DAG successors of J, while the triangles marked with P_1, P_2 and P_1', P_2' are, respectively the P-DAG 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 either potentially interferes with J or its task-graph predecessor 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.

Example 3.4.2. *Consider again the instance and the priority table of Example 3.2.15. Let us apply MCPI on them. The table PT is already precedence compliant, so* Dependency-ComplianceTransform *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 Figure 3.7(a), where it is easy to check that no job misses its deadline.*

Then we apply subroutine MCPI_PDAG*. The graph G obtained in the first few iterations before the first PullUp is illustrated in the left side of Figure 3.21. 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 and we have $s3 \overset{\mathbf{J'}}{\sim} s4$.*

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 Figure 3.21. 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:*

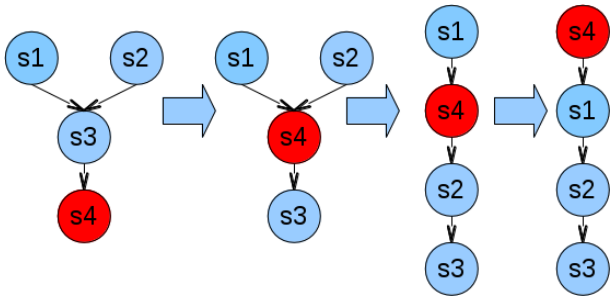


Figure 3.21: The effect of subroutine *PullUp* on job $s4$.

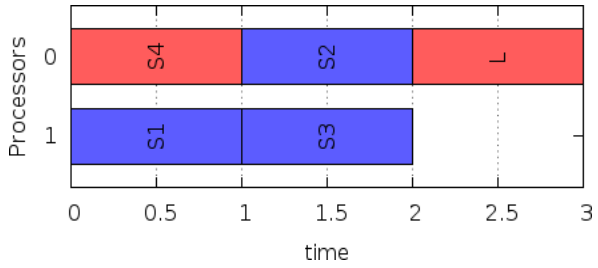
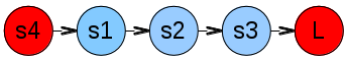


Figure 3.22: The schedule obtained by MCPI in Example 3.4.2.



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 Figure 3.22. The reader may easily verify that using the initial priority assignment, whose LO-mode table is shown in Figure 3.7(a), will fail if instead of following the LO scenario job* $s4$ *will continue execution until* $C(HI) = 3$ *time units (which, in fact, happens in scenario HI-*$s4$*). At the same time, using the table generated by MCPI , which results in the LO-mode behavior shown in Figure 3.22,* $s4$*, having the highest priority, starts earlier and would meet its deadline even in this scenario.*

Below we give two theoretical results for MCPI.

Lemma 3.4.3. *The Graph produced by MCPI_PDAG procedure is a forest P-DAG.*

Proof. MCPI_PDAG proceeds similarly to *Forest_PDAG*, whose correctness was already shown by Theorem 3.2.16. There are only two differences:

1. more edges are added at each step
2. the *swap* modification is performed

Since, by Lemma 3.2.6, with extra edges added, G still remains a P-DAG, we observe, by Theorem 3.2.16, that *MCPI_PDAG* ensures that G is a P-DAG at least until the first swap.

To complete the proof we have to show that after a *swap* operation G remains to be a P-DAG. Let us consider pulling up job J^s and let $TreeSwap(J^s, J^p_k, G)$ be the k-th swap. Notice that J^s is a HI job, and before the swap root of some tree T. By construction, all trees that are not connected to J^s contains only job that are not in potential interference relation with it, so their execution will not be influenced by any change in T. Thus, without loss of generality, we can assume that G is composed of only one tree (*i.e.*, $G = T$).

After the first swap, G is still a tree, such that J^p_1 is the new root After multiple swaps, the situation will be as illustrated in Figure 3.23. On the left side of the figure we have a

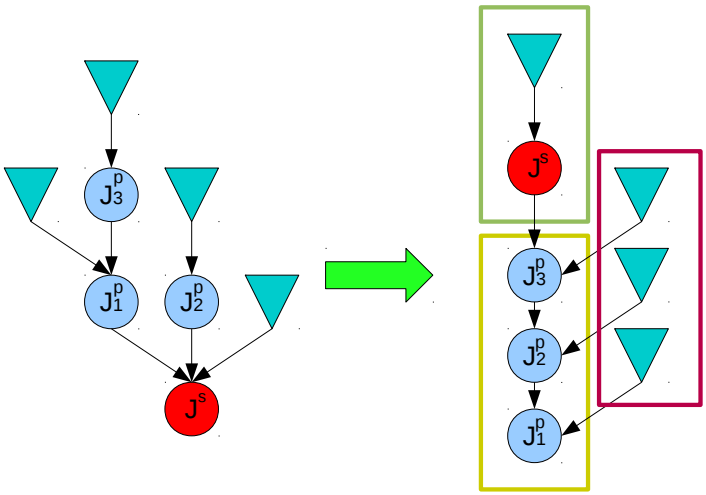


Figure 3.23: The effect of multiple Swaps, $k = 3$.

tree with HI job J^s as root. After swapping J^s with J^p_1,J^p_2 and J^p_3 (in this order), we obtain the tree on the right side. We can distinguish three areas in the tree: a tree composed of a chain of LO jobs in the lower part (inside the yellow box), connected to a subtree that has J^s as root (green box) and some subtrees connected to the LO jobs in the chain that are not in potential interference relation with J^s (red box).

Let us assume by contradiction that after the k-th swap – $TreeSwap(J^s, J^p_k, G)$ – the resulting graph $G' = (\mathbf{J}', \triangleright)$ is no longer a P-DAG. By Lemma 3.2.12 and the contradicting hypothesis, we have that G' can generate a table PT' that leads to a schedule $\mathcal{S}$ such that:

$$\exists J', J'' \colon J' \vdash_{\mathcal{S}} J'' \ \wedge \ J' \not\triangleright^* J''$$

For $TreeSwap(J^s, J^p_k, G)$ all the possible $J' \vdash J''$ relations that were not present before the swap are such that either $J'' = J^p_k$ or $J^p_k \rightarrow^* J''$. This is because, by lowering J^p_k priority (*i.e.,* shifting forward its execution), it might enter in the execution window of another job and get blocked by it. The same holds for its successors in $\mathbf{T}$.

For $J'' = J^p_k$, we can then rewrite our contradicting hypothesis as follows:

$$\exists J' \colon J' \vdash_{\mathcal{S}} J^p_k \ \wedge \ J' \not\triangleright^* J^p_k$$

After the swap, J^p_k is the root of a subtree ST. So $\forall J \in ST, \ J \triangleright^* J^p_k$. All jobs J' that are not in ST are either the chain below J^p_k (yellow box) or in the side subtrees branched into them, present in the red box. For the jobs J' in these subtrees we have we can show that: $J' \nvdash_{\mathcal{S}} J^p_k$. This is so because by properties of swap they are not in potential interference relation with J^s and hence they also are not in potential interference relation with J^p_k, as J^s is swapped only with jobs in the same potential interference equivalence class. For jobs $J' = J^p_1, J^p_2, \ldots J^p_{k-1}$ in the yellow box we have that they have lower priority than J^p_k and hence: $J' \nvdash_{\mathcal{S}} J^p_k$.

Let us now consider jobs J'' such that $J^p_k \rightarrow^* J''$. An invariant of our algorithm is precedence compliance, *i.e.,* $J^p_k \rightarrow^* J'' \Rightarrow J^p_k \triangleright^* J''$. This means that all such J'' are among $J^p_1, J^p_2, \ldots J^p_{k-1}$ in the chain below J^p_k. The same reasoning as in the previous case holds. □

Theorem 3.4.4. *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*

$$O(Ek^2 + k^3(\log k + m)) \tag{3.10}$$

Proof. One of the main contributions to the computational complexity of the algorithm is given by the high number of list schedule simulations. The complexity of one simulation is, by Lemma A.2:

$$O(E + k(\log k + m)) \tag{3.11}$$

Let us now analyze the algorithm of Figure 2.4 line by line. Routine *DependencyComplianceTransform* has a complexity of $O(k^2)$. This is because we run through the linked list of all jobs, and for each of them we move all the predecessors that have a lower priority in front of the current job, and the maximum number of predecessors for each job is $O(k)$. *CheckLOscenarioSchedulability* does one simulation, thus it has a complexity of (3.11). *MCPI_PDAG* gives the highest contribution, and its complexity will be discussed later. *TopologicalSort* has complexity $O(k)$ [CSRL01]. Finally, *anyScenarioFailure* does $O(k)$ simulations, and thus its complexity is $O(k(E + k(\log k + m)))$.

Let us now analyze routine *MCPI_PDAG*. This is a recursive subroutine that is called exactly k times. This subroutine, after some $O(1)$ operations, performs a simulation, which gives a total contribution of (3.11). Then for each subtree a *ConnectAsRoot* operation is performed. One such operation has a linear complexity in jobs, because we have to find the root of a subtree. There are $O(k)$ subtrees, thus this operation yields a total contribution of $O(k^3)$. Finally we have to analyze the complexity of *PullUp* (Figure 3.18). In this subroutine there is a while loop that is executed once for each LO predecessor of the current job, thus a $O(k)$ number of times inside *PullUp* and $O(k^2)$ number of times per one one execution of *MCPI*. All the operations performed in the subroutine are $O(1)$ except for *CanSwap*, which performs a simulation and thus it has complexity (3.11). Thus the *CanSwap* subroutine gives the main total contribution to the complexity of the algorithm, executing in total $O(k^2)$ simulations of complexity (3.11), which gives the result given in (3.10).

□

Notice that for large practical problem instances it can be expected that $m \ll k$, and also m is usually considered as a constant given by the platform. Also, even if in general $E = O(k^2)$, having a quadratic number of precedence edges is unrealistic in parallel programs, as this situation is likely to seriously restrict the possibility of parallel execution. If we consider only the cases where the number of job inputs and outputs is bounded by a constant then the number of precedence edges would grow linearly with the number of jobs. Under the assumptions mentioned here, the complexity can be assumed as follows:

$$O(k^3 \log k)$$

3.4.3 Support Algorithm

By default we assume that the support algorithm is *EDF with modified deadlines and density threshold* (EDF-DS). First we compute graphs $\mathbf{T}_{\text{MIX}}$ and $\mathbf{T}_{\text{HI}}$ (see Section 2.2.3), and we will use them to compute, respectively, PT_{LO} and PT_{HI} as explained below. In table PT_χ the priorities are assigned in the EDF way (considering ALAP deadline) if the job density $\delta_j(\chi)$ is smaller than the (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 [DB11].

To indicate explicitly that we are using a support algorithm ALG, we use the notation MCEDF(ALG). Thus we will use notation MCPI(EDF-DS) to indicate the use of MCPI with the above described support algorithm, while we will use MCPI(EDF) if we use the unmodified (global) EDF policy.

3.4.4 Predictable Online Policy for MCPI

The online policy should be predictable per scenario, as discussed in Section 2.1.2. Recall that list scheduling is, in general, non-predictable. Therefore, online we execute a predictable policy that behaves the same way as list scheduling in basic scenarios. Recall that offline we check schedulability by simulating all basic scenarios. For each of them we record all jobs start times in a table and provide the table to the online policy. Online, we keep track of the current basic scenario, assuming LO when in LO mode and HI-J_h when job J_h causes a switch to HI mode. Online we assume that jobs arrive not at their nominal arrival times, but at their offline-computed schedule start times specified in the table of the current scenario. The modified arrival times ensure that precedences are satisfied. Therefore, while preserving the precedence constraints, instead of list scheduling our online policy uses the default classical global fixed priority scheduling, which is known to be predictable.

An alternative online scheduling policy is a time-triggered one where there are two tables (for LO and HI modes). How to construct such a policy is described in Chapter 4.

3.5 Common properties of MCEDF and MCPI

In this section we give some theoretical properties of MCEDF and MCPI for independent jobs on single processor. The main results are the optimality of MCEDF and MCPI over all scheduling algorithm where HI jobs are in relative EDF order. Also MCEDF and MCPI are equivalent. Note that because all these algorithms use LO-mode schedules to construct the priority tables, under the 'scheduling' we always mean the LO-mode scheduling unless mentioned otherwise.

In this section we assume that both algorithms use the same EDF-compliant support priority table SPT and that the jobs are independent.

The following lemma establishes for MCPI a property that is true for MCEDF by construction.

Lemma 3.5.1. *In MCPI, as in MCEDF, each tree of the P-DAG contains jobs from one and only one busy interval.*

Proof. (sketch) For MCPI, we argue that this property is true by demonstrating that at each basic step of the algorithm: the initial connection of a new job to the P-DAG and the swapping. When a LO-job is connected to a P-DAG, the criterion is to connect it to the trees that block the given job when it has the least priority. Since they block the given job then they must be in the same busy interval. When a HI job is initially connected, the property holds by construction, as on single processor MCPI should always use splitting into busy intervals to evaluate $\overset{\mathbf{J}'}{\sim}$.

Now consider the swapping. After the swapping, the current HI job forms one same busy interval with the subtrees connected to it by the same argument as the ones we used for the initial connection. The LO job which was swapped forms one busy interval with the current HI job tree and other trees that are plugged to it by observation that this was already the case before the swap and the busy intervals do not change when priority assignment changes. □

Lemma 3.5.2 (Per-criticality EDF Compliance of P-DAG)**.** *In the P-DAG G of MCPI(EDF) or MCEDF, consider any P-DAG path between two jobs of the same criticality: $J_i \rhd^* J_j$. This path can only join J_i and J_j in the direction that is compliant with their relative priority in SPT. Mathematically:*

$$\forall i, j \;.\; \chi_i = \chi_j \;\;\wedge\;\; J_i \rhd^* J_j \;\;\Rightarrow\;\; J_i \succ_{SPT} J_j \tag{3.12}$$

Proof. (sketch) For MCEDF the Property (3.12) holds by construction, as it requires that J_j be the root of a subtree that contains J_i and *MCEDF_PDAG* assigns the least SPT-priority job of a given criticality as the root of the subtree.

For MCPI, as P-DAG construction evolves, the property can only be potentially broken by the swap operations. However, for criticality level HI it is not broken because we never swap two HI jobs. For criticality LO it can be only invalidated if a call to *CanSwap* returns 'false' and then a subsequent call returns 'true' in the same *PullUp* subroutine call. This is so because the LO jobs are evaluated for swapping in an order compliant with reverse SPT and the stem of swapped jobs forms a chain in the same order as the swapping is done. The job for which *CanSwap* would return 'false' would stay as P-DAG predecessor of the current HI job and the job with 'true' would become successor, thus forming a pair of LO jobs connected inconsistently with SPT. However, this cannot happen if SPT is EDF-compliant, as the first 'false' result from *CanSwap* will be necessarily followed by other 'false' results. To show this, recall that by Lemma 3.5.1 the HI job forms one busy interval (τ_1, τ_2) with its subtree. When *CanSwap* evaluates different LO jobs for the least priority it evaluates for the possibility that the swapped job can terminate at time τ_2 while meeting its deadline. The jobs are evaluated in reverse EDF order, so the jobs with the larger deadline will be evaluated first. Therefore, if a job misses its deadline at time τ_2 then the other jobs will fail as well. □

By the above lemma, for MCEDF and MCPI(EDF), it is always possible to find a topological sort of graph G such that the resulting priority table satisfies the following property:

Definition 3.5.3 (HI-criticality EDF Compliance of Priority Table)**.** *Given an EDF-compliant SPT priority table, any LO-mode priority table PT is said to be HI-criticality EDF-compliant according to table SPT if the HI jobs appear in PT in the same order as in SPT, that is:*

$$\forall i, j \;.\; \chi_i = \chi_j = HI \;\wedge\;\; J_i \succ_{PT} J_j \;\;\Rightarrow\;\; J_i \succ_{SPT} J_j \;\wedge\; D_i \leq D_j$$

Consider a problem instance $\mathbf{J}$ where h jobs are HI-critical. We can partition an EDF-compliant priority table generated by MCEDF/MCPI(EDF) into the following sequence of job sets:

$$PT \;:\; \mathbf{J}_1^{\mathrm{LO}} \succ_{PT} \{J_1^{\mathrm{HI}}\} \succ_{PT} \mathbf{J}_2^{\mathrm{LO}} \succ_{PT} \{J_2^{\mathrm{HI}}\} \succ_{PT} \ldots \mathbf{J}_h^{\mathrm{LO}} \succ_{PT} \{J_h^{\mathrm{HI}}\} \succ_{PT} \mathbf{J}_{h+1}^{\mathrm{LO}} \tag{3.13a}$$

$$\text{HI jobs} \;:\; J_1^{\mathrm{HI}} \succ_{SPT} J_2^{\mathrm{HI}} \succ_{SPT} \ldots J_{h-1}^{\mathrm{HI}} \succ_{SPT} J_h^{\mathrm{HI}} \tag{3.13b}$$

where subscript l and h denote LO and HI jobs and relation '$\succ$' between two job sets means that any job in the first set has a higher priority than any job in the second set.

Let us denote by $\rhd^{*\text{LO}}$ a relation between two jobs that are joined in the P-DAG by a path that may have only LO jobs as intermediate nodes. The following is trivial:

Lemma 3.5.4. *There always exists a priority table PT obtained from a topological sort of P-DAG G of MCEDF or MCPI(EDF) that has the structure shown in Formulas (3.13) where, in addition, the LO job sets $\mathbf{J}_i^{LO}$ are defined as the sets of LO jobs related to J_i^{HI} by $\rhd^{*LO}$:*

$$\text{for } \; i = 1..h \; . \; \mathbf{J}_i^{LO} = \{J_j \mid \chi_j = LO \; \wedge \; J_j \rhd^{*LO} J_i^{HI}\} \tag{3.14a}$$

$$\mathbf{J}_{h+1}^{LO} = \{J_j \mid \chi_j = LO \; \wedge \; \nexists i : J_j \rhd^{*LO} J_i^{HI}\} \tag{3.14b}$$

Definition 3.5.5 (A Least LO-Priority Table)**.** *Given a P-DAG G that is per-criticality compliant to support priority table SPT. A priority table obtained from graph G that can be partitioned as shown in Formulas (3.13) and (3.14) is called a least LO-priority table.*

The reason to give a priority table this name is that such a table puts each LO job at the highest-i (and hence also the least-priority) set $\mathbf{J}_i^{\text{LO}}$. The following lemma states that one cannot give any LO job even less priority *w.r.t.* a HI job.

Lemma 3.5.6. *Let $\mathbf{J}$ be a problem instance where MCEDF or MCPI(EDF) generates a P-DAG based on an EDF-compliant SPT, let $\mathbf{J}_i^{LO}$ characterize its least LO-priority table. Let PT' be some HI-criticality SPT-compliant priority table where some LO jobs in some job sets $\mathbf{J}_i^{LO}$ 'violate the least LO priority constraint' in the sense that they have less priority than the corresponding HI job J_i^{HI}. Then at least one of such jobs will miss its deadline.*

Proof. Let i' be the smallest-index i of the job sets $\mathbf{J}_i^{\text{LO}}$ that contain LO jobs that in table PT' 'violate' the least priority constraint. Let J_j be the least-priority violating job from the respective set $\mathbf{J}_{i'}^{\text{LO}}$. Let us show that it will miss its deadline. The part of the priority table PT' that contains jobs of priority higher or equal to J_j can be represented by (dropping the curly braces for singleton sets):

$$PT' \mid_{\succeq j} \; : \; \mathbf{J'}_1 \succ J_1^{\text{HI}} \succ \ldots \succ \mathbf{J'}_{i'-1} \succ J_{i'-1}^{\text{HI}} \succ \mathbf{J'}_{i'} \succ J_{i'}^{\text{HI}} \succ \mathbf{J''}_{i'} \succ J_j$$

For some LO-jobs sets $\mathbf{J}'_1, \mathbf{J}'_2, \ldots, \mathbf{J}'_{i'}$ and (possibly mixed) job set $\mathbf{J''}_{i'}$. Observing that in single processor scheduling the relative priority order of higher-priority jobs does not matter for the least priority job, let us reorder the priority of the last HI job and obtain table PT'' that results in *equal* termination time for job J_j:

$$PT'' \; : \; \mathbf{J'}_1 \succ J_1^{\text{HI}} \succ \ldots \succ \mathbf{J'}_{i'-1} \succ J_{i'-1}^{\text{HI}} \succ \mathbf{J'}_{i'} \succ \mathbf{J''}_{i'} \succ J_{i'}^{\text{HI}} \succ J_j$$

From the definition of violating jobs and from the assumption that the sets $\mathbf{J}_m^{\text{LO}}$ for $m \leq i'-1$ contain no violating jobs (i' being the lowest 'violating' index) we have:

$$\text{for } 1 \leq m \leq i'-1 \quad : \quad \bigcup_{i=1}^{m} \mathbf{J}_i^{\text{LO}} \subseteq \bigcup_{i=1}^{m} \mathbf{J'}_i$$

Also because, by our assumptions, J_j is the least priority violating job in set i' we have that $\mathbf{J''}_{i'}$ contains all other violating jobs from $\mathbf{J}_{i'}^{\text{LO}}$, and hence:

$$\bigcup_{i=1}^{i'} \mathbf{J}_i^{\text{LO}} \subseteq \left(\bigcup_{i=1}^{i'} \mathbf{J'}_i \cup \mathbf{J''}_{i'} \cup \{J_j\} \right)$$

By the job-set inclusion relation above, the following priority table PT''' when compared to PT'' has at most the same but possibly *less* jobs of higher-priority than J_j:

$$PT''' \ : \ \mathbf{J}_1^{\text{LO}} \succ J_1^{\text{HI}} \succ \ldots \succ \mathbf{J}_{i'-1}^{\text{LO}} \succ J_{i'-1}^{\text{HI}} \succ \left(\mathbf{J}_{i'}^{\text{LO}} \setminus J_j\right) \succ J_{i'}^{\text{HI}} \succ J_j$$

By properties of MCEDF resp. MCPI(EDF) we have that job $J_{i'}^{\text{HI}}$ forms one busy interval BI with the higher subtrees connected to it and by observation that $J_j \rhd^{*\text{LO}} J_{i'}^{\text{HI}}$ we have that J_j also belongs to the same busy interval BI. Now observe that the reason why MCEDF resp. MCPI(EDF) assigned $J_{i'}^{\text{HI}}$ the least priority in the given BI is because the highest-deadline LO job belonging to the same interval would miss the deadline. J_j, by construction, cannot have a higher deadline, so it should also miss its deadline as the least-priority job in BI. Therefore it will also miss its deadline in PT''', and hence also in PT'' and PT'. □

We can now prove the following:

Theorem 3.5.7. *For a given EDF-compliant SPT, MCEDF and MCPI(EDF) are optimal among the FPM algorithms that are HI-criticality EDF-compliant according to SPT.*

Proof. (sketch) Consider an instance $\mathbf{J}$ that is MC-Schedulable, The MCEDF and MCPI(EDF) algorithms will never fail in LO mode . This is so because, firstly, both algorithms are based on iterative improvement of an EDF table, which is optimal in the LO mode. Secondly, at every improvement step the LO-schedulability of the problem instance is preserved as an invariant. This leads to two important conclusions:

1. The only possible schedulability failure that MCEDF or MCPI(EDF) can have is when a HI job that misses its deadline in a HI scenario.
2. For MC-schedulable instance, even if we see the worst case presented in Point 1, both algorithms manage to construct a LO-schedulable P-DAG that satisfies all lemma's and properties presented in this section.

Consider an instance $\mathbf{J}$ with h HI jobs. Suppose by contradiction to the theorem statement that MCEDF (resp. MCPI(EDF)) fail to produce a feasible schedule due to a failure in a HI scenario, whereas the optimal EDF-compliant algorithm can. By lemma's above, we can present the (failing) solution of both algorithms as shown in Formulas (3.13) and (3.14).

By our assumptions the optimal priority table PT' is also HI-criticality EDF-compliant according to SPT and hence it can also be presented in a similar form:

$$PT' \ : \ \mathbf{J'}_1 \succ \{J_1^{\text{HI}}\} \succ \mathbf{J'}_2 \succ \{J_2^{\text{HI}}\} \succ \ldots \ \mathbf{J'}_h \succ \{J_h^{\text{HI}}\} \succ \mathbf{J'}_{h+1}$$

By Lemma 3.5.6 we should have:

$$\text{for } 1 \leq m \leq h \quad : \quad \bigcup_{i=1}^{m} \mathbf{J}_i^{\text{LO}} \subseteq \bigcup_{i=1}^{m} \mathbf{J'}_i$$

where $\mathbf{J}_m^{\text{LO}}$ are the least LO-priority job sets of MCEDF resp. MCPI(EDF).

This means that, compared to MCEDF or MCPI(EDF), for every HI job J_m^{HI} the optimal algorithm puts at least the same but possibly a larger set of jobs as higher-priority *w.r.t.* to J_m^{HI}. On a single processor this can only reduce the progress made by each HI jobs up to any given point in time in the LO mode. Therefore, after a mode switch, all the HI jobs in the optimal algorithm will have at least the same or possibly more workload to terminate than in MCEDF or MCPI(EDF). Therefore, if the latter would fail in some HI scenario all the more so the former would also fail in the same scenario, therefore the optimal algorithm would fail and thus we have a contradiction. □

The next theorem follows as a corollary of Theorem 3.5.7:

Theorem 3.5.8. *When using the same EDF-compatible SPT table MCEDF and MCPI(EDF) are equivalent.*

Note that, despite equivalence, MCEDF has a lower computational complexity than MCPI, which has $O(k^3 \log k)$ (where k is the number of jobs). Intuitively, this is so because MCEDF is 'specialized' for the single-processor scheduling problems, which is inherently 'simpler' than the multiprocessor ones, handled by MCPI.

It can be easily shown that OCBP can also be restricted to be HI-criticality EDF-compliant, thus Theorem 3.3.5 can be seen as a corollary of Theorem 3.5.7.

3.6 Implementation and Experiments

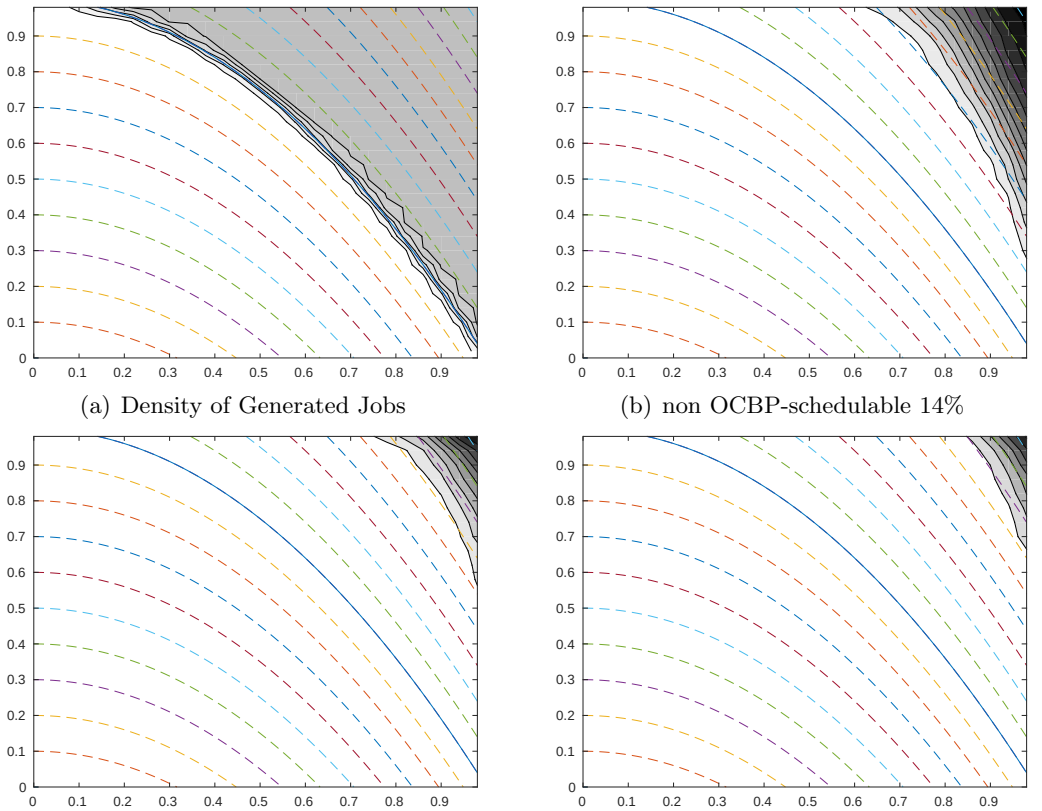
We evaluated the schedulability performance of MCEDF and MCPI in experiments with randomly generated job instances. The instance size was restricted due to the computation delays of job generation algorithm and our intention to evaluate a large number of points. with integer timing parameters, simulating CPU clock cycle count of some imaginary machine. Every job instance was generated for a target LO and HI load or stress pair.

The method to generate a job instance was as follows. First we randomly generated a tentative instance, not paying attention to the target loads. This was done by repeatedly generating a new sporadic task, i.e. sequence of jobs arriving one after another at random arrival intervals. For every job, both the job deadline and the arrival interval were uniformly distributed in a range 5K-25K (kilocycles), and the job's criticality level was set to HI (*i.e.,* $\chi = \text{HI}$) with a probability 50%. Every sporadic task produced just enough jobs to fill a random interval from 0 to a bound in range 15K-100K. The WCET $C_j(\text{LO})$ of each job was uniformly distributed between 0 and the relative deadline, each HI job had a $C_j(\text{HI})$ obtained by scaling the value $C_j(\text{LO})$ by a random factor [1..1000]. For MCEDF new sporadic tasks were invoked until all tasks together have produced more than 20 jobs, and then jobs were randomly removed until only 20 remained. For MCPI, instead of 20, we produced 30, 60 or 120 jobs in a similar way for processor counts $m = 2, 4, 8$. To finalize the job instance generation, the algorithm calculated the loads of the tentative instance and scaled the execution times to obtain the target load in the final instance.

When scaling the loads, we took care that when $C_j(HI)$ would have to be scaled below $C_j(LO)$, it is instead set to $C_j(LO)$. This could result in imprecise final $Load_{\text{HI}}$. As a result, there was a load scaling problem, as the scaling sometimes failed to approximate the target load with the specified precision. In this case we cancelled the generated instance and made another attempt to generate it until multiple attempts produced no satisfactory load scaling result within a timeout. Due to this, and due to high complexity of load calculations the job generation process itself took a considerable time in the experiments.

3.6.1 MCEDF

We ran multiple job generation experiments, ranging each target of $Load_{\text{LO}}$ and $Load_{\text{HI}}$ from 0.0025 to 1 with step 0.0025. Per each target, ten experiments were run, generating the points lying near the target with tolerance 1%. We only selected the 'overloaded' targets *i.e.,* those lying at or above the parabola $Load^2_{\text{LO}}(\mathbf{T}) + Load_{\text{HI}}(\mathbf{T}) = 1$, yielding instances where OCBP could potentially fail. By looking at the loads below 1 we compare both OCBP



(a) Density of Generated Jobs

(b) non OCBP-schedulable 14%

(c) non MCEDF-schedulable 2.1%

(d) non MCEDF-schedulable after split 1.25%

Figure 3.24: The contour graphs of random instances; the horizontal axis is $Load_{\mathrm{LO}}$, the vertical is $Load_{\mathrm{HI}}$.

and MCEDF to the clairvoyant scheduler, which can schedule all such points and which gives an upper bound on the best scheduling performance. Fig 3.24(a) gives the contour graph of the density of the generated points in grayscale. The grid follows the parabolic lines of equal $Load^2_{\text{LO}}(\mathbf{T}) + Load_{\text{HI}}(\mathbf{T})$. The total number of trials was 537460.

Around 14% (75203) of points showed failure for OCBP. In those 14%, roughly 2.1% (11316) were not schedulable by MCEDF as well, whereas 11.9% (63887) were schedulable by MCEDF. Thus, MCEDF proved to reduce the set of non-schedulable instances by a factor 6-7. The density distributions in Figure 3.24 suggest that MCEDF is less sensitive to high loads.

For the 2.1% (11316) non-MCEDF schedulable jobs we ran additional experiments. We considered *splitting* (see Section 3.3.4), a theoretical transformation[6] of a job instance into a new instance where a HI job is equally divided into a certain number (called *split factor*) of equal smaller jobs, whose total execution times $C_j(LO)$ and $C_j(HI)$ add up to that of the original job. Splitting reduces the uncertainty and $Load_{\text{MIX}}$. Therefore, for mode-switched policies, such as MCEDF, the splitting can translate an unschedulable instance into a schedulable one. An infinitely large splitting of all HI jobs can bring the optimality of a mode-switched policy infinitely closer to that of the clairvoyant scheduling. For some instances, a finite splitting is enough to equate the clairvoyant scheduling. Mode-ignorant policies, such as OCBP, cannot take any advantage of splitting by construction.

We split all HI jobs by factors $2, 3$, and 4. This kept the load the same but reduced the WCET uncertainty. After splitting the instances remained to be non-OCBP schedulable (as OCBP cannot take advantage of less uncertainty) but the number of non-MCEDF schedulable instances has reduced, coming to 1.25% (6735). So if we can accept this load-preserving transformation, we go from 14% non-schedulability of OCBP to the 1.25% non-schedulability of MCEDF. Note that 0.85% (4581) were gained due to the splitting, whereby in the most of cases, 0.55% (2961), split factor 2 was sufficient. So assuming that in practice we can split the HI jobs into a few sub-jobs such that both WCET values scale, then we can in many cases obtain a schedulable instance. That the fragmentation of jobs would preserve the same total WCET is likely to be an overly optimistic assumption for the WCET tools, but still doing this is worth a try.

We also performed some experiments to evaluate the computation times of both algorithms, implemented using the same software library. Every point was obtained as the average computation time for 20 different randomly generated instances with $Load_{\text{LO}} = Load_{\text{HI}} = 0.8$. The results are shown in Figure 3.25. They confirm our expectation of almost one order of magnitude of difference, as we estimate the best direct implementation of OCBP to be $O(K^3)$ and the best MCEDF to be $O(K^2)$ for k jobs, according to Lemma 3.3.3.

3.6.2 MCPI

We evaluated the schedulability performance of MCPI comparing it with those the performance of its support algorithm. We restricted our experiments to "hard" task graphs, *i.e.,* those satisfying the following formula:

$$Stress_{\text{LO}}(\mathbf{T}) + Stress_{\text{HI}}(\mathbf{T}) \geq \sigma_s \tag{3.15}$$

The reason of this choice is that task graphs under that line are relatively easy to schedule. We ran multiple job generation experiments, ranging the target of $Stress_{\text{LO}}$ and $Stress_{\text{HI}}$ in

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

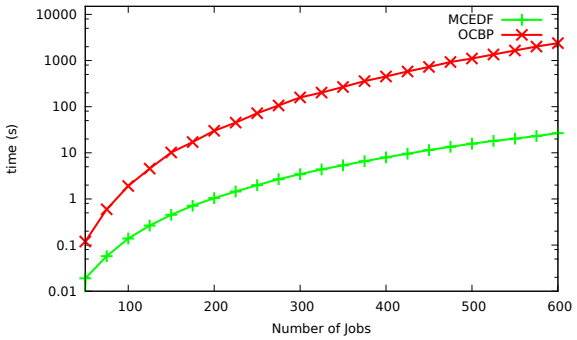


Figure 3.25: The measured computation times of OCBP and MCEDF

m	jobs	arcs	step	δ	σ_s	instances
2	30	20	0.005	0.01	3.2	128800
4	60	40	0.02	0.05	6	50500
8	120	80	0.05	0.125	12	31575

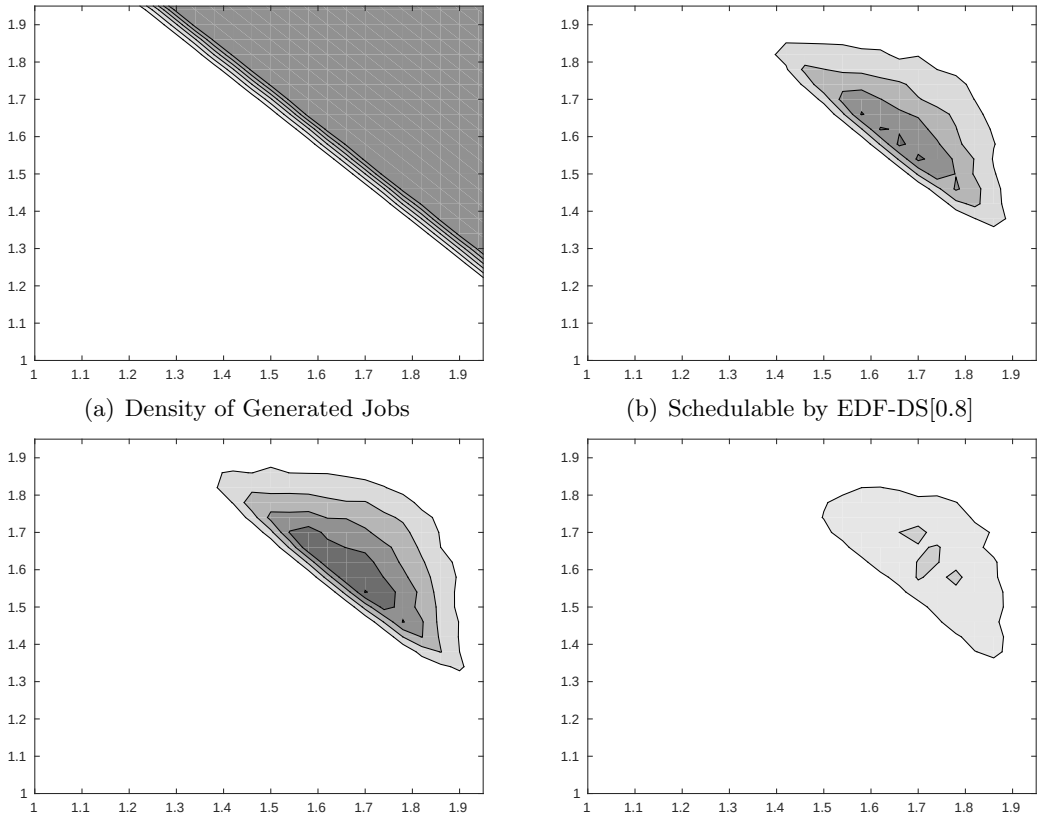
m	EDF	EDF-DS	MCPI(EDF)	MCPI(EDF-DS)	diff(%)	diff-DS(%)
2	20924	21023	27375	27467	**30.83%**	**30.65 %**
4	6839	6887	8263	8310	**20.82%**	**20.66 %**
8	3065	3082	3521	3538	**14.88%**	**14.80%**

Table 3.1: Experimental results for MCPI.

the area defined by (3.15) with a fixed step s. Per each target, ten experiments were run, generating the points lying near the target with a certain tolerance δ. All points satisfied $Stress_{\text{MIX}} \leq m$. The result of the experiments are shown in Table 3.1. We ran experiments for 2, 4 and 8 processors. For each generated task graph, we checked the schedulability of EDF, EDF-DS, MCPI(EDF), MCPI(EDF-DS). For EDF-DS we used a threshold of 0.8. All algorithms were applied using the FPM scheduling policy, the ALAP and ASAP arrivals and deadlines, based upon modified deadline D_{MIX} in the LO mode, as described in Section 3.4.3. From the result we can see that MCPI gives a big improvement in schedulability compared to the support algorithm, reaching a maximum of 30.83%.

Fig. 3.26 and Fig. 3.27 give the contour graph of the density of the generated points in grayscale, where black is the maximum value and white is 0. The horizontal axis is $Stress_{\text{LO}}$, the vertical is $Stress_{\text{HI}}$. Figures from Fig. 3.26(a) to Fig. 3.26(d) refer to the experiments made for 2 processors. In particular Fig. 3.26(a) shows the density of the generated task graphs, Fig. 3.26(b) shows the percentage of instances schedulable by EDF-DS among the generated ones. Likewise Fig. 3.26(c) shows the percentage of task graphs schedulable by MCPI (EDF-DS) and Fig. 3.26(d) shows the percentage of task graphs schedulable by MCPI (EDF-DS) and not schedulable by EDF-DS. As expected the schedulability decreases while the distance from the axis origin increase. Fig. 3.26(d) is particularly interesting, because it shows how MCPI increases the schedulability over the support algorithm when the load increases. Notice that approximately around point $(1.7, 1.7)$ the density is higher, suggesting that around this point MCPI is more effective.

Figures from Fig. 3.27(a) to Fig. 3.27(d) show respectively the same information of figures from Fig. 3.26(a) to Fig. 3.26(d), but referred to experiments on 4 processors. From those graph we have confirmation of the conclusions made above. Also in Fig. 3.27(d) we have an



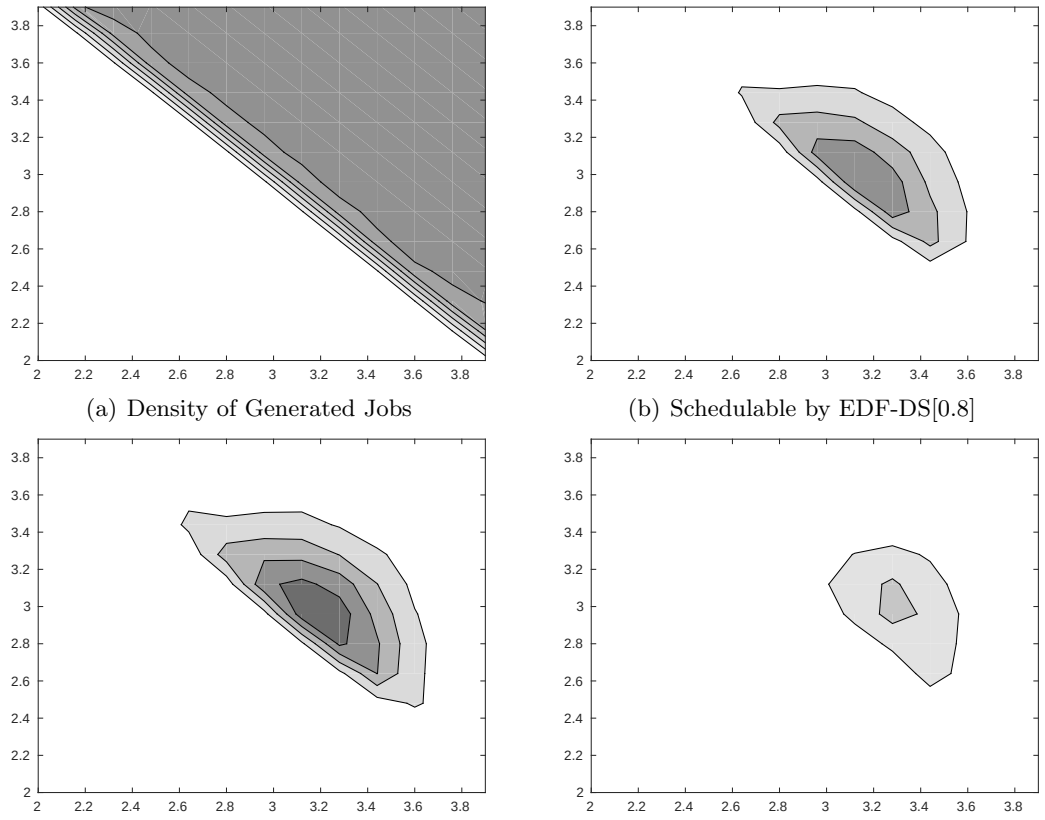
(a) Density of Generated Jobs

(b) Schedulable by EDF-DS[0.8]

(c) Schedulable by MCPI

(d) Schedulable by MCPI and not by EDF-DS[0.8]

Figure 3.26: The contour graphs of random task graphs for 2 processors. The horizontal axis is $Stress_{\mathrm{LO}}$, the vertical is $Stress_{\mathrm{HI}}$.



(a) Density of Generated Jobs

(b) Schedulable by EDF-DS[0.8]

(c) Schedulable by MCPI

(d) Schedulable by MCPI and not by EDF-DS[0.8]

Figure 3.27: The contour graphs of random task graphs for 4 processors. The horizontal axis is $Stress_{\mathrm{LO}}$, the vertical is $Stress_{\mathrm{HI}}$.

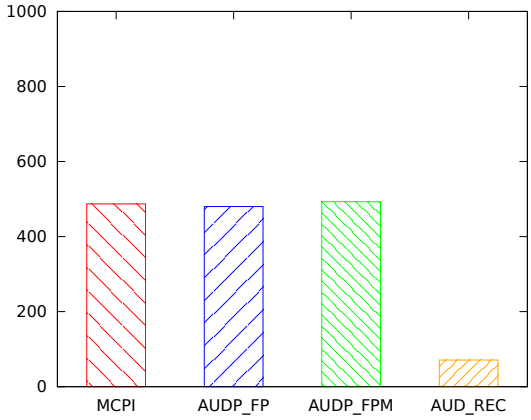


Figure 3.28: Comparison of MCPI with Audsley approach

area where MCPI is particularly effective, approximately around point $(3.3, 3.1)$.

Comparison with Audsley approach on multiple processors

One of the main motivations for this work was to overcome the limitations of Audsley approach, as discussed in Section 3.1.1. In this section we show through experiments that we successfully reached our goal, by comparing MCPI with Audsley approach.

First we compared with two 'ideal' implementations of Audsley approach, AUD_FP and AUD_FPM. In both of them we implemented an exact version of function $GetTerminationTime$ of Figure 3.1. To achieve this, to test if we can assign the least priority to a job J, we compute its worst case termination time by simulating all possible combinations of relative priority of jobs with higher priority. In AUD_FP we assumed to use a fixed priority scheduling, *i.e.,* we do not consider the possibility of dropping LO jobs. Thus when computing the worst case termination time we simulate a scenario where all jobs J_i run for $C_i(\text{HI})$. Note that this algorithm is equivalent to OCBP on single processors. Conversely in AUD_FPM we assumed to use a fixed priority per mode scheduling, *i.e.,* we drop LO jobs when a mode switch happens. In this case when computing the worst case termination time we simulate all basic scenarios for all possible combination of job priorities. Note that both algorithms have an exponential complexity and easily become computationally intractable.

Finally we also compared the results using a computationally implementation of Audsley approach, AUD_RC, which is based on the recursive formula given in Section 3.1.1.

We tested the algorithms on 1000 randomly generated instances of 8 jobs on 2 processors. The instances had a target $Stress_{\text{LO}} = Stress_{\text{HI}} = 1.8$. MCPI Solved 487 instances, AUD_FP 480, AUD_FPM 493 and AUD_RC only 71. The experimental results are shown in Figure 3.28. Even if the experiments were not extensive, due to the computational complexity of AUD_FP and AUD_FPM, we can state that MCPI behaves "as good as" a 'perfect' implementation of Audsley approach, which is computationally intractable, and outperforms a reasonably complex implementation. This proves the superiority of our P-DAG based 'bubble-sorting based' approach.

3.7 Chapter Summary

In this chapter we studied the problem of fixed priority scheduling of dual criticality systems. Our dissertation is focused on the finite set of job model, instead of the more general task model. We motivate this by the observation that reasoning on tasks is more complex, and thus the finite job set model, when applicable, potentially allows for better processor utilization. A major drawback of this approach is that it has to consider the hyperperiod, which can grow exponentially. However we think that this is passable for two main reasons. First, for sake of a easily manageable system analysis, we are interested in static scheduling (which is the topic of next chapter), and in this kind of scheduling considering a whole hyperperiod is not avoidable. Second, tasks in real-life systems are designed to avoid hyperperiod explosion.

We started the chapter by introducing one of the most common approaches to schedule mixed critical systems. This approach has, however, some limitations, which where discussed in Section 3.1.1. To overcome these limitations we introduced the *Fixed Priority per Mode* (FPM) scheduling policy, as an alternative to the usual *Fixed Priority* (FP). To properly model how jobs interact with each other, we introduced in Section 3.2 the *Priority-DAGs* (P-DAGs) and the *potential interference relation*. These formal tools are the foundation of our proposed scheduling algorithms.

In Section 3.3, we present the *Mixed Critical EDF* (MCEDF), an FPM algorithm for single processor, that was first presented in [3] (see Contributions Bibliography, page 123). This algorithm was compared with the *Own Criticality Based Priority* (OCBP) algorithm, which is the optimal FP algorithm. We formally proved the dominance of MCEDF over OCBP (and hence, over all FP algorithms) from which MCEDF inherits the optimal load characterization. Also MCEDF, unlike Audsley approach based algorithms, can exploit job splitting to improve schedulability.

The *Mixed Criticality Priority Improvement* (MCPI) algorithm was then discussed in Section 3.4. It is an FPM scheduling algorithm that supports multiprocessors and job dependencies. To the best of our knowledge, in literature there are no other multiprocessor algorithms that support dependencies, if not under very restrictive assumptions. This algorithm was first presented in [5].

In Section 3.5 we show some common properties of MCEDF and MCPI. We formally proved that, when applied on single processor with no dependencies, they both are optimal among the FPM policies that keep the priorities HI-critical job in relative EDF order. From the above property, we also deduce their equivalence. Those results are presented in this thesis for the first time.

We conclude the chapter with experimental results (Section 3.6) where we show noticeable improvement over OCBP with randomly generated instances and satisfying performances for MCPI. We conclude the experiment section by showing and empirical comparison between MCPI and Audsley approach, from where we can deduce that our approach is comparable to Audsley approach in "ideal" cases (*i.e.,* assuming an exact worst case termination time estimation) and by far outperforming a more computationally reasonable version of Audsley approach. The comparison with Audsley approach is also an unpublished result.

3.7.1 Future Work

As future work we are planning to extend MCEDF to precedence constraints and to prove, if possible, equivalence with MCPI even in this extended case. MCPI currently uses a weak

method to estimate the potential interference relation. Finding a more precise technique should increase its performance. Also, an alternative way of handling Dhall effect beyond density separation would be desirable.

Finally we would like to extend both algorithms to non-preemptive case and to multiple level of criticality. The latter is important for most standards, like DO-178B, but addressing it is not trivial.

Chapter 4

Time Triggered Policy

In this chapter we will discuss the problem of Time-Triggered (TT) scheduling policy for mixed critical systems. In contrast to priority-based algorithms discussed in the previous chapters, the Time Triggered 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.*.

4.1 introduction

In [BF11, Bar12] 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 [Bar13] the STTM scheduling was extended from single to multiple processors, though being restricted to the systems where all jobs have the same deadline. Formally, an ordinary (*i.e.,* non mixed-criticality aware) *Time Triggered* (TT) policy defines a static, pre-computed table that determines 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 [BF11] 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(LO) time units in **LO** table and each HI job C_j(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(HI) time units.

Producing such tables is not trivial, since, in general, as many tables as there are HI jobs are potentially needed to cover all the corner cases of a MC scheduling, the job-specific scenarios

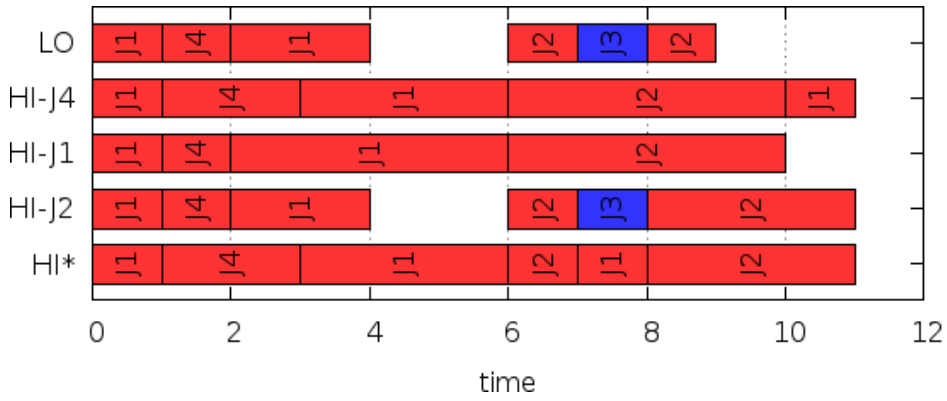


Figure 4.1: Basic scenarios and TT tables

(see Theorem 2.1.2). We recall these scenarios in the following:

Example 4.1.1. *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):

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

The schedules in the job-specific basic scenarios for this instance are shown in Figure 4.1. The first row represents the LO scenario, that can be used as ***LO*** *table for a STTM policy.*

The next three rows, represents the three HI basic scenarios. The reader may easily check that none of them may be used as a ***HI**** *table. For example, if we use the scenario HI-J4, and there is a switch from* ***LO*** *to* ***HI**** *table at time 9, job J2 will not have enough time to complete (it will have one other time unit reserved, but it needs 2).*

The last row shows a correct table ***HI****. *That table is obtained by the 'transformation' algorithm proposed in this chapter. Note that the correct table differs from all the job-specific scenarios, and presents unique behavior, as it runs job J1 in time interval* $(7, 8)$.

This chapter 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 always succeeds in converting FPM to STTM on single processor, which proves that STTM algorithms dominate FPM. The following gives a relation between the sets of schedulable instances for dual-critical problems:
Single Core

$$FP \subsetneq MCEDF \subsetneq FPM \subsetneq STTM = OPT$$

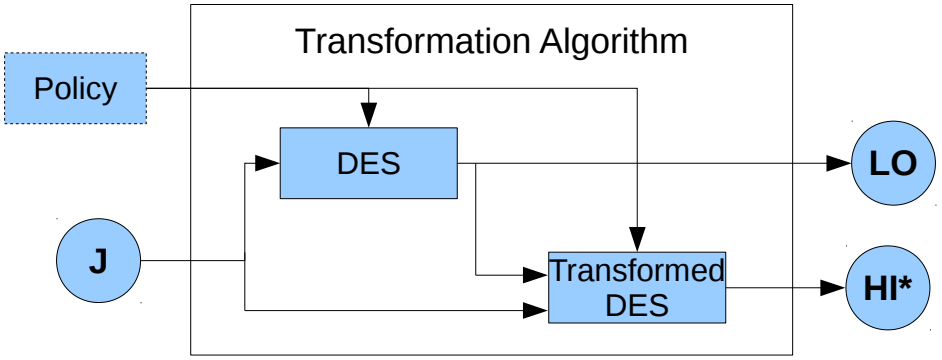


Figure 4.2: An overview of '$\mathcal{T}$' transformation algorithm

where FP represents optimal fixed priority, $STTM$ represents optimal STTM algorithm, OPT represents optimal policy without restrictions and the set inclusion represents dominance relation between different policies. The last equality is a contribution of this work (Corollary 4.3.6).

For multiprocessor platforms our knowledge on schedulability is more modest, as this is new area of research:

Multi Core

$$FP, MCPI \subsetneq FPM; \quad FPM, STTM \subseteq OPT$$

The experiments show, nevertheless, that the set of schedulable instances in our algorithm is almost equal to MCPI, as it manages to convert MCPI results into an STTM table, so we manage to leverage the most part of MCPI performance for the time-triggered scheduling.

4.2 Transformation Algorithm

Our algorithm, denoted $\mathcal{T}(ALG)$ transforms an arbitrary 'basic' scheduling policy ALG to STTM policy by augmenting it with additional rules. In practice, we use an FPM policy, in particular, MCPI, as the basis policy. The algorithm can be applied indifferently on single- and multiprocessor scheduling. Figure 4.2 shows an overview of the algorithm. The policy ALG and the instance **J** are taken as input. Table **LO** is simply obtained by simulating ALG for the LO basic scenario and using the generated schedule S^{LO} as a TT table. We then generate **HI*** by simulating ALG in the HI mode (*i.e.,* initializing χ_{mode} = HI), in what we call "transformed simulation". In this simulation we execute HI jobs with C(HI) times assuming that a HI job can be *disabled* at any time if some *enabling rules*, defined in Section 4.2.3, 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.

4.2.1 Generating the LO table

Scheduling decision can, in principle, be taken at any time, but it was shown in [BLS10] that without loss of generality, we can restrict to take decisions only when one of such events happen:

1. a new job arrives
2. a job terminates its execution

Algorithm: *SimulateEventDrivenPolicy*
Input: **jobset J**
Output: **Schedule $\mathcal{S}$**

1: $St \leftarrow PolicyInit()$
2: $t \leftarrow 0$
3: $E \leftarrow GetArrivalEvents(\mathbf{J})$
4: **while** there are events in E **do**
5: $\quad \mathbf{J}_s \leftarrow PolicyDecision(St)$
6: $\quad (\mathcal{S}, St, t) \leftarrow SimulateUntilNextEvent(\mathbf{J}_s, St, E, t)$
7: $\quad E \leftarrow UpdateTerminationEvent(\mathbf{J}_s, St, E, t)$
8: **end while**

Figure 4.3: Event-driven scheduling policy simulation

3. a job switches

The last event type is needed only in MCS scheduling. Since in this section we are only interested in simulating the LO scenario, we will only consider arrival and termination. We call this kind of scheduling policy *event-driven*. The behaviour of such policy may be reproduced using Discrete-Event Simulation (DES), where the events are the above listed ones. The status of the system St at time t consists of two parts: $St = (Pr, Mem)$. Pr – 'progress of ready jobs' – is defined as, initially empty, set of pairs of the kind (J_j, T_j), where J_j is a job that has already arrived ($A_j \leq t$) but has not terminated yet, and T_j is the cumulative execution time of job J_j at time t. Formally $Pr \subset \mathbf{J} \times \mathbb{Q}$. Mem – 'Memory' – is some status information specific for given policy. To be able to apply our transformation algorithm we restrict the policy to keep Mem orthogonal to Pr, *i.e.,* if one modifies Pr without modifying Mem the policy should still be able to make meaningful decisions.

DES is based on three main steps:

1. Calculate future events
2. Simulate until next event
3. Update system state

The three steps are executed cyclically, until Step 1 returns no event. Pseudocode of Figure 4.3 shows how to implement such a simulator. First of all, the status St, the current time t and the event list E are initialized. The latter is initialized by a function that returns all the "Job arrival" events. Then the algorithms enters into the main loop. Here the *PolicyDecisions* function selects the set $\mathbf{J}_s$ of jobs to be scheduled. For example global EDF on m processor would put in $\mathbf{J}_s$ the m ready jobs with smallest deadline, while an FPM policy would pick the firsts m jobs on the priority table of criticality χ, where χ is the current criticality mode. Next, we perform the simulation until the next event appears, updating the specification of the schedule $\mathcal{S}$, the status St and the current time t. Finally the termination event int list E are updated.

In Appendix A, the pseudocode of an implementation of such simulator is given for the list scheduling algorithm.

Algorithm: *SimulateTransformedPolicy*
Input: **jobset J**
Input: **Gantt chart LO**
Output: **Schedule** $\mathcal{S}$
1: $St \leftarrow PolicyInit()$
2: $t \leftarrow 0$
3: $E \leftarrow GetArrivalEvents(\mathbf{J})$
4: **while** there are events **do**
5: $St' \leftarrow FilterDisabledJobs(\mathbf{J}, St, t, \mathbf{LO})$
6: $\mathbf{J}_s \leftarrow PolicyDecision(St')$
7: $(\mathcal{S}, St, t) \leftarrow SimulateUntilNextEvent(E, \mathbf{J}_s, t)$
8: $E \leftarrow UpdateTerminationEvent(E, \mathbf{J}, St, t)$
9: $E \leftarrow UpdateRulesEvent(E, \mathbf{J}_s, St, t, \mathbf{LO})$
10: **end while**

Figure 4.4: transformed simulation

4.2.2 Generating the HI* table

To generate the **HI*** table, a simple simulation is not enough. In fact, we need to guarantee that 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(HI) time units. To obtain this, we modify the simulation as follows:

- **LO** table is given as input
- we can disable some job, based on some rules on **LO**
- all jobs J_j run for a time C_j(HI)

The reason to disable jobs is explained in the next subsection. To disable a job J_j it is sufficient to hide the pair (J_j, T_j) from $St.Pr$, so that the scheduling policy will not regard it as a ready job. Note that we here exploit the imposed property that the policy permits to modify Pr without modifying policy-specific Mem.

The pseudocode of Figure 4.4 shows how the transformed simulation works. With respect to the algorithm of Figure 4.3, we added two lines. Line 5 hides some of the pairs (J_i, T_i) from St, and saves the modified status in temporary variable St'. The rules to choose the jobs to be disabled are explained in the next section. Line 9 updates the list of events E by computing the events that change the rules of the rules to disable the jobs.

A detailed pseudocode of an implementation of an transformed scheduler for generating table **HI*** for the list scheduling policy is given in Appendix B.

4.2.3 Transformation Rules

In this section we will discuss the rules for disabling jobs in the transformed simulation. Before that, 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(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 it is ready and at least one of the following *rules* is true:

$$T_j^{LO}(t) = C_j(\text{LO}) \tag{4.1a}$$

$$T_j^{HI*}(t) < T_j^{LO}(t) \tag{4.1b}$$

$$T_j^{HI*}(t) = T_j^{LO}(t) \wedge \exists p \;.\; S_p^{LO}(t) = J_j \tag{4.1c}$$

Informally, rule (4.1a) permanently enables all switched jobs, while rule (4.1b) and (4.1c) assure that a job will not run in **HI*** for more time than in **LO** before the switch. Rule (4.1c) enables the execution of a job J_j when it runs on some processor $p \;:\; S_p^{LO}(t) = J_j$. Note that we assume that at start of each new interval of job execution $S_p^{LO}(t) = \epsilon$, *i.e.,* that the job is not yet running but that it starts running immediately after. In other words, $S_p^{LO}(t)$ is not left-continuous when it changes values. Also we assume that it is not right-continuous, *i.e.,* the job stops just before the end of the interval of execution. This is to ensure the intervals where jobs are enabled are *open*. The goal is to have a convenient way to define *busy intervals* in table **HI***, which are, by our convention, also assumed to be open.

Consider the instance and the scheduling defined in Example 4.1.1. We will now show how the enabling Rules can generate the table **HI*** of Figure 4.1. At time 0, only J_1 has arrived, and it is enabled by Rule (4.1c). At time 1, J_4 arrives, it has higher priority then J_1 and it is enabled by Rule (4.1c), so it is chosen by the algorithm to be executed. At time 2 for job J_4 Rule (4.1c) will be false, but Rule (4.1a) 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 (4.1b) until time 5 and by Rule (4.1a) from 5 on. So J_1 will continue its execution till time 6, when J_2 arrives. J_2 is enabled by Rule (4.1c), and it has higher priority than J_1, so it will be executed until time 7. At this instant Rule (4.1c) becomes false for J_2. So J_2 get disabled and we execute J_1. At time 8 J_1 terminates and J_2 is enabled by Rule (4.1c). At time 9 Rule (4.1c) gets false for J_2, while Rule (4.1a) 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 4.2.1. *If at any time we switch from **LO** to **HI***, then all the non-terminated jobs will have enough time reserved in **HI*** to terminate their work.*

Before presenting the proof, first, let us comment that, according to our rules to construct **HI***, no HI jobs get disabled forever because eventually Rule (4.1a) 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 all HI jobs are guaranteed to get $C(\text{HI}) - T_j^{HI*}(t)$, but need 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 4.2.1. 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 (4.1b) or (4.1c) 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, which has important implications, not only for time triggered, but also for other policy.

4.3 Testing Correctness for Single-processor Policies

In this section we always assume single-processor problem instances.

It is known that 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 4.3.1 ('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.*

Note that the reasonable policy definition needs to be generalized in the case of task-graph dependencies by explicitly saying that EDF table should use ALAP deadlines. We will provide detailed generalized definition later on in this chapter.

Theorem 4.3.2 (Transformation Correctness)**.** *For a given task graph on single-processor if the basis policy ALG is correct and reasonable then the policy $\mathcal{T}(ALG)$ is also correct.*

The above theorem is proved in Subsection 4.3.1 for job instances without dependencies and extended to task graphs in Section 4.3.3. We also have a proof for the reverse result:

Theorem 4.3.3 (Reverse Correctness)**.** *For a given task graph on single-processor, 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.*

The proof is given in Subsection 4.3.2 for job instances without dependencies and extended to task graphs in Section 4.3.3. The two theorems above can be expressed, using the notation introduced at the beginning of this chapter, as follows:

$$\mathcal{T}(ALG) = ALG$$

where ALG is reasonable, single-processor basis policy.

Corollary 4.3.4 (Testing Correctness based on two Tables)**.** *For single-processor problem instances and reasonable 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.*

Testing over only two tables implies substantial computational improvement over the correctness test proposed in Theorem 2.1.2, which tests the scheduling policy over $H + 1$ 'tables', where H is the number of HI jobs in the problem instance.

Corollary 4.3.5 (FPM Correctness Testing Complexity)**.** *On single processor testing correctness of FPM policies can be done with the same algorithmic complexity as testing one basic scenario.*

This result follows from the fact that adding the transformation rules to the simulation does not increase the complexity, *i.e.,* generating table **HI*** has the same complexity as simulating one basic scenario. This result follows from Lemma B.2 in Appendix B where we analyze the complexity of transformed fixed-priority simulation for generating table **HI***. Note that this result permits to complete the proof of $O(k^2)$ complexity for MCEDF.

Also, since we do not make any assumption on how the basis policy behaves in the LO mode and because EDF is optimal for single processor single criticality problems the following is trivial:

Corollary 4.3.6 (Of Theorem 4.3.2)**.** *In single processor, if an instance* **J** *is MC schedulable, then it can be scheduled by an STTM policy. Using the notation introduced at the beginning of this chapter:*

$$STTM = OPT$$

Proof. To prove this result, consider an optimal policy and modify it such that it uses EDF in the HI mode, which preserves its optimality. Applying the transformation on the resulting policy gives an STTM policy, which is correct by Theorem 4.3.2. □

Unfortunately these correctness results do not extend to multiple processors. A reason for that is that they rely on optimality of EDF. Nevertheless, from our experiments – see Section 4.4.2 – 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 4.3.7. *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:

$$\begin{aligned} PT_{LO} = &\quad J_1 \succ J_3 \succ J_5 \succ J_7 \succ J_2 \succ J_5 \succ J_6 \\ PT_{HI} = &\quad J_1 \succ J_3 \succ J_5 \succ J_7 \end{aligned}$$

Fig. 4.5 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.

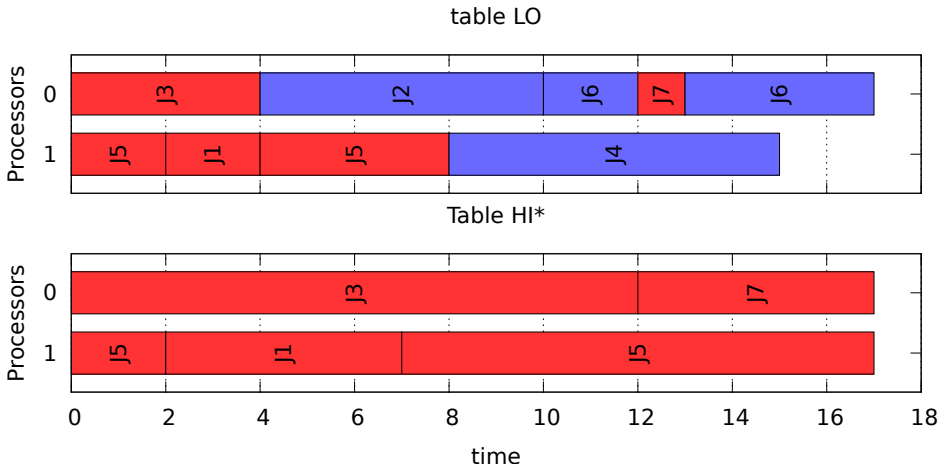


Figure 4.5: TT tables for Example 4.3.7

4.3.1 Proof of Direct Correctness

Theorem 4.3.2 (Transformation Correctness)**.** *For a given task graph on single-processor if the basis policy ALG is correct and reasonable then the policy $\mathcal{T}(ALG)$ is also correct.*

We will prove the theorem for job instances without precedences, the results are extended to task graphs in Section 4.3.3. Let $TT_J^{HI*(LO|HI\text{-}J')}$ be the termination time of J in **HI*** obtained from $\mathcal{T}(Alg)$ (respectively, **LO**, **HI**-J' obtained from Alg).

Theorem 4.3.8. *Let J^{least} be the least priority HI job in the priority table applied in the 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.

Recall the concept of of *busy interval* (Definition 3.2.17). In between busy intervals, 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 (4.1a), (4.1b), (4.1c) 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 4.1 in **HI*** there are two busy intervals: (0,8) and (8,11), thus we have a hole of size 0 at time 8. This hole appears under the following circumstances. Immediately before time 8 J_1 is enabled by Rule (4.1a) while J_2 is disabled. Then at time 8 J_1 gets disabled (because it terminates) while immediately after that time J_2 is enabled by Rule (4.1c) to continue its execution after that time.

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 4.3.9. *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 4.3.10. *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 4.2.1. In addition, by the same lemma we have:

$$T_J^{HI*}(a) \leq T_J^{LO}(a) \tag{4.2}$$

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 (4.1a) 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 (4.1b) is false, which means:

$$T_J^{HI*}(a) \geq T_J^{LO}(a) \tag{4.3}$$

Combining (4.2) and (4.3) we have the equality of the cumulative progress. □

Corollary 4.3.11. *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 (4.1c), which is due to the fact that the other two rules require a switch to have occurred. As long as Rule (4.1c) 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 4.3.8. Let $BI^{HI*} = (a, b)$ be the busy interval in which J^{least} terminates. By Lemma 4.3.9, $TT_{J^{least}}^{HI*} = b$. By Lemma 4.3.10, 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 4.3.11 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 that are ready in interval BI^{HI*} have terminated by time b, we have:

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

Let J_s be the first job to switch in BI^{HI*}, at time t_s. By Lemma 4.3.10 and Corollary 4.3.11, 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 the same set of jobs as in **HI*** may arrive and become ready, and in **HI-**J_s, under EDF policy, the ready jobs will occupy the processor until all of them have terminated – which is the same behavior as for **HI*** in this case. 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 all ready jobs. Also in this case J^{least} by our hypothesis and Lemma 4.3.9 will terminate at time b, but in this case by construction not all jobs of BI^{HI*} terminate by time b:

$$b < a + \sum_{j \in BI^{HI*}} \big(C_j(\mathrm{HI}) - T_j^{HI*}(a)\big) \tag{4.4}$$

Let J_s be the first job to switch in BI^{HI*}, at time t_s. Again by Lemma 4.3.10 and Corollary 4.3.11 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, and at time a their total remaining work is equal to:

$$\sum_{j \in BI^{HI*}} \big(C_j(\mathrm{HI}) - T_j^{HI*}(a)\big)$$

In line with equation (4.4), 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

$$TT_{J^{least}}^{HI\text{-}J_s} \geq a + \sum_{j \in BI^{HI*}} \big(C_j(\mathrm{HI}) - T_j^{HI*}(a)\big) \tag{4.5}$$

Combining (4.4) and (4.5), and observing that $TT_{J^{least}}^{HI*} = b$, 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 4.3.8. □

Proof of Theorem 4.3.2. From Lemma 4.2.1 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 policy *Alg* and Theorem 4.3.8. 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 EDF 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 4.3.8 applied to instance $\overline{\mathbf{J}}$ it will terminate no later than the latest termination under policy *Alg*. Obviously, also the latest termination of the policy *Alg* for job $J^{\overline{least}}$ is the same for both **J** and $\overline{\mathbf{J}}$. Because by our hypothesis this policy is correct 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. □

4.3.2 Proof of Reverse Correctness

In this section we prove the reverse correctness of transformation according to Theorem 4.3.3, *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. As for the previous case, we will prove the theorem for job instances without precedences, the results are extended to task graphs in Section 4.3.3.

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

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

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:

$$WL_{\text{HI}}^{sc}(t) = \sum_{j \in \mathbf{J}: \chi_j = \text{HI}} \left(C_j(\chi_j^{sc}) - T_j^{sc}(t) \right)$$

For table **HI*** we have:

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

Lemma 4.3.12. *Given a reasonable basis policy, we have that:*

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

Proof. Before the mode switch, for any HI job j that did not terminate by 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 4.2.1. On the other hand, for a HI job that has already terminated we have that $C_j(\chi_j^{sc}) - T_j^{sc}(t) = 0$. By the above remarks 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 and LO jobs have been dropped). Next to this, observe that some jobs that are ready in sc may be at the same time disabled in **HI***. Thus, after the switch $WL^{HI*}(t)$ decreases at most as fast as $WL_{\text{HI}}^{sc}(t)$. □

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 4.3.13 (Worst Case Scenario)**.** *Let us consider a reasonable basis policy. Then, for the least priority job J_{least} we have:*

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

*where J_s is the first job to switch in the busy interval of **HI*** where J_{least} terminates and sc' is either the LO basic scenario or any job-specific HI scenario.*

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

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

1. after the switch we have $WL_{\text{HI}}^{sc} = WL^{sc}$.

2. consider two 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 $TT^{sc}_{least} \geq TT^{sc'}_{least}$; (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).

3. In the theorem statement we can ignore the case where sc' is the LO scenario without loss of generality. This is because there always exists a HI scenario where J^{least} terminates at the same time or later, for example HI-J^{least}.

Let t_s be the time when J_s switches in **HI***. We know by Corollary 4.3.11 that $WL^{HI*}(t_s) = WL^{HI\text{-}J_s}_{\text{HI}}(t_s)$. Then, by Lemma 4.3.12:

$$\forall sc' \qquad WL^{HI\text{-}J_s}_{\text{HI}}(t_s) \geq WL^{sc'}(t_s) \tag{4.6}$$

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 (4.6), 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 prove the theorem statement for the other case, $t_{s'} > t_s$. Let $t_{least} = TT^{LO}_{least}$, *i.e.,* the time at which J_{least} terminates in the LO scenario. Note that we can ignore the case $t_{least} < t_{s'}$, as in this case $TT^{sc'}_{least} = TT^{LO}_{least}$ and Observation 3 applies. So, we can assume $t_{s'} \leq t_{least}$. Due to this assumption, we also have: $t_{s'} \leq TT^{HI*}_{least}$ and $t_{s'} \leq TT^{HI\text{-}J_s}_{least}$. Adding to this that $t_s < t_{s'}$ we see that $t_{s'}$ falls inside the busy interval where J_{least} terminates in the end, both for **HI*** and $HI\text{-}J_s$. By construction, t_s belongs to the same busy interval BI^{HI*} that ends at TT^{HI*}_{least}, 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 $WL^{HI*}(t_s) = WL^{HI\text{-}J_s}_{\text{HI}}(t_s)$, 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 4.3.3. *For a given task graph on single-processor, 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:

$$(\forall j \quad TT^{HI*}_j \leq D_j) \quad \Rightarrow \quad (\forall sc, \forall i \quad TT^{sc}_i \leq D_i)$$

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 4.3.2 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:

$$TT_{least}^{HI*} \leq D_{least} < TT_{least}^{HI\text{-}J_s} \tag{4.7}$$

where $HI\text{-}J_s$ is the worst case scenario for J_{least} according to Theorem 4.3.13. We distinguish two cases:

1. J_{least} **terminates before an "empty interval".**

   By the reasoning of the proof of Theorem 4.3.8, we have:

   $$TT_{least}^{HI*} = TT_{least}^{HI\text{-}J_s}$$

   which contradicts (4.7).

2. J_{least} **terminates before a "hole".** Considering $BI^{HI*} = (a, b)$, as in the proof of Theorem 4.3.8, and observing that, by Lemma 4.3.10, $T_j^{HI\text{-}J_s}(a) = T_j^{HI*}(a)$ we have that:

   $$TT_{least}^{HI\text{-}J_s} = a + \sum_{j \in BI^{HI\text{-}J_s}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right) \tag{4.8}$$

   Let J_e be the last job to terminate in **HI***. For this job, by construction:

   $$TT_e^{HI*} \geq a + \sum_{j \in \mathbf{J}} \left(C_j(\text{HI}) - T_j^{HI*}(a)\right) \tag{4.9}$$

   The right side of Equation (4.9) is no less than the right side of Equation (4.8). Therefore, $TT_e^{HI*} \geq TT^{HI\text{-}J_s}$. Also, in EDF: $D_{least} \geq D_e$. From these observations and (4.7), 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.

□

4.3.3 Extending the Proofs to Task Graphs

In this section we will show that the theoretical results so far obtained for single-processor case without dependencies extends to the case of task graph as well.

Definition 4.3.14 (Modeling job set)**.** *Given a task graph $\mathbf{T}(\mathbf{J}, \rightarrow)$, its modeling job set $\hat{\mathbf{J}}$ is the set of jobs whose arrival times and deadline are calculated as ASAP arrivals A_j^* and ALAP deadlines D_j^*.*

The above definition can be applied to LO-criticality, MIX-criticality and HI-criticality graph (Section 2.2.3). In this case we will talk of LO (resp. MIX,HI) modeling job set $\hat{\mathbf{J}}_{\text{LO}}$ (resp. $\hat{\mathbf{J}}_{\text{MIX}}$, $\hat{\mathbf{J}}_{\text{HI}}$).

In this subsection we will assume reasonable policies, according to the following definition:

Definition 4.3.15 (Reasonable policy for task graphs)**.** *A reasonable policy for task graph uses in HI table EDF priorities according to ALAP deadlines $D*$.*

Observation 4.3.16. *In a modeling job set, the following holds:*

$$\forall i,j \quad J_i \to J_j \Rightarrow A_i^* \leq A_j^* \tag{4.10}$$

Observation 4.3.17. *Meeting the ALAP deadlines in all three modeling instances (assuming LO mode for $\mathbf{J}_{MIX}$) is necessary and sufficient for for policy correctness, and PT_{EDF} for ALAP deadlines is precedence compliant.*

The following holds for single processor:

Lemma 4.3.18. *Consider a task graph $\mathbf{T} = (\mathbf{T}, \to)$. The LO basic scenario schedule $\mathcal{S}_{LS}$ obtained by applying list scheduling on $\mathbf{T}$ using a priority compliant table PT_{LO} is equal to the schedule $\mathcal{S}_{FP}$ obtained by applying fixed priority scheduling on $\hat{\mathbf{J}}_{LO}$ ($\hat{\mathbf{J}}_{MIX}$) without dependencies using the same table PT_{LO}.*

Proof. The difference between the simulation for constructing $\mathcal{S}_{LS}$ and $\mathcal{S}_{FP}$ is that in the former case some jobs may be postponed due to dependency constraints on their predecessors. Thus the results are not equal if and only if at a certain time instant a job J has arrived, but is postponed in $\mathcal{S}_{LS}$ while it is the highest priority non terminated job in $\mathcal{S}_{FP}$. Let us assume by contradiction that such a time instant exists. By Equation (4.10) if J has arrived, so are all of its predecessors and some of them are non-terminated, otherwise J would not be postponed. Since, by hypothesis, PT_{LO} is priority compliant, all the predecessors have higher priority. This contradicts the hypothesis that J is the highest priority non terminated ready job. □

Dually, we also prove the following:

Lemma 4.3.19. *Consider a task graph $\mathbf{T} = (\mathbf{T}, \to)$. The schedule $\mathcal{S}_{LS}$ obtained by applying transformed list scheduling on $\mathbf{T}_{HI}$ using a priority compliant table PT_{HI} is equal to the schedule $\mathcal{S}_{FP}$ obtained by applying transformed fixed priority scheduling on $\hat{\mathbf{J}}_{HI}$ without dependencies using the same table PT_{HI}.*

Proof. Unlike Lemma 4.3.18, in $\mathcal{S}_{FP}$ we do not always execute the highest-priority non-terminated ready job J', since J' may be disabled by the transformation rules. We have to prove that this does not break a precedence relation $J' \to_{\text{HI}} J$, *i.e.,* in S_{FP}, J must not run when J' is disabled. Formally:

$$J' \to_{\text{HI}} J \Rightarrow \forall t \leq TT_{J'}^{\mathcal{S}_{FP}}, \;\; T_J^{\mathcal{S}_{FP}}(t) = 0 \tag{4.11}$$

Notice that $J' \to_{\text{HI}} J \Rightarrow J' \to J$, which should be respected in the LO table, so:

$$J' \to_{\text{HI}} J \Rightarrow \forall t \leq TT_{J'}^{\mathbf{LO}}, \;\; T_J^{\mathbf{LO}} = 0 \tag{4.12}$$

Consider a time instant $t' \leq TT_J^{\mathbf{LO}}$. By (4.12) we have that J is not started yet in **LO**, thus none of the enabling rules is true for it and hence J is disabled. On the other hand, at time instant $t'' > TT_J^{\mathbf{LO}}$, J' is always enabled until its termination by Rule 4.1a. Thus we cannot have a situation where non-terminated J' is disabled and J is enabled at the same time. The lemma may now be proved with a similar reasoning as in the proof of Lemma 4.3.18. □

Lemma 4.3.18, Lemma 4.3.19 and previous observations show that task graphs can be modeled by the precedence-free modeling job instance, thus we trivially have the following:

Observation 4.3.20. *Theorems 4.3.2 and 4.3.3 extend to task graphs.*

4.4 Experiments with Multiprocessors

4.4.1 Extending the Scope for Transforming the Policies

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) EDF scheduling policy is applied in the HI mode. Nevertheless, our algorithm is applicable in practice also when the above restrictions are alleviated, though one has to check the correctness of the result even when the basis policy is correct.

In this section we experiment with an approach that goes beyond the above restrictions and works in practice, even if not supported by theory. Thus, we abandon the usual mindset of previous works on mixed-critical translation of event-triggered to time-triggered table [BF11, Bar12, Bar13], which study only the cases where correctness of transformation can be proved. Instead, we test the correctness after the translation.

Thus we now consider multiple processors and non-EDF policy in HI mode as basis policy. Instead we use MCPI with EDF-DS support algorithm which we showed efficient for multiple processors in Chapter 3.

Our experiments in next subsection show that even for hard scheduling problem instances the proposed approach results in a correct transformation in a grand majority of problem instances.

The **HI*** table correctness consists of two main requirements:
(a) no job should ever make more progress in **HI*** than in the **LO** table;
(b) deadlines should be met by the HI jobs.

Requirement (a) is satisfied by Lemma 4.2.1, which also applies for multiprocessor case. Requirement (b) can be easily verified after the construction of the **HI*** table, which we do in our experiments.

Recall that MCPI implementation is based on list scheduling. Therefore, for transforming the MCPI to STTM we had to implement the transformed list scheduling. The implementation is described in detail and analyzed for algorithmic complexity in Appendix B. In the next subsection we apply it in experiments.

4.4.2 Experimental Results

To estimate the probability of getting a correct solution from a transformed policy in the the multiprocessor case, we performed measurements for randomly generated problem instances using an implementation of $MCPI$ and $\mathcal{T}(MCPI)$.

The random job generation algorithm was the same as in Section 3.6. The instances were scaled to obtain a target '$Stress$' parameter. In our experiments the stress in the two modes was set to be equal to a target value $Stress_{\text{LO}} = Stress_{\text{HI}} = S$. For each instance, we first applied the MCPI algorithm. If it did not produce a correct solution we canceled and restarted the experiment. In the case of a correct solution from MCPI we applied the transformation algorithm to the result and checked if the experiment was a 'success', *i.e.,* whether a feasible STTM **HI*** table was obtained. We performed the experiments to test the effectiveness of the algorithm under different assumptions. In Table 4.4.2 the parameters used for each experiment are reported. It is shows for each number of processors m the maximum allowed error δ on the Stress value and for each experiment the number of jobs and the number of precedence arcs. For each point, 1000 MCPI-schedulable instance were generated, in order to get a representative sample.

		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 4.1: Experiments' parameters

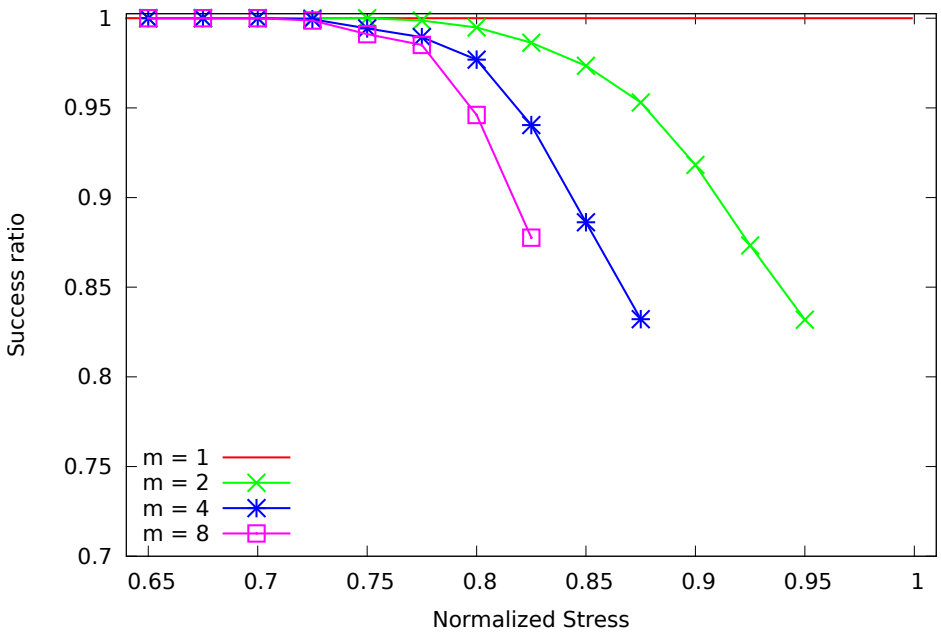


Figure 4.6: Experiment 1 - Without dependencies

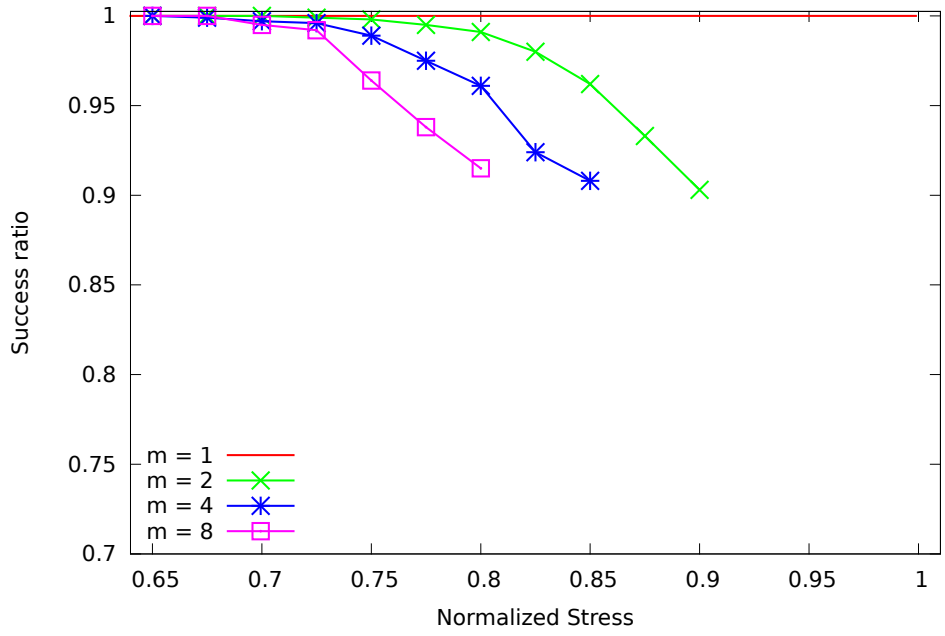


Figure 4.7: Experiment 2 - With dependencies

Experiment 1 shows the performance of the algorithm with no dependencies. Fig. 4.6 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. 4.7 shows the results of experiment 2, where we added task graph dependencies. From the graph we can see that there is no substantial 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.

4.5 Chapter Summary

In this chapter we discussed the problem of transformation from a dynamic online policy to static policy. Static policies are useful in time-critical systems, since, in general, they make analyses easier, by reducing the number of states that the system may reach. This, together with the observation that the mixed critical literature contains few works on static policies, but many on priority based solutions, motivated the studies presented in this chapter.

For single processor case, we proved that the proposed policy transformation algorithm is optimal, in the sense that it will always find a correct static schedule in case the original policy is correct as well. We also proved that this relation holds in the reverse direction as well, *i.e.,* if the generated static policy is correct, so is the original one. This has both theoretical and practical implications. From the theoretical point of view, we showed that the *Single Time Table per Mode* (STTM) policy is optimal for single processor case, *i.e.,* if

there exists a correct scheduling policy then there exists a correct STTM policy. From the practical point of view, it means that our translation algorithm may also be used to test the correctness of a non static policy. This procedure has a lower computational complexity then simulating all job-specific basic scenarios. Preliminary single processor results where presented in [4] (see Contributions Bibliography, page 123). The reverse correctness and its theoretical and practical applications are new, unpublished results.

For the multiprocessor case, we lose the optimality. However we were able to show that the algorithm has a very low failure rate even for instances with high utilization. This results are also present in [6].

As future work we plan to extend the proposed algorithm to multiple level of criticality.

Chapter 5

Application Programming and Implementation

An important challenge in the design of mixed-critical system is lack of consolidation in programming. Embedded software design has in common with hardware design that it has to satisfy not only functional, but also extra-functional requirements, first of all, timing. However, unlike hardware languages, the software languages have an important deficiency: they were conceived without any concern on timing in mind [Lee05]. Real-time programming is a very heterogeneous area of research, as it employs many different models of computation (MoCs), such as synchronous languages, timed Petri nets, various extensions of synchronous dataflow (SDF), etc. Expressing the software design in a given MoC is difficult, but, worse still, even when this is done, the real-time scheduling and timing analysis still remains challenging, due to a gap between the MoCs and the real-time scheduling policies [FKRvH06]. The policies themselves are sometime very heterogeneous, and 'exotic', especially those proposed for multiple processors and for mixed criticality *e.g.,* in Chapter 3 and 4 of this thesis.

Therefore the user should be given freedom to configure his preferred model of computation and scheduling policy on top of a metamodel that can express all the spectrum of choices and their combinations. So there is a need in a common 'backbone' language expressive enough to redefine and reuse different components of middleware. Partly, this idea was implemented in the SystemC project, offering a common way to express scheduling policies, MoCs and functional code [LMPC04]. However, this language lacks a formal semantics, and it mainly offers facilities for simulation only, but not for fully-automated deployment of software.

Therefore, as an alternative, we propose to use *combined procedural and automata languages*. We also propose to 'compile' high-level descriptions of custom models of computation and scheduling policies into the automata language to have a unified 'backbone' model from which one can do code generation and timing analysis in a unified way. To demonstrate the concept, we offer public prototype tools [PBS$^+$] for multicore timing-critical system design based on the timed-automata language BIP [ACS10] extended for the support of tasks.

This chapter describes the design flow for programming and implementing mixed critical systems in this language. Section 5.1 gives an overview of the proposed design flow, Section 5.2 gives an introduction into BIP. In Section 5.3 we present our proposed MoC for real-time systems. In Section 5.4 we show how to translate an instance of MoC and scheduling policy into BIP. The BIP programming language can run on top of our BIP run-time environment for multicore systems used for simulation and deployment. In Section 5.5 we

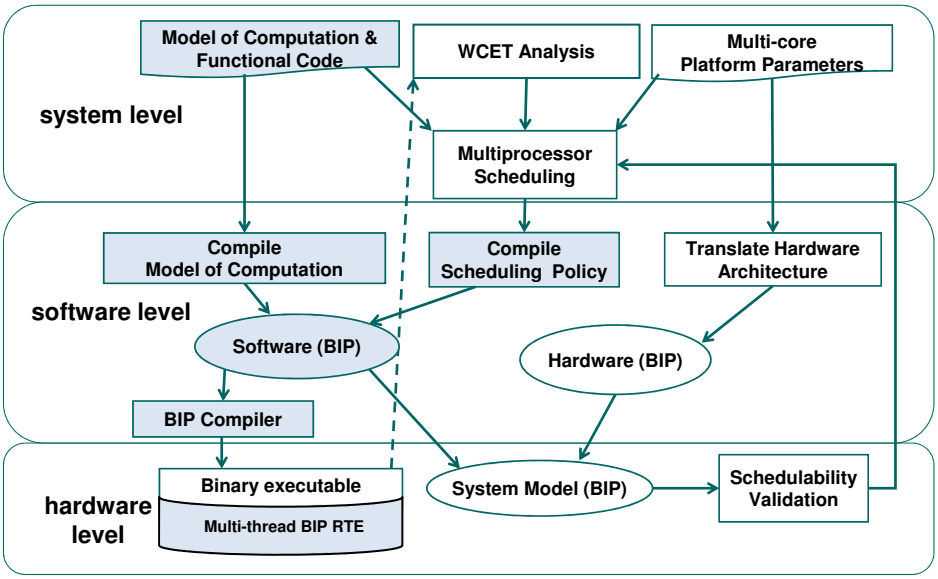


Figure 5.1: Design flow (highlighting the steps covered in this chapter)

describe implementation of an industrial use case on Kalray MPPA® multi-core platform and some experimental results. Finally, Section 5.6 concludes the chapter.

5.1 Design Flow

In this section we present our proposed design flow. It is based on a combined procedural and automata-based language, referred to as *backbone language*. Examples of such languages are IF, SDL-RT, BIP and TIMES [AFM+02]. A backbone language can be used for modeling, validation and simulation, but our main point is to use it as a programming language. Because in many cases, an automata-based language may be too low-level for direct use in application programming, we can *compile* higher-level models into the backbone language automatically. In the ideal case, this is ensured by letting the user create a set of grammar rules for automatic translation of the high-level model patterns into the automata. The user would provide a set of automata templates that implement the primitives of the preferred MoC and scheduling policy. Hence the backbone language would serve as a meta-model and meta-policy used to program the desired timing-critical systems middleware. The specified set of rules and templates would allow to compile the MoC and the scheduling policies into a network of timed automata that can be analyzed and deployed on a platform.

This idea is partly implemented in our design flow, see Fig. 5.1. The design flow accepts a high-level specification of application tasks (the MoC and the functional code) at the input and compiles it into the backbone language, for which we use BIP [ACS10]. Also the flow takes from the offline scheduling tool the specification of the online scheduling policy and the selected scheduling parameters (such as priorities) of the tasks. The scheduler is also compiled into backbone language and 'plugged' into the common BIP software model. This model is deployed on the platform on top of the BIP run-time environment (RTE) for multi-cores. The software model can also be combined with the hardware model to represent the complete software-hardware system and to perform timing analysis for validation of

schedulability properties, but the validation part of the design flow is beyond the scope of the prototype toolset [PBS+] and this chapter.

Currently we support only one MoC – Fixed Priority Process Networks (FPPN – see Section 5.3), which combines the abilities to model both the reactive-control and signal-processing applications. As for the scheduling policies, we support a time-triggered with synchronization (TTS) policy [GSHT13]. In future we consider to provide means to the user to specify templates for his preferred MoC and policy. We also consider to add support for other relevant MoCs, such as synchronous languages, as in the Prelude [CBF+11] framework, and SDF, such as in CompSoC [HGBH09].

5.2 Real-Time BIP

A *backbone language* defines the concurrency and timing semantics of all system software components. After compilation from MoC and policy into a backbone language, one obtains an executable model that can be simulated for functional validation. This model is also used as reference for system analysis and code generation. In our design flow the backbone language is BIP.

Under 'BIP' we refer to the so-called 'RT-BIP' dialect [ACS10], which is designed to express networks of connected timed automata components (Section 5.2.1). In [GPS+], we extend BIP from timed to task automata, by allowing *self-timed* automata transitions. This extension allows expressing control decisions based on runtime monitoring of task response times in timed automata, similarly to task automata [FKP07] in TIMES tool [AFM+02]

This feature is important for runtime resource management mechanisms, such as those employed for mixed criticality. For example, recall that our mixed criticality scheduling policy makes online decisions based on the monitoring executions times of jobs. A particular feature of BIP is the ability to specify a *network* of components, so that multiple tasks can be executed in different components concurrently. This makes it particularly suitable for multi-core platforms. Our extensions to the original RT-BIP dialect are presented in Section 5.2.2. They are necessary to realize the models of MC systems presented in this chapter.

5.2.1 Introduction to BIP

To familiarize the readers with BIP notation, Figure 5.2 shows a BIP example, representing two tasks, A and B. These can be scheduled on one of the two available threads running on two different cores. The model consists of four components, namely, 'PeriodicA', 'DelayableB', 'Thread1' and 'Thread2'. All the components are defined by an automaton and a set of *ports* (shown in white rectangles), used for connecting to other components via *connectors* (shown as green lines that join the bullets).

A BIP component has multiple *locations*, denoted in Figure 5.2 as 'S0', 'S1'. The *execution run* of a component consists of going from location to location by taking a *transition*, denoted by an arc. For example '(Skip)' is a transition from location 'S1' to location 'S0' in component 'DelayableB'. Each component has an *initial transition*, which brings it to initial location at system start. Initial transition is shown as an arc without origin pointing to the initial location, such as location 'S0' in 'DelayableB'. A transition may have an *enabling condition* and may trigger some *action*. In our figures, we show the conditions in blue color and square brackets, *e.g.,* component 'DelayableB' has condition '$[D_{\text{OUT}} \neq 0]$' for transition 'StartB'. The actions are shown in red color.

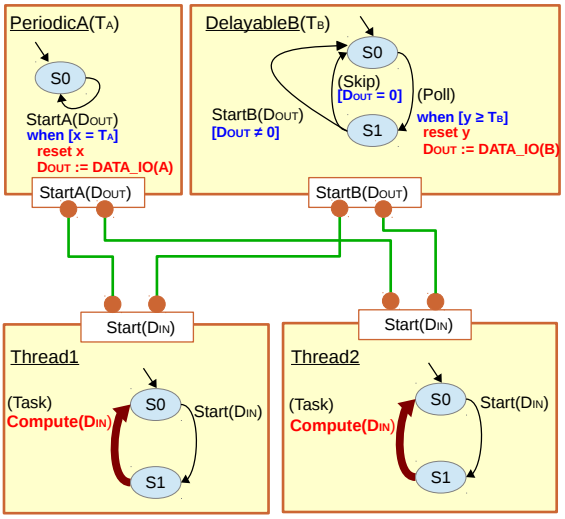


Figure 5.2: BIP model example

The transition labels such as 'StartB' signify a port of the component, in which case the transition *participates in interactions* through this port, which means that it is synchronized with transitions in other components whose ports are connected, *e.g.,* 'StartB' may interact with 'Start' in 'Thread1' or 'Thread2'. Note that a port may participate in one interaction at a time. In our example, each port is linked to two connectors, so if both of them have an enabled interaction, a non-deterministic choice has to be made between them. There are also *internal transitions*, not associated to ports, executed by a component independently. We put their labels in parentheses, *e.g.,* '(Skip)' and '(Poll)'.

In BIP, every component is seen as an object in an object-oriented programming sense. Every component encapsulates some data and some subroutines to manipulate the data. The actions of transitions can call subroutines written in an imperative language (C/C++). In the figures, the actions are depicted as blocks of pseudo-code in red color, *e.g.,* in component 'DelayableB', transition '(Poll)' executes action '$D_{\mathrm{OUT}} :=$ DATA_IO(B)', where a subroutine is called and its return value is assigned to variable 'D_{OUT}'. The actions have access only to the local variables of the component itself, but the components may exchange data from 'OUT' to 'IN' variables at interactions via ports. For example, port 'Start(D_{IN})' receives the new value of D_{IN} from the D_{OUT} of either 'StartA' or 'StartB', depending on the component with which it interacts. Note that the data exchange between ports precedes the transitions, *e.g.,* port 'StartA(D_{OUT})' sends the value of D_{OUT} *before* it is modified by the respective transition.

As for the data variables, in this work we consider four main types: integer, Boolean, reference, and queue. A *reference* is a pointer to a user-type object that is allocated at component initialization. Our models for critical systems do not dynamically allocate data after system initialization. A *queue* is a circular buffer of statically-known size. Unless explicitly done otherwise in the initial transition or in natural-language annotations, in the presented figures we assume that the initial transition implicitly sets the data variables to zero in the case of integers, 'False' for Booleans *etc.* Besides data variables, the components can have compile-time parameters, such as period T_A and minimal execution interval T_B in Figure 5.2.

The condition to execute a transition in fact consists of two parts: a data condition and a timing constraint, indicated by the keyword 'when'. The *timing constraint* defines an interval of time when a transition may be enabled. By default it is 'always', *i.e.,* the whole time axis.

To define the timing constraints a component uses private *clock variables*. The clocks are real-valued variables that are initialized to zero and whose values are continuously and synchronously increasing with the passage of physical time. In our models, we use letters x, y and t for the clocks, e.g., the model in Figure 5.2 uses two clocks. The usage of clocks is restricted to two possible scenarios. Firstly, a clock can be reset to zero inside a transition action (*e.g.,* 'reset x' in 'PeriodicA'). Secondly, it can be used in the timing constraint of a transition, see, (*e.g.,* 'when $x = T_{\text{A}}$' in 'PeriodicA').

In our models we assume that all transitions are marked as '*urgent*' in BIP. The presence of 'urgency' attribute means that the transition should start *as soon as (and no later than)* this transition and all those that participate in the same interaction (if any) get enabled. For example, consider timing constraint 'when $[y \geq T_B]$' in Figure 5.2. Due to this constraint, if component 'DelayableB' is in location 'S0', then it should execute transition '(Poll)' immediately when it sees that clock y has reached a value at least equal to T_B. Note that the 'urgency' property is usually not directly available in timed automata languages, but it is very useful for modeling compute-intensive real-time systems, where typically the system must make progress *immediately* when several conditions become true. For example, in list scheduling policy a job should become ready immediately when all its predecessors have terminated and its arrival time has elapsed.

In our BIP programs for time-critical systems we often use *queues*. This well-known data structure can be easily implemented using a circular buffer. We define the following operations on the queue:

- **NewTail**() Gives a reference to the cell where the next data will be written (new 'tail' of the queue)
- **Push**() push the last allocated cell into the *tail*
- **Pop**() extract the *head* of the queue

Notice that the cell are allocated statically. The NewTail() function does not dynamically allocate new memory.

5.2.2 BIP Extension for Modeling the Tasks

By default, BIP assumed that all data-processing actions cost zero time (at least, conceptually). However, real-time tasks may occupy the processing cores at significant utilization levels, and to properly model them one should allow executing their data-processing operations in non-zero time. Therefore, in the extended version of BIP, we distinguish between the 'starting' and the 'finishing' times of a transition, and we refer to the time duration in between as *transition response time*. Further, new '*self-timed*' attribute is introduced for the transitions and we assume that all transitions are conceptually instantaneous (*i.e.,* have zero response time) unless they have this attribute. A transition marked as self-timed has a response time equal to the time required to finish the corresponding action on a finite-speed physical resource. This can take any time duration, not known at the moment when the transition starts.

We use *internal self-timed transitions* to represent task processing steps and *self-timed interactions via ports* to represent inter-task communication. In our figures, we denote self-

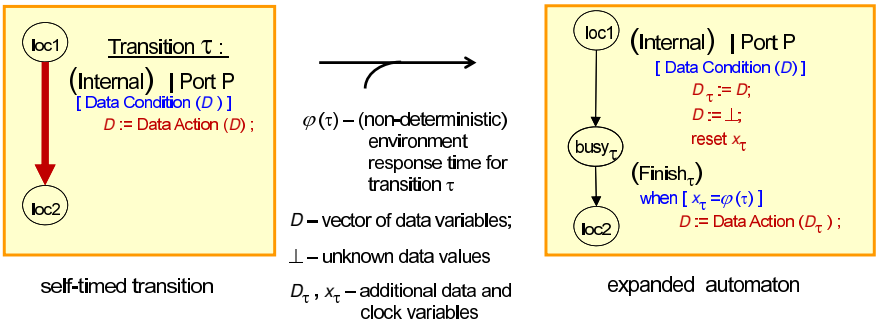


Figure 5.3: Modeling tasks in BIP

timed transitions by thick arrows, *e.g.,* '(Task)' transitions in Figure 5.2. Note that by putting a self-timed transition in between two instantaneous transitions, one can measure its response time by resetting a clock before and checking the clock value after the self-timed transition. Measuring the response time is necessary to program mixed-criticality scheduling policies.

Though the self-timed transitions represent a new concept added into BIP language to model tasks, at the *semantics level* the behavior can be interpreted into the default BIP language, *i.e.,* timed automata with instantaneous transitions. Nevertheless, at the implementation level, the BIP framework needed certain extensions to handle these transitions correctly. Figure 5.3 shows a self-timed transition τ of a task automaton in the extended BIP and its interpretation in timed automata of the 'default' BIP. In the timed automaton model, transition τ is represented by two instantaneous transitions, one modeling the start and other one the finish. In between these transitions, there is a location 'busy$_\tau$', which models the state where the system is busy waiting until the platform executes transition τ. Note that the data variables are explicitly set into 'unknown' state, because during the execution they can potentially take arbitrary values. Note also that if the transition interacts with other components via a port, then in the expanded automaton the port is inherited by the start transition, which indicates that the interacting components synchronize with each other at the start of their transitions.

An additional clock x_τ measures the elapsed time since the start and the execution of transition τ. The execution finishes when the response time of transition τ, denoted $\varphi(\tau)$, has been reached. Model-wise, it is important to observe that the 'Finish$_\tau$' transition and time $\varphi(\tau)$ are controlled not by the system itself, but rather by an external party, *i.e.,* the *environment*. Indeed, the software cannot directly influence the time it takes to execute a given, arbitrarily complex piece of the task's code. This is determined by the target platform, which actually acts here as environment. For simulation or modeling purposes, one can make an abstraction of the the environment by letting $\varphi(\tau)$ take non-deterministic values. However, when *implementing* the BIP program on a real platform, the BIP system may not 'decide' by itself, non-deterministically, how long delay $\varphi(\tau)$ should be. Instead it should let the environment 'decide' this. Therefore, it should start the execution of the transition on the platform and wait until the platform eventually signals its completion. This observation makes the difference between executing the BIP model on the left and on the right of Figure 5.3.

5.3 Fixed Priority Process Networks

5.3.1 Model of Computation

In our framework, we currently work with a functionally-deterministic MoC intended to support both the reactive control and the signal-processing applications, the so-called *Fixed Priority Process Networks* (FPPN) [PSPBB15], which is closely related to synchronous languages. An instance of FPPN is composed of three main entities: *Processes*, *Data Channels* and *Event Generators*. The determinism is ensured by *Functional Priority* relation between the processes.

A *Process*, or task (in the terminology of real-time systems), represents a software subroutine that operates with internal variables and input/output channels connected to it through ports. One call to this subroutine models a job execution. Contrarily to [PSPBB15], here we assume a version of FPPN extended for mixed-criticality, where each process has a criticality-level attribute 'χ_p' and its job subroutine has 'mode' argument, which indicates whether the job should execute in normal or degraded mode. The latter replaces the usual Vestal's model dropping of LO jobs, in FPPN the LO jobs are not entirely dropped, but may execute in degraded mode instead. We borrow this feature from the DOL critical MoC, discussed later in this section.

The *functional code* of the application is defined in processes, whereas the necessary *middleware* elements of FPPN are channels, event generators, and priorities.

Data Channels ensure *non-blocking read and write operations* for communication. There are inter-process and external (environment) channels. In this paper we consider only the inter-process channels. We define two channel types: FIFO and blackboard. Other types can be introduced by extension of the library of BIP components. The FIFO has a semantics of a queue. The blackboard remembers the last written value that can be read multiple times. Reading from an empty FIFO or a non-initialized blackboard resets an indicator of data validity.

An *event generator* e is defined by the set of possible sequences of time stamps τ_k that it can produce. We define two types of event generators: *periodic* and *sporadic*. *Every event generator is associated with a unique process* and determines whether the given process is periodic or sporadic one. Every process p has a deadline d_p. Interval $[\tau_k, \tau_k + d_p)$ determines the time interval when the k-th process job can be executed. At τ_k the job gets 'activated' and then it remains active until it is scheduled. After being scheduled, the job should terminate before the deadline. Periodic processes are activated at period T_p, for sporadic processes T_p denotes the minimal inter-arrival time. We define the *job queue length* as $q_p = \lceil d_p/T_p \rceil$, this quantity is the maximum number of jobs of process p that can be active simultaneously.

An FPPN network can be described by two directed graphs. The first graph is the default process network graph (P, C), whose nodes are processes P and the edges are channels C. This graph can be cyclic and defines the communicating pairs of processes and the direction of dataflow: from writer to reader. The second graph is the functional priority DAG: (P, FP). No cyclic paths are allowed in this graph. The edges define functional priority relation between the processes. It is not a partial-order relation, as it is not necessarily transitive. We require that any two communicating processes have a priority relation: if $(p_1, p_2) \in C$ then $(p_1, p_2) \in FP$ or $(p_2, p_1) \in FP$, *i.e.,* a functional priority should either follow the direction of the data flow or the opposite direction. In sequel we use notation $p_1 \rightarrow p_2$ to denote $(p_1, p_2) \in FP$ and say informally that p_1 has 'higher' functional priority than p_2.

Fig. 5.4 below gives an example of a process network. This process network represents

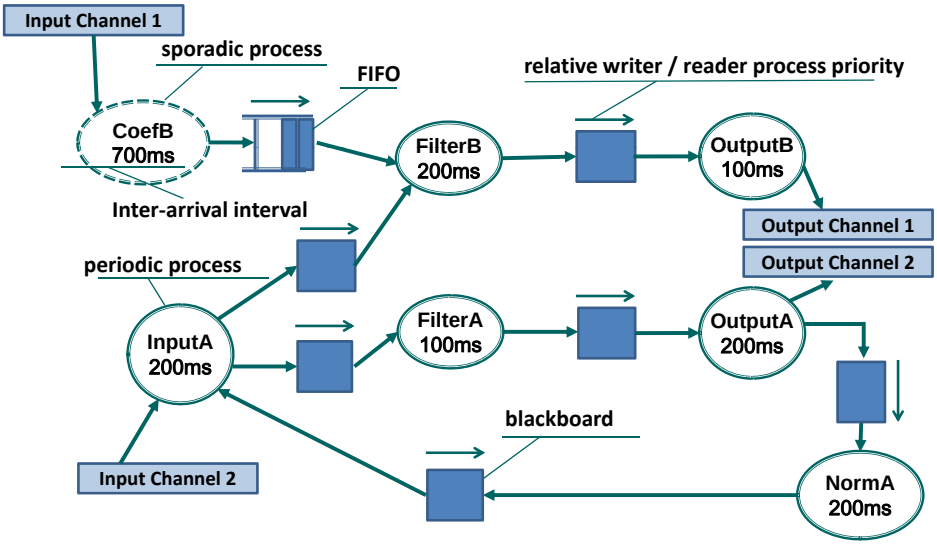


Figure 5.4: Example of Process Network

an imaginary signal processing application with input sample period $200ms$, reconfigurable filter coefficients and a feedback loop. The filter coefficients are reconfigured by a sporadic event (a command from the environment) that activates the sporadic process CoefB.

We see several periodic processes, annotated by their periods, and a sporadic process, annotated by minimal inter-arrival time. We also see inter-process channels – the blackboards and a FIFO, annotated by an arc of the functional priority relation FP. Also the environment input/output channels are shown.

The semantics FPPNs is described in [PSPBB15]. The main idea is that every pair of processes that share a channel are executed in well-defined relative order determined by (i) their activation times and (ii) functional priorities. This order of process execution instances (jobs) is compatible to the total order derived from zero-execution-time simulation of fixed-priority scheduling, hence the name of the MoC. Because the ordering is imposed only between communicating jobs, it is a partial order, allowing for parallelism. For a certain class of FPPN this order can be expressed in static task graph.

5.3.2 Task Graph Derivation

FPPN is a model of computation designed to formalize the behavior of real-time tasks with deterministic communication, including those uniprocessor systems that exploit the FP schedule priority to ensure determinism. For them there exists a family of relevant scheduling techniques, such as [F$^+$10, Bar12]. The latter supports mixed criticality. Such techniques can be seen as ready-to-use uniprocessor scheduling methods applicable to FPPN and related models, such as synchronous languages [Bar12].

As a formal language, FPPN should show the same deterministic behavior no matter which platform it is implemented on. A functionally correct implementation of a formal language would ensure deterministic execution on multiple processors, but ensuring also timeliness would remain to be challenging and a subject of schedulability analysis. This problem gets even harder when sporadic tasks are involved. Therefore, to demonstrate scheduling for FPPNs, we consider a practically relevant subclass of FPPNs where the use

of sporadic tasks is restricted.

From the subclass of FPPNs considered here one can statically derive a *task graph* which then serves as input to an offline scheduling algorithm. The algorithm generates a static schedule, where we model sporadic processes by periodic ones with strictly higher demand of resources. To make it possible, we put a restriction that each sporadic process p be connected by a channel to exactly one 'user' process $u(p)$, which must be periodic and which must have at most the same period[1]: $T_{u(p)} \leq T_p$. This restriction is practically relevant, because a sporadic process often plays an utility role, 'configuring' some application parameters of a periodic process.

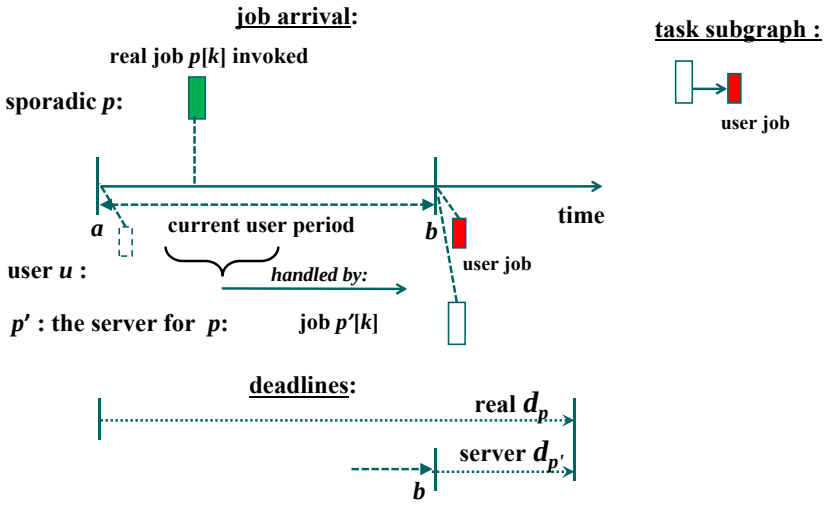


Figure 5.5: Handling a Sporadic Process.

The run-time sporadic jobs invoked inside the user period are modeled by 'periodic server' jobs that arrive at the boundaries of the user period intervals. As indicated in the task subgraph, the server jobs at time b must have precedence over the user job that also arrives at time b. This is so because for causality reasons the server jobs can only handle the real jobs that have been invoked in the past, *i.e.,* inside (a, b), whereas FPPN semantics requires that the earlier invoked jobs have precedence over the later ones. For convenience, we say that the server jobs for process p are generated by an imaginary periodic 'server process' p'. To ensure the precedence of the server jobs we set: $p' \rightarrow u(p)$. Note that this does not mean that sporadic processes must always have priority over their users (*i.e.,* $p \rightarrow u(p)$ is not required), the higher priority is only required for their 'servers', which are imaginary processes introduced to define the static task graph for offline scheduling.

The deadlines of the server jobs are corrected to compensate for worst-case one-period postponement of job arrival due to waiting until the job is handled by the server[2]: $d_{p'} = d_p - T_{u(p)}$. Thus, we effectively assume arrival at time b but count the deadline from time a, in order to be conservative.

Definition 5.3.1 (Task Graph)**.** *A task graph is a directed acyclic graph (DAG)* $\mathcal{TG}(\mathcal{J}, \mathcal{E})$ *whose nodes are jobs:* $\mathcal{J} = \{J_i\}$*. A job is characterized by a 6-tuple* $J_i =$

[1] one could relax the restrictions on the number of user processes and their periods at the cost of somewhat more complex task graph construction

[2] here we implicitly require that $d_p > T_{u(p)}$ but if it is not the case we can use server jobs with a period T' being a fraction of $T_{u(p)}$ instead, so that the server deadlines become positive

$(p_i, k_i, A_i, D_i, \chi_i, C_i)$, where: p_i is the process to which the job belongs, k_i is the invocation count of job, $A_i \in \mathbb{Q}_{\geq 0}$ is the arrival time, $D_i \in \mathbb{Q}_+$ is the required time (absolute deadline), χ_i is the criticality level, $C_i : \{LO, HI\} \mapsto \mathbb{Q}_+$ are LO and HI WCET. A job can be denoted $p[k]$, i.e., *k-th job of process p. The edges $\mathcal{E}$ are called precedence edges and represent constraints on job execution order.*

Note that this definition of $\mathcal{TG}$ resembles the definition of task graph $\mathbf{T}$ in Chapter 2 except that jobs are now identified by p_i and k_i instead of index j.

The task graph for $\mathcal{PN}$ is derived as follows:

1. Obtain an imaginary process network $\mathcal{PN}'$ where each sporadic process p is replaced by periodic 'server' process p' with period: $T_{p'} = T_{u(p)}$, and priority relation: $\mathcal{FP}' : p' \rightarrow u(p)$.

2. Simulate the job invocation order in $\mathcal{PN}'$ for one hyperperiod, *i.e.,* time interval $[0, \mathcal{H})$, where $\mathcal{H}$ is the least common multiple[3] of T_p in $\mathcal{PN}'$. The simulation results in a sequence of jobs $J = (p_i[k_i])$. Sequence J defines a total order $<_J$. It should respect FPPN semantics,*i.e.,* it should simulate fixed-priority zero-delay execution of jobs according to priority relation $\mathcal{FP}'$.

3. Construct graph $\mathcal{TG}(\mathcal{J}, \mathcal{E})$ where the nodes $\mathcal{J}$ are the elements of sequence J and the edges are defined for a pair of jobs $J_a = p_a[k_a]$ and $J_b = p_b[k_b]$ as follows:

   - $(J_a, J_b) \in \mathcal{E} \Leftrightarrow J_a <_J J_b \ \wedge (p_a \bowtie p_b \vee p_a = p_b)$,
     where:
   - $p_a \bowtie p_b \Leftrightarrow (p_a, p_b) \in \mathcal{FP}' \vee (p_b, p_a) \in \mathcal{FP}'$.

   and the job parameters for job $J_i = p[k]$ defined by:

   - $\chi_i = \chi_p$
   - $A_i = T_p \cdot (k-1)$ and $D_i = A_i + d_p$ if p is periodic
   - $A_i = T_{p'} \cdot (k-1)$ and $D_i = A_i + d_p - T_{p'}$ if p is sporadic

4. To support cyclic non-pipelined scheduling policy, truncate all the required times D_i to the hyperperiod: $D_i := \min(\mathcal{H}, D_i)$.

5. Remove redundant edges by transitive reduction.

Fig. 5.6 shows an example assuming $C_i = 25$ms and ignoring mixed criticality for simplicity.

In this example, $\mathcal{H} = 200$. Every process is represented by $\mathcal{H}/T_p$ vertices. We assumed implicit process deadlines. Since CoefB is represented by its server process, its interval 700 is replaced by the period of its user (FilterB), 200. Also its deadline 700 was first reduced to 500 (subtracting the user period) and then truncated to hyperperiod. InputA has priority over FilterA and NormA, and hence it is joined to both of them. However, in the latter case the edge is redundant due to a path from InputA to NormA.

[3] $T_p \in \mathbb{Q}_+$, so the *lcm* is computed for rational numbers

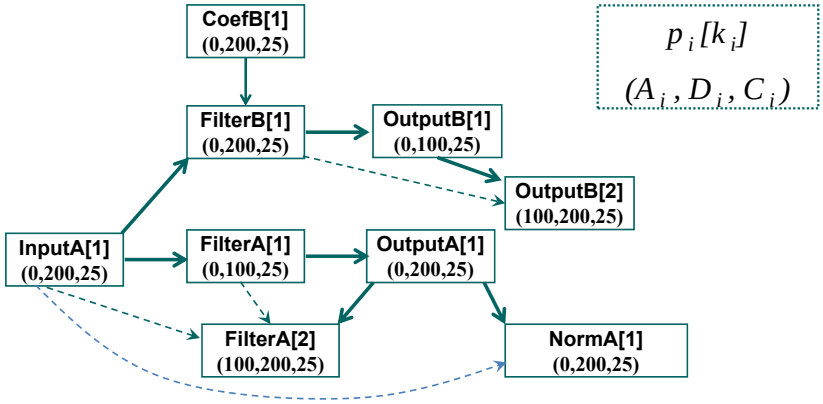


Figure 5.6: Task Graph for the Process Network in Fig. 5.4

5.3.3 Specification in DOL-Critical Language

We specify FPPN applications using *DOL-Critical* language. DOL-Critical is a specification language, a MoC and a tool suite for specifying and scheduling mixed critical multi-core applications. [GSHT] DOL-Critical is not our contribution, but we reuse it for specification of FPPN. DOL-Critical is very much related to FPPN, so that they can be both specified in the same language. However, a comparative analysis between the two MoCs are beyond the scope of this thesis.

To specify a mixed-criticality application in DOL-Critical, we distinguish between two layers: a *functional* layer which consists of processes and data channels, and a *control* layer which consists of process event generators and process functional priority. The specification of each process contains source code and its execution times, while the process event generators (one per process) specify the processes' activation patterns and deadlines. For the specification, DOL-Critical uses two distinct languages: C/C++ to program the process functionality, and XML for the process attributes (period, deadline, criticality), for the process connections through data channels and for functional priority.

An example of a DOL-Critical process can be found in Figure 5.7. A process has an internal state data structure, an initialisation subroutine, and a subroutine defining one execution of a job. In the DOL-Critical application programming interface (API), these are denoted `<Process>_state`, `<Process>_init()`, and `<Process>_fire()`, respectively. Furthermore, the API supports two main functions for the communication between processes: `DOLC_read()` and `DOLC_write()` (see Figure 5.7 for an example). These functions enable reading/writing from/to a data channel and have different semantics depending on the type of the target data channel. A detailed presentation of the API as well as XML templates for the specification of mixed-criticality applications in DOL-Critical can be downloaded from the site of DOL-Critical tool suite [GSHT].

5.4 Compiling the MoC and the Policy into BIP

The time-critical software consists of functional code and middleware, the latter providing elements for communication, synchronization that belong to the given MoC and real-time scheduling policy. Compiling means translating the functional code and middleware specification into components of BIP language and connecting them with each other. The compo-

```
01 struct Square_state {
02   int index;
03   int length;
04 };
05 struct DOLCData {
06   bool valid;
07   float value;
08 };
09
10 void Square_init(Square_state *ST) {
11   ST->index =0;
12   ST->length = 200;
13 }
14
15 void Square_fire(Square_state *ST, int mode) {
16   DOLCData x,y;
17
18   if (mode == DEGRADED) {
19     return;
20   }
21
22   if (ST->index < ST->length ) {
23     DOLC_read ("pIN", &x, sizeof(float));
24     if (x.valid) {
25       y.value = x.value * x.value;
26       y.valid = true;
27       DOLC_write ("pOUT", &y, sizeof(float));
28     }
29   }
30   ST->index = ST->index + 1;
31 }
```

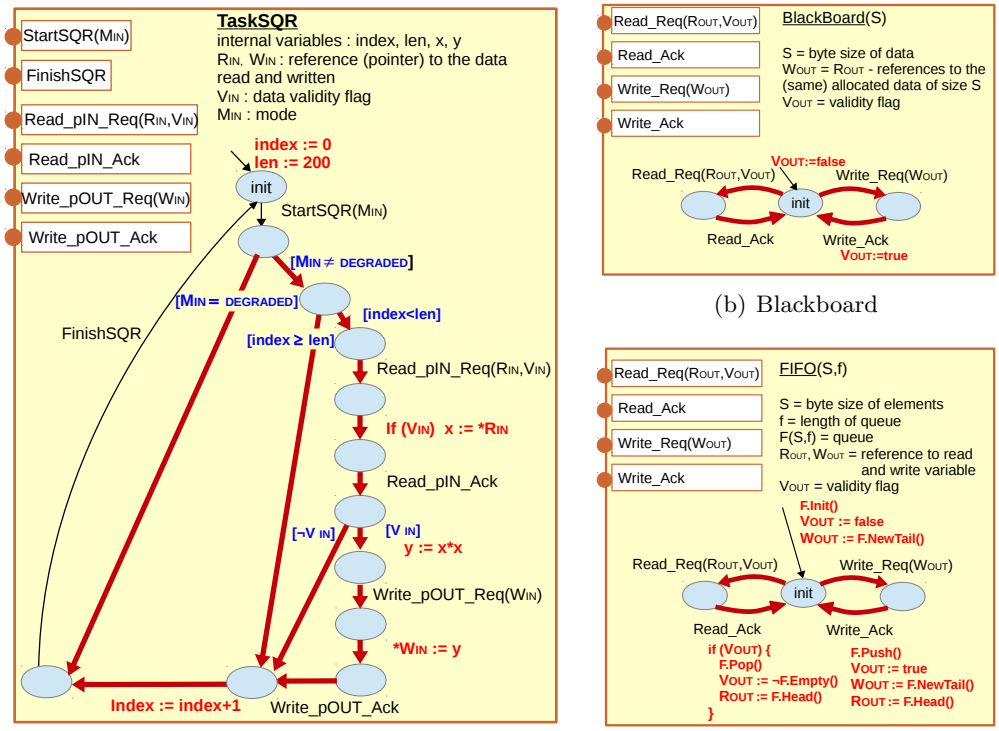
Figure 5.7: C source code for process `Square` Example

nents express the correct timing behavior by timing constraints and transitions. A situation where for a component automaton no transitions are possible anymore in future is called local deadlock and is detected as a runtime error. The BIP components generated at compilation are constructed in such a way that a deadlock indicates that either the hardware resources cannot handle the workload on time or that the workload does not conform to specification. For example, in Figure 5.2, component "PeriodicA" is ready to execute an interaction at port "StartA" only when $x = T_A$. If at this moment of time both "CPU"components are busy executing the previously started "(Task)" transitions, then component "PeriodicA" will deadlock as the clock x will continue counting the time, never come back to T_A. To avoid a deadlock in "PeriodicA", at least one of the "CPU" components should be ready for interaction at periodic instances in time: $T_A, 2T_A, 3T_A, \ldots$. Similar conditions hold for certain BIP components generated at compilation.

5.4.1 Compiling the processes

The BIP model of a process is automatically extracted from its source code. For example, the code of the `square` process in Figure 5.7 is compiled into the BIP automaton shown in Figure 5.8(a). The local state variables of a DOL-Critical process become internal data variables of the BIP component. The initial transition implements the '⟨*process*⟩_init()' subroutine. The rest of the process component implements the source code of the process's job, *i.e.,* the '⟨*process*⟩_fire()' subroutine (DOL-Critical API). We enwrap the job execution between process start and process finish interactions ('Start/Finish_⟨*process*⟩'). They are used both to enable the job executions upon their activation by the corresponding DOL-Critical controller and to delay them until the scheduled time by TTS containers (*e.g.,* Figure 5.9).

When translating the '⟨*process*⟩_fire()' subroutine to a BIP model, the source code is parsed, searching for primitives that are relevant for the interactions between the process and the other components of the system. The relevant primitives are calls to 'DOLC_read()' and



(a) 'Square' process example compiled to BIP

(b) Blackboard

(c) FIFO

Figure 5.8: Compiling Processes and Data Channels to BIP

'DOLC_write()' for reading/writing from/to the data channels. We see that the behavior of the resulting automaton is consistent with the behavior of the original source code, whereby the interaction primitives are replaced by patterns with interactions via BIP ports. As shown in Figure 5.8(a), the pattern for 'DOLC_read()' and 'DOLC_write()' consists of three transitions: (i) request ('Req'), (ii) data-copying, and (iii) acknowledgment ('Ack').

Let us consider reading data for example. First, we have an interaction 'Read_⟨*port*⟩_Req', which is an interaction requesting access to the channel via the DOL-Critical port '*port*'. In the corresponding interaction, the process receives from the data channel a reference 'R_{IN}' to the memory area from where it can read and a validity flag 'V_{IN}'. The next transition copies the data from the provided reference to the local variable to effectuate the data reading, and the third transition acknowledges the success of the read operation. Writing is performed in a similar way.

5.4.2 Compiling Channels

According to the process-to-channel connection topology specified in the XML files, BIP connectors are inserted between 'Read/Write_⟨*port*⟩_Req/Ack' at the process and the 'Read-/Write_Req/Ack' ports at the data channel components.

Recall the DOL-Critical data channels introduced in Section 5.3.1. A basic notion of the supported data channels is the validity flag. The meaning of this flag is availability of data, given the non-blocking nature of read and write operations in DOL-Critical. A blackboard channel represents a shared variable and a FIFO is a queue buffer.

Figure 5.8(b) shows the model for a blackboard. At the initial transition, we (implicitly)

allocate a user-type variable of given byte size. Read (Write) operations are separated into request and acknowledge transitions, coherently to the process model of Figure 5.8(a). During the request the blackboard communicates to the process the memory address, from (to) which it should read (write). In case of a read, the validity flag is communicated as well.

The BIP model of a FIFO is shown in Figure 5.8(c). It is similar to blackboard, but instead of allocating a scalar user-type variable, the component initially creates a queue, i.e., a circular buffer, of user-type elements with a given capacity ('length'). Read (write) operations on a FIFO give the address of the tail (head) of the queue.

5.4.3 Compiling the Scheduling Policy

Our BIP run-time environment (RTE) currently does not support the interruption of running transitions. Therefore, in our current middleware for time-critical systems we do not yet support preemption. It should be noted that many multi-core platforms choose to not support preemption, instead providing a large number of cores to ensure sufficient degree of multi-threading concurrency without preemption.

We demonstrate programming the mixed critical scheduling policies in timed automata extended with tasks by considering a policy that combines static-order execution and time-triggering [GSHT13]. For a given task graph $\mathcal{TG}$ as defined in Section 5.3.2 TTS defines a cyclically repeating non-preemptive schedule with cycle time equal to the hyperperiod $\mathcal{H}$. The cycle is split into a certain number of frames, which also have fixed, but possibly different time-lengths $\mathcal{L}_f$, where $f = 1, 2, \ldots$ is the frame index. All the nodes of $\mathcal{TG}$ –jobs J_i – are distributed between frames. The frame for a job should be selected such that it fits inside the job scheduling window $[A_i, D_i]$ and this window is effectively reduced to the frame boundaries. The schedule partitions the processes between the cores, *i.e.,* all jobs $p[k_i]$ of the same process 'p' are mapped to the same core μ_p, there is no process migration.

In dual-criticality system case (which is currently assumed by our compiler), every frame is partitioned into two subframes of flexible time-lengths, where jobs are executed on each core in static order. In the first subframe only HI-criticality jobs are executed, in the second one only LO jobs. At the end of the first subframe all cores synchronize on a barrier. Let the relative time *w.r.t.* begin of the frame be t. If $t > \mathcal{B}_f$ then the LO jobs in the second subframe execute in degraded mode. If $t \leq \mathcal{B}_f$ then they execute in normal mode. $\mathcal{B}_f$ is estimate of the first subframe total response time when the jobs execute for their C_i(LO) execution time. Task dependencies are handled as follows. If $(J_a, J_b) \in \varepsilon$ then either J_b should be in a later (sub)frame then J_a or it should be mapped to the same core, later in the sequential order. See [GSHT13] for further details.

Consider the example in Figure 5.9. The figure shows a partial TTS schedule for an application with processes denoted 'A', 'B', 'C', *etc.* In our models, we use notation '$f[k]$' to denote the k-th sub-frame and 'L$\langle f \rangle$' (*i.e.,* L1, L2, ...) to denote the frame duration $\mathcal{L}_f$. We use 'Bar$\langle f \rangle$' to denote $\mathcal{B}_f$. Depending on whether the actual runtime length of the first sub-frame respects this barrier or not, the processes in the second sub-frame will run in normal or degraded mode. This is the main mixed-criticality runtime mechanism we aim to reflect in the generated BIP components.

To the right of the Gantt chart in Figure 5.9, we show a (slightly simplified) general structure of the 'Frame$\langle f \rangle$' component, taking 'Frame1' as example. This component controls the mode '$\mathrm{M_{OUT}}$' of execution of the two sub-frames contained in the frame. Initially the mode is set to 'normal'. When frame f is about to start, interaction 'BeginF$\langle f \rangle$' ('begin

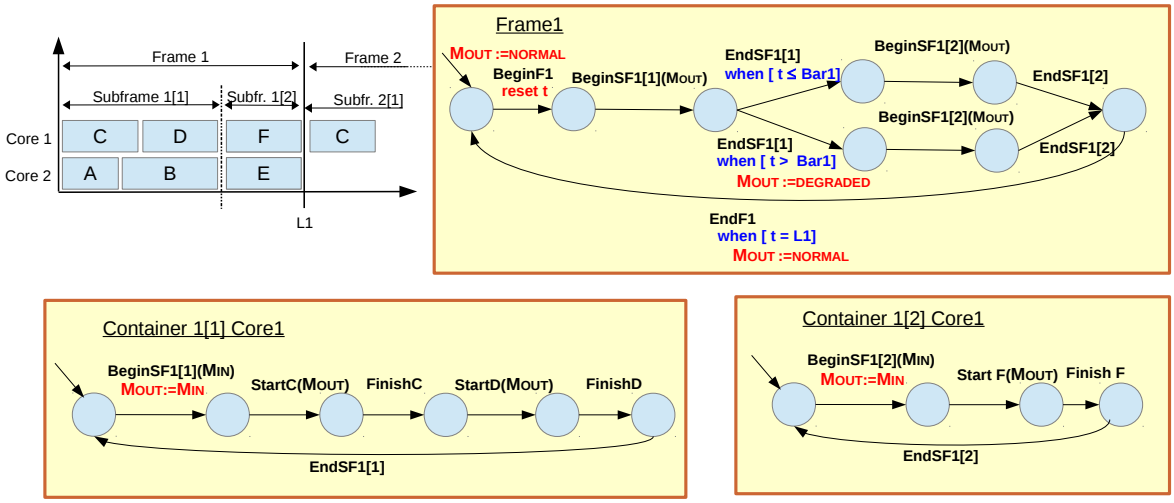


Figure 5.9: TTS Scheduling Frames in BIP

frame f') gets enabled. At this point we reset clock t so that it measures the elapsed time in frame f. Then, we signal the begin of sub-frame $f[1]$ via interaction 'BeginSF$\langle f\rangle[1]$'. At the moment when the sub-frame finishes, the interaction 'EndSF$\langle f\rangle[1]$' gets enabled, and we check the elapsed time t. We keep the normal mode if t does not exceed barrier 'Bar$\langle f\rangle$', otherwise the mode is set to degraded. After executing the second sub-frame, the frame finishes, which is signalled via 'EndF$\langle f\rangle$'.

Examining this component, we conclude that it is free from local deadlock provided that the schedule is correct and the processes scheduled in the frame finish their execution by time 'L$\langle f\rangle$'. Otherwise the component will be blocked forever at the origin of transition 'EndF$\langle f\rangle$'.

The two components given at the bottom of Figure 5.9 are *Containers*, which are in charge of triggering jobs' execution according to the given TTS schedule. The container components are specific per sub-frame $f[k]$ and core. They trigger jobs according to the corresponding sequential schedule. In the figure, the left component implements the sequential schedule assigned to Frame 1, Sub-frame [1] on Core 1, which executes first a job of process 'C' and then of process 'D'. Therefore, in this component we see a chain of transitions that start and finish these jobs. By convention, we use the notation 'Start_$\langle process_name\rangle$' for the job start interaction, and a similar notation for the job finish interaction. For synchronization with the frame component, the sequence of calls to the jobs is enwrapped in 'BeginSF/EndSF' interactions. At 'BeginSF', the frame component transmits the value of variable 'mode', which is passed through to the process components via the 'Start' interactions.

In Figure 5.10 we show how frames and containers are connected to each other. There is a 'Cycle' component, which just executes a cyclic 'Begin/End' sequence. The 'begin' of a cycle triggers the execution of all frames in the cycle in the order of their index f, whereby we join the 'end' of frame f to the 'begin' of frame $f+1$. In the given example we assumed two frames per cycle. For every sub-frame the 'begin' and 'end' connectors join together all the containers for the specific sub-frame on Core 1, Core 2, ... Therefore, the employed 'barrier' mechanism to synchronize the cores at frame and sub-frame boundaries is a multi-party BIP interaction.

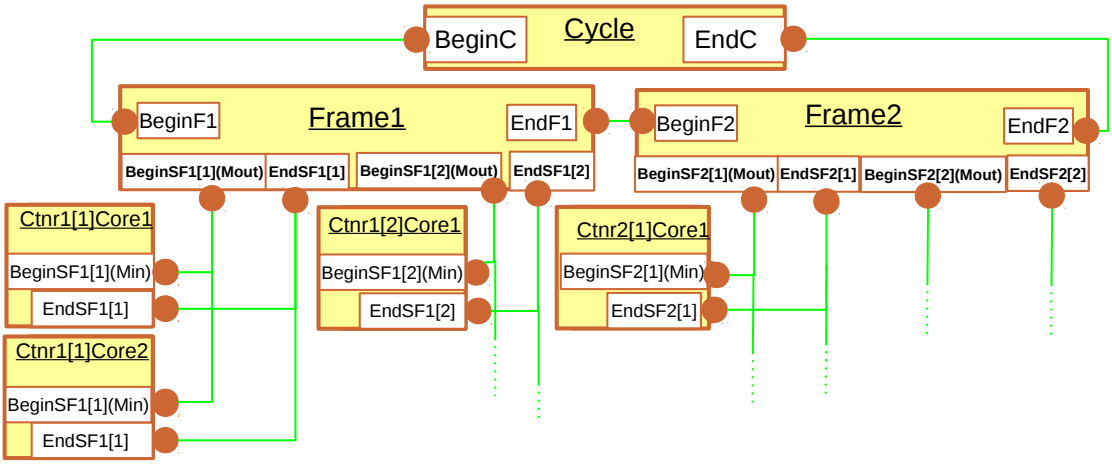


Figure 5.10: Composing Cycle, Frames and Containers

5.4.4 Periodic Server for Sporadic Processes

In Figure 5.11 the periodic server is shown, which is a supplementary adaptor for connecting sporadic processes to cyclic schedulers. This component manages a queue of active jobs. When a job is activated, it is inserted in the queue, and it is removed when it is scheduled. For simplicity we assume that it is scheduled once per minimal inter-arrival in the presented model. The queue may contain "false" jobs. This is used for reconciling cyclic schedulers and sporadic tasks. We explained before that in TTS container we execute jobs in a sequential way. If a frame contains a sporadic job, and this job does not activate, we will have a deadlock in the TTS container. Thus, to avoid this problem, whenever a sporadic process is not activated, we introduce in the queue a "false job" with zero execution time. The bottom sub-component of the periodic server in Fig. 5.11 distinguishes between "active" and "false" jobs. In the case of an active job, it signals the job start to the scheduler TTS container, then to the process, then it waits for the job termination and finally signals it to the container. In case of a false job, the execution of the process job is skipped. A more detailed explanation of the employed method of handling sporadic jobs by a periodic server can be found in [PSPBB15].

Fig. 5.12 shows how scheduler containers and a sporadic processes are connected. The Event Generator generates the activation signal and sends it to the Periodic Server, which triggers the Process in the order defined by the frames. For a periodic process, the periodic server is not necessary and the process can be connected directly to its containers and generator.

5.4.5 Compiling the Event Generators

We describe here the Event Generator component, individual for each process. The main purpose of this component is to enable the start of jobs after their activation. It also manages the "false" activation for sporadic jobs. The idea of the latter is that for sporadic process p at small intervals $\delta = T_p/K$ for some integer K the environment is polled for the need to activate the sporadic process by calling some platform-dependent subroutine *protocol*() that returns a Boolean value indicating activation ('true') or false activation ('false'). The point is that to ensure functional determinism in FPPN MoC, at each moment of time when a

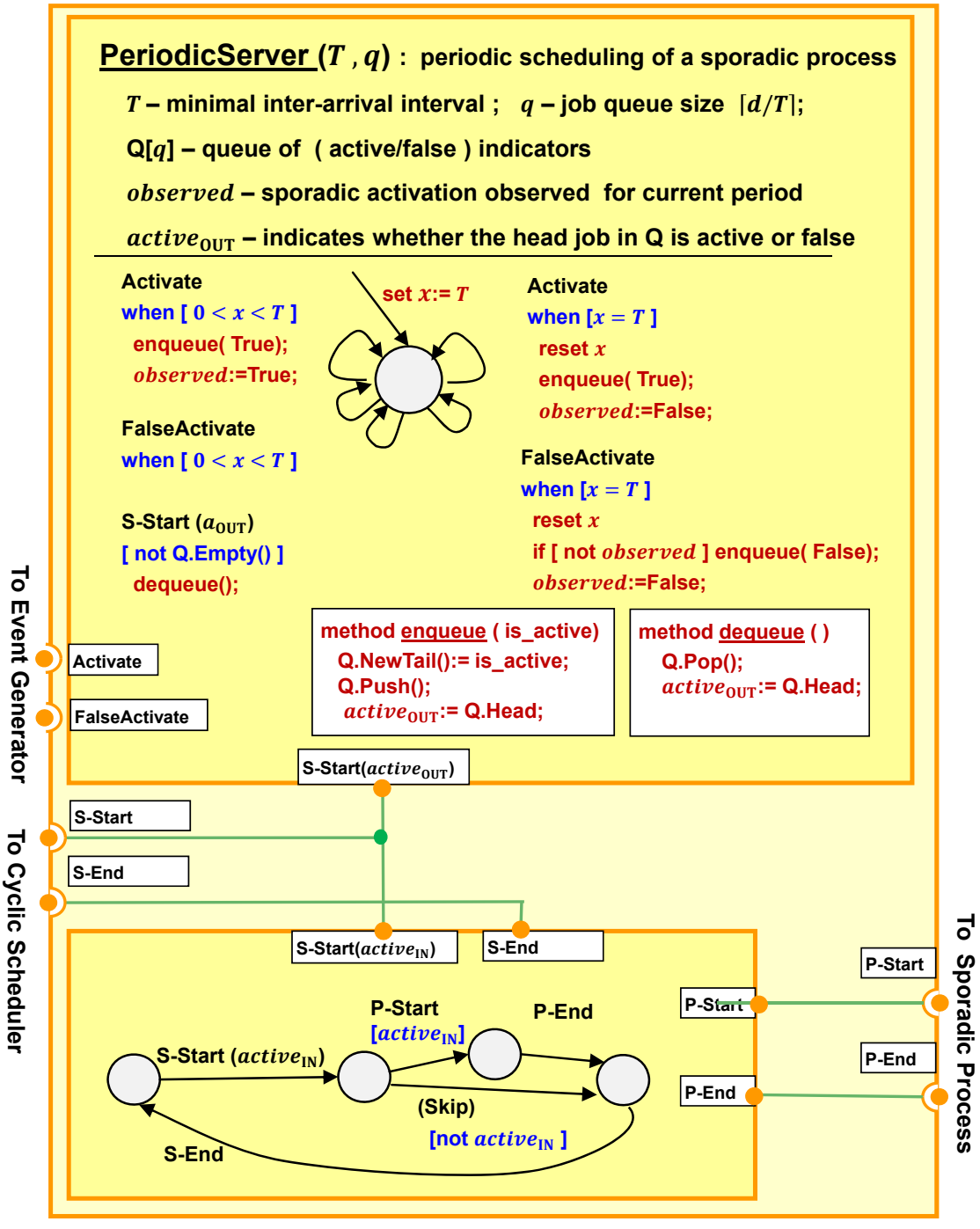


Figure 5.11: Periodic server for sporadic processes

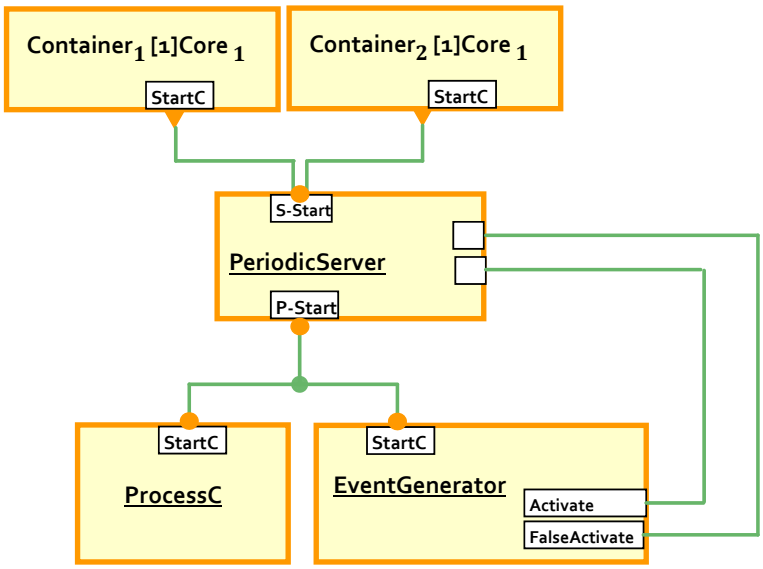


Figure 5.12: Connection between a Sporadic Process and its Scheduler

sporadic process may potentially get activated it should be always explicitly signaled whether it is activated or not. For periodic processes $\delta = T$ and $protocol()$ always returns 'true'. For a sporadic process we may have for example $\delta = T_p/10$ and $protocol()$ returns true at most once per 10 calls.

The event generator is shown in Figure 5.13. It contains a few subcomponents. The Source triggers periodically or sporadically (depending on its protocol) the activation signal. The Sink checks whether any job misses its deadline. For this it uses a "Latency - Burst Shaper" component, which can be seen as a delay line of delay d_p and capacity up to q_p events, where deadline d_p and queue size q_p are process parameters. At activation, the burst shaper starts a new timer (a clock), and when the deadline time has elapsed it enables the output. As shown in Figure 5.14, the sink checks whether at the end of its deadline interval, when "Meet" and "Miss" are enabled, the oldest pending job is not running anymore, in which case "Meet" is executed. The component goes into local deadlock state if deadline is missed (and this leads to runtime error).

The "Throughput – Burst Shaper" – ensures that the source cannot activate the jobs more than once per time T_p. This subcomponent can be omitted for periodic processes.

The implementation of Source is shown in Figure 5.15. This component polls the "protocol()" at periodic intervals δ, as explained earlier. Depending on $protocol()$ it executes "Activate" or "FalseActivate", which is needed for periodic server.

Figure 5.16 shows how a Burst Shaper is implemented. Its main purpose is to limit the amount of burst to at most σ events per time P where σ and P are given during the definition of the component. This component also signals when an event that arrived P time units ago has elapsed (terminated). The main idea of implementation is to use a queue of clock variables implemented as circular buffer.

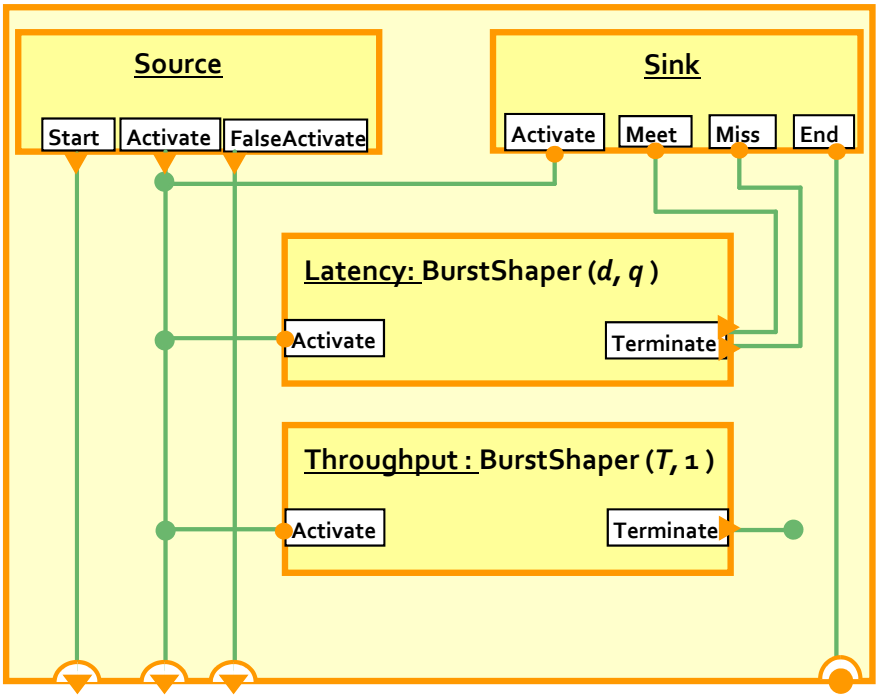


Figure 5.13: Event Generator

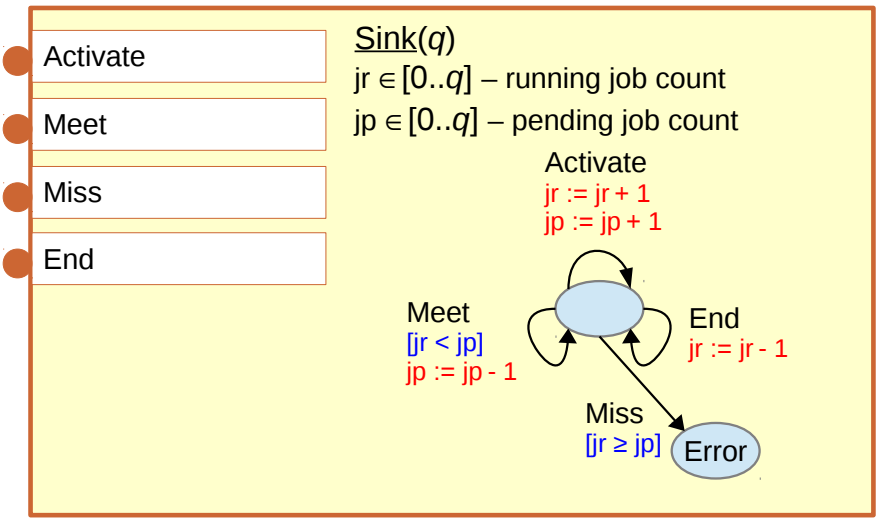


Figure 5.14: Sink

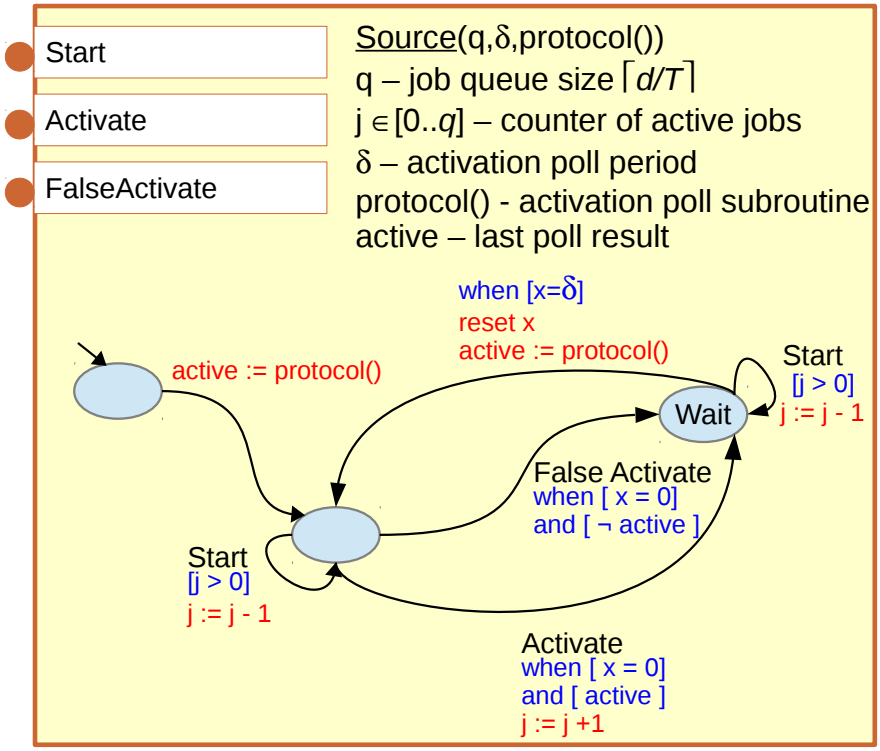


Figure 5.15: Source

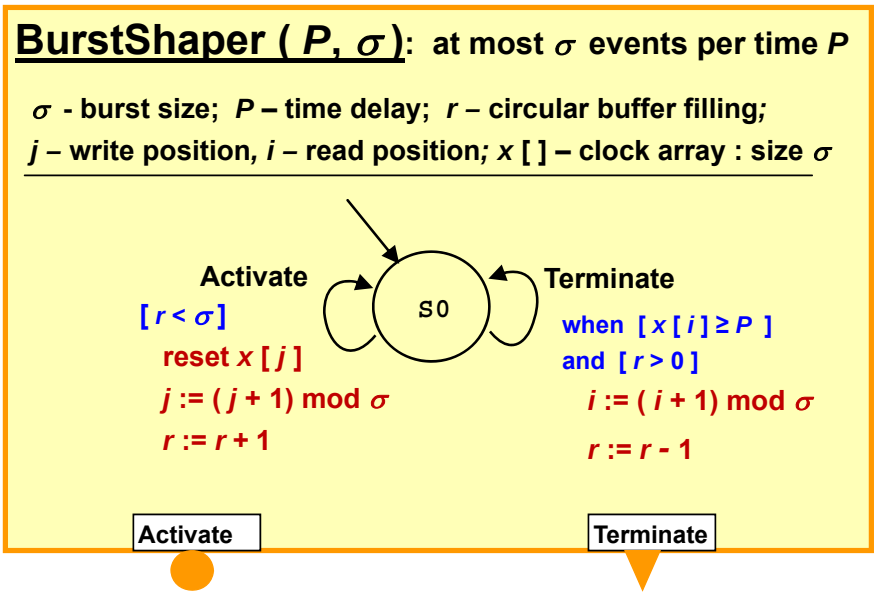


Figure 5.16: Burst Shaper

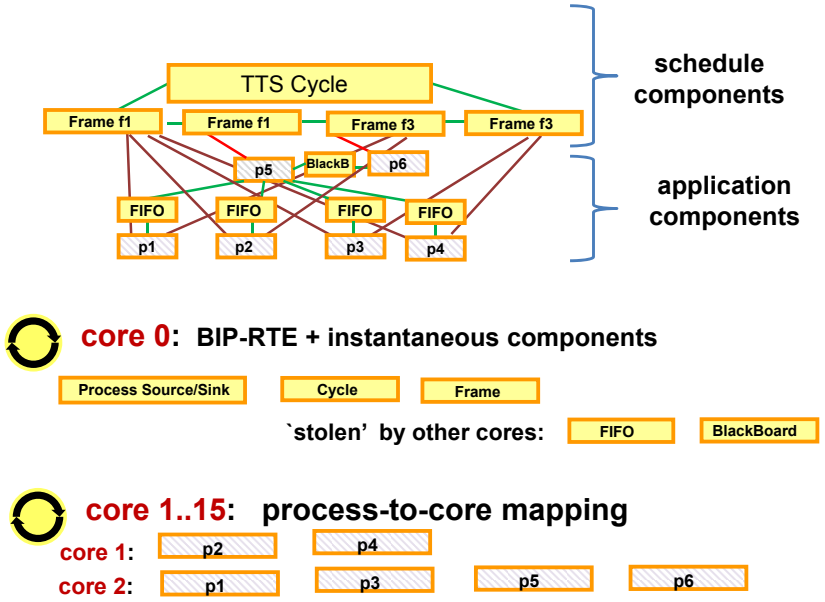


Figure 5.17: Distributing BIP Components between Cores

5.5 Implementation and Experiments

5.5.1 Run-Time Environment

As illustrated in Fig. 5.17, after compiling the application and TTS schedule into BIP, the BIP design can be partitioned into the schedule components and the application components. The components are joined by BIP connectors, through which they can perform interactions with each other. The application components include the components dedicated to DOL-Critical processes, denoted p1, p2, . . . , and data channels, denoted BB, FF, depending on the type: blackboard and FIFO. The schedule components include one component that models the schedule cycle and a set of components that model frames. The schedule components are connected to the application components to coordinate their execution according to the schedule. The schedule also provides the process-to-core mapping, which is used to generate component-to-thread mapping, as illustrated in the bottom part of the figure.

Deployment

We implemented our framework on the Kalray MPPA manycore architecture, inside a single shared-memory compute cluster. The cluster provides 16 processor cores, each one running one POSIX thread. The BIP executable is coordinated by an adapted version of multi-thread BIP RTE engine originally described in in [TCBS13] and available at [PBS$^+$]. The main improvements made compared to the original version are support of self-timed transitions, arbitrary component-to-thread mapping, mapping of components to RTE engine thread, and the "stealing" the self-timed interactions from RTE engine to execute them faster. The latter two features permit to significantly reduce the RTE engine overhead.

In our framework we can use up to 16 cores, whereby Core 0 is reserved for BIP RTE

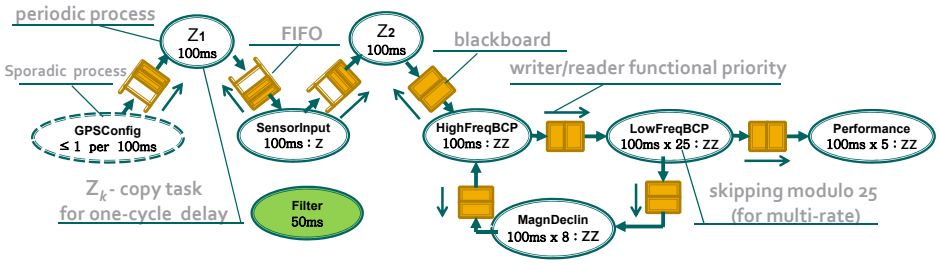


Figure 5.18: Flight Management System FPPN model

engine. The engine schedules every interaction according to the semantics of BIP. Therefore the components have to notify the engine about their interactions and wait until there are scheduled. Note that self-timed transitions can be executed by components themselves, without involving the engine. Also the self-timed interactions are "stolen" executed without the engine. For example, 'reads and writes' to the channels are self-timed and hence are stolen.

On Core 0, next to BIP RTE, we map all "instantaneous" components, i.e. all components except processes, which execute only "lightweight" instantaneous transitions. Cores 1-15 are allocated for the process-to-core mapping computed by the offline scheduler tool in DOL-Critical. Unlike instantaneous components, these components may execute self-timed transitions.

5.5.2 Case Study: FMS Application

To demonstrate the applicability of the complete design flow, we employ an industrial representative implementation of a flight management system (FMS) [DFG+14].

The FMS is a safety-critical embedded avionics system, responsible for aircraft localization, flightplan computation for the auto-pilot, detection of the nearest airport, *etc.* In this experiment we look into a sub-system of the FMS. Figure 5.18 shows the corresponding DOL-Critical application, which is responsible for calculating the best computed position (BCP) and predicting the performance (*e.g.,* fuel usage) of the airplane, based on periodically collected sensor data and sporadic configuration commands from the pilot, e.g., for configuring the Global Positioning System (GPS).

Specifically, after being pre-processed by process 'SensorInput', the input data are processed by process 'HighFreqBCP'. Then, they arrive at process 'LowFreqBCP', which post-processes the data at low frequency, and makes them available to other sub-systems of the FMS. 'LowFreqBCP' also provides the results to a feedback loop that takes into account the magnetic declination for computing the airplane position.

All depicted processes are periodic except for the sporadic process 'GPSConfig', which can execute at most once in any 100-ms interval. All periodic processes of the FMS are specified with period 100 ms. However, some of them contain in their C code a wrapper to skip the processing at all but every n-th job, to represent processes with original period $n \cdot 100$ ms. This is done for two reasons: (i) to reduce the effective hyperperiod $\mathcal{H}$, (ii) to comply with the DOL-Critical offline scheduler requirement for equal period among processes with dependencies. Note that keeping the original $\mathcal{H}$ (in the FMS case, equal to 40 seconds) would result in generating hundreds of frames and container components for the TTS scheduler in BIP, which would lead to unfeasible memory requirements for the implementation on a single

Process	Criticality Level	Period [ms]	LO WCET [ms]	HI WCET [ms]	RTE Access Count
Filter	LO	50	32	2	3
SensorInput	HI	100	1	26	3
GPSConfig	HI	100	1	21	4
HighFreqBCP	HI	100	1	11	3
LowFreqBCP	HI	100	1	11	3
MagnDeclin	HI	100	1	11	3
Performance	HI	100	1	11	3
Z1	HI	100	1	26	3
Z2	HI	100	1	26	3
Cycle_Begin	HI	100	0	0	10
Frame_Begin	HI	50	0	0	4
Subframe_Bar	LO	50	0	0	2

Table 5.1: FMS process execution profiles

MPPA®-256 cluster.

The given process structure originally allowed only limited parallelism due to the functional-priority branching from 'LowFreqBCP' to 'MagnDeclin' and 'Performance'. To introduce pipelining parallelism, we inserted two new processes, denoted as Z_1 and Z_2. These copy input data to the output, thus ensuring double-buffering, which is required for pipelining. Because each inserted Z_k process leads to an additional data-propagation delay of one period, this delay is subtracted from the deadlines of the processes that follow in the process chain, which, therefore, should be sufficiently large.

All processes of the FMS sub-system are used to calculate critical information, i.e., the current position of the airplane. Therefore we assign criticality level HI to them. The execution profiles of the processes are shown in Table 5.1. The processes are protected from exceptional execution times overruns (due to potential faults and fault correction) by defining a significantly more pessimistic execution profile at level HI than at level LO. Not having WCET tools for the MPPA®-256 platform at our disposal, we derive LO worst-case execution times based on extensive measurements. For the HI estimates, we augment the LO bounds by a margin of 10 up to 25 ms, which also makes them at least 10x larger. We introduce a possibility to simulate fault injection, by programming an optional prolongation of the process execution by up to the HI execution time through an additional dummy loop in the C code.

Table 5.1 includes also the bounds on *RTE engine accesses* for each process. This is because the scheduling policy and other control interactions are handled by centralized RTE engine, which is thus a *shared resource*, and DOL critical tools can take shared resource access counts in TTS-policy response-time analysis. For the periodic processes, we observe that their execution causes always exactly three interactions: Start, Finish and deadline check (the latter is done in fact in the generator). Sporadic processes cause one extra interaction, which is related to the periodic server. Note that when counting BIP interactions, we neglect self-timed interactions, as they do not lead to RTE engine accesses.

Table 5.1 includes also three *virtual processes*, whose purpose is to account for RTE accessed that cannot be attributed to regular processes. such as RTE accesses from scheduling components. Note that the virtual processes account not only for the TTS components such as cycle, frames, and containers, but also for other components that cause BIP interactions at the boundaries of the cycle, frame, and sub-frame, respectively. For example, at the beginning of each cycle all eight periodic processes get activated by their generator, which

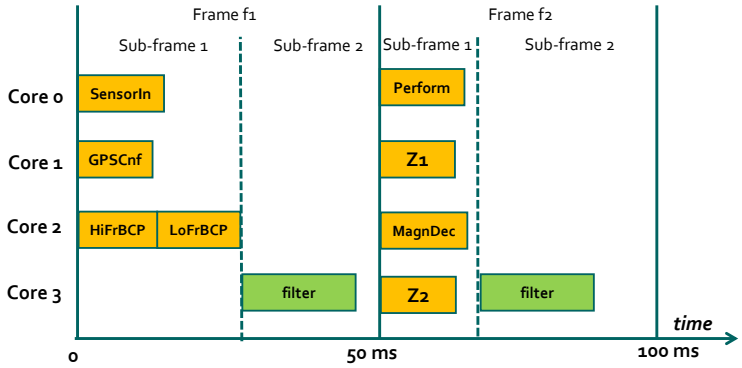


Figure 5.19: Optimized Static TTS Schedule for the FMS sub-system

explains the high access count of the virtual process 'Cycle_Begin'.

Through extensive measurements on the MPPA®-256 platform (again, due to non-availability of suitable WCET tools), we derived a (pessimistic) upper bound on the BIP RTE-engine delay per access, which amounts to $T_{acc} =$ 0.42 ms. It is used by DOL-Critical tools to quantify the shared resource overhead.

Finally, since the considered sub-system of FMS includes only processes of criticality level HI, to obtain a dual-critical application we added an artificial periodic process called 'Filter', with period 50 ms. It models some digital signal processing functionality, considered as a less critical LO process. Since 'Filter' is low-criticality, we model two execution modes: *normal* and *degraded*. Specifically, 'Filter' executes a loop resembling a digital filter, the number of loop iterations being significantly lower in degraded mode, to represent the possibility of providing a reduced level of quality for a smaller number of digital filter coefficients.

5.5.3 Case Study: Design Flow Results

For the FMS sub-system, the maximal degree of parallelism is four (three pipeline stages and one branching). Therefore, we choose to allocate a subset of five MPPA®-256 cores: four for task execution and one for the BIP RTE engine. For the mapping and scheduling optimization, we provide the DOL-Critical specifications of the FMS sub-system and the 5-core subset of the MPPA®-256 cluster to the DOL-Critical scheduling tools (see [GPS+] for details).

In the resulting schedule, the TTS scheduling cycle has a period of 100 ms (equal to the hyper-period of the tasks) and it is divided into two frames, each with a fixed length of 50 ms. TTs schedule is illustrated in Figure 5.19.

The optimized TTS schedule for the FMS sub-system, along with the application specification, are compiled into BIP automata, as described in Section 5.4. Functional correctness is validated through simulation, and code is automatically synthesized for the deployment on the MPPA®-256 platform (subset of 5 cores within a cluster). Figure 5.20 presents Gantt charts of the FMS execution traces on the MPPA®-256 for three alternative scenarios. Each chart depicts six consecutive TTS scheduling cycles.

'LO' and 'HI' scenarios represent corner-cases for timing analysis, where all tasks activated simultaneously (which happens on the hyper-period boundaries) and according to their maximal execution times at the given level. The 'ordinary' scenario represents a possible execution of the system, where periodic tasks skip some periods due to pipelining and original

multi-rate periods, and the sporadic task is activated by some arbitrarily chosen (encoded in DOL-Critical) protocol. In this scenario, we simulated some fault injections in tasks 'Z1', 'Z2', 'HighFreqBCP', and 'SensorIn' in the fifth scheduling cycle (between 400 and 500 ms). Note that the tasks take considerably longer to execute in this cycle, with their execution time being close to their HI profile in Table 5.1. This triggers a HI execution scenario, which results in providing degraded service to the lower-criticality 'Filter' task in both frames of this cycle. In degraded mode, 'Filter' runs for approximately 2 ms instead of the usual 32 ms.

The experiments in [GPS+] also present the results of response time analysis in different scenarios, taking into account the shared RTE-engine resources. The results show good tool accuracy and necessity of taking runtime overhead into account.

In summary, the deployment of the FMS sub-system on the MPPA®-256 illustrates and validates various novel attractive features of our design flow for the implementation of mixed-criticality systems on commercial multi-core architectures. Based on this first evidence, we are convinced that the presented design flow can provide a viable foundation for the rigorous design of mixed-criticality systems, with potential to be applied to complex industrial-scale settings.

5.6 Chapter Summary

In this chapter we have presented a design flow for deployment of mixed critical applications on multicores. It is based on compiling the application and policy model into timed automata extended with tasks. Such an approach can potentially address the challenge of the lack of consolidation in real-time systems programming and scheduling, which is particularly pronounced in the case of multi-core scheduling and scheduling with runtime resource-management mechanisms, such as Vestal's approach for mixed criticality. Instead of trying to design an operating system with native support of a wide spectra of various models of computation and run-time policies and instead of low-level programming of middleware, we propose to compile high-level models into task automata and use them for both deployment and validation. We have demonstrated this concepts in a concrete design flow example, where special emphasis was put on mixed criticality with Vestal approach, through using different policies than the ones proposed in this thesis.

The concept of timed-automata middleware was first published in Contribution [7] (see Contribution section), whereas in [1] we extended it to mixed criticality. The synchronous language related MoC for real-time multiprocessor system called FPPN was first described in [2].

As future works we are planning to extend the design flow in order to test it with different MoCs and scheduling policies. In particular we are interested in implementing the scheduling policies described in Chapter 3 and 4, which would allow a better processor utilization. This would require our task automata language and its RTE environment to be extended with explicit support of task preemption and dropping. Also we are interested into test the design flow into other multicore computational platforms and implementing other case studies.

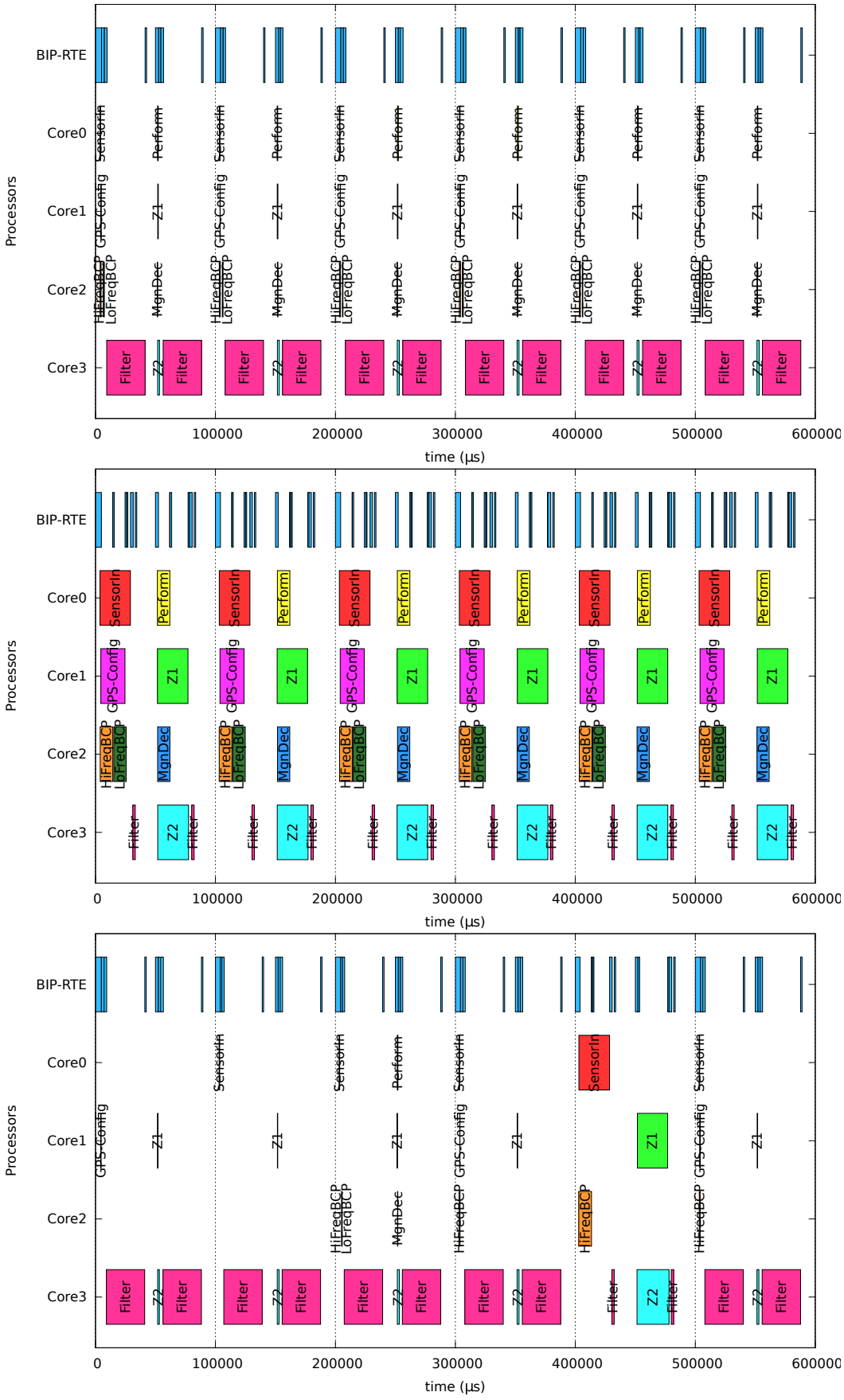


Figure 5.20: FMS Test Case: 'LO-scenario','HI-scenario', and 'Ordinary' Traces on MPPA®

Chapter 6

Conclusions

6.1 Thesis summary

Mixed Criticality Systems are a novel and challenging research area in the context of real-time systems. We discussed motivations and challenges of this new research topic in Chapter 1, together with a quick overview of the state of the art. Also we discussed the migration of embedded systems from single to multicore platforms, explaining both technical and economical reasons behind this technological shift. The next generation embedded systems will be multicore and mixed critical, and their design presents a lot of challenges. The work done in the context of this thesis is an effort towards solid design of such systems.

Our main focus was on scheduling, that is the most relevant area of research in MCS. We claim this for two main reasons. First, as shown by Baruah [Bar09], the problem is highly intractable and that traditional techniques may not be successfully applied [BV08], thus there is the need of novel scheduling techniques. Second, in some applications using more computational power by using more powerful processors or by increasing the clock frequency, in order to schedule more workload, is not possible in some modern applications like UAVs, that have very strict requirements in terms of area, size weight and power consumption. Another research topic taken into consideration was to construct a design flow to produce easily analyzable mixed critical-systems.

To simplify the problem we considered finite set of jobs with two criticality levels. We preferred the job set model instead of the more popular task model because in synchronous systems it enables potentially better processor utilization and it simplifies the analysis of the system from a theoretical point of view, especially in the case of multi-cores and data dependencies between the tasks. Its main drawbacks, such as being forced to consider an entire hyperperiod, were considered acceptable for two main reasons:

1. Since we are interested in simplicity, we addressed the problem of Time-Triggered scheduling, since this scheduling strategy allows an easier extension to auxiliary analyses of inferences on other media than cores (such as buses) by limiting the number of possible combinations of jobs running concurrently. In time triggered scheduling one is forced to consider the whole hyperperiod.

2. The hardware and software components of a real-time embedded system hardly have periods that can be divided by big prime numbers, in order to avoid the hyperperiod to grow.

The scheduling model adopted in the context of the thesis is formally described in Chapter 2. Our formalization is compliant to Vestal model [Ves07]. We introduced a new priority-based scheduling classification, *Fixed Priority per Mode* (FPM), that uses a different priority table for each criticality mode, in opposition to the classical fixed-priority scheme. We also introduced the load metric, and extended it to be more suitable to model the workload of multiprocessor systems. Then we extended the model and load metric to systems with job dependencies. In our model also dependencies from low critical to high critical jobs are allowed. Even if this approach is not popular in literature for obvious reasons, it is sometimes needed by industrial applications. To better model such problems we introduced the concepts of ASAP arrival times, ALAP deadlines and high-criticality and mixed-criticality task graph.

In Chapter 3 we gave contributions to the priority-based scheduling of MCSs. We started by describing the limitations of one of the most used techniques: "Audsley approach", to motivate the study of a new formal way of describing the interaction between jobs, the *Priority Direct Acyclic Graphs* (P-DAGs). We then proposed two scheduling algorithms based on the P-DAG. The first is *Mixed Criticality Earliest Deadline First* (EDF), a priority based algorithm for single processor. We formally proved that MCEDF dominates the state of the art Audsley approach based algorithm *Own Criticality Based Priority* (OCBP) while having a lower computational complexity. The second algorithm is called *Mixed Criticality Priority Improvement* (MCPI). This algorithm can be applied to multiprocessor instances with dependency constraints. To the best of our knowledge no algorithm to solve this kind of problem has been proposed in literature, if not under specific constraints. We formally proved that MCEDF and MCPI are equivalent when applied to single processor instances with no job dependencies. An interesting theoretical result is that both MCEDF and MCPI are optimal among all algorithms that put HI-critical jobs in EDF order. We concluded the chapter by showing an experimental evaluation of the two algorithms. MCEDF is compared with OCBP, and reduces, in random job experiments, the non schedulable instances by a factor of 7. MCPI is compared with classical (*i.e.,* non mixed critical) techniques, which apply only a simple naive heuristic to bias critical tasks to higher priority, since we are not aware of comparable algorithm. The number of schedulable instances is improved up to 30%. Finally we compare MCPI with Audsley approach based solution. MCPI gives comparable results to "perfect" Audsley approach implementation, *i.e.,* implementations that use perfect estimation of jobs termination time. However this implementation are applicable only to small instances, having exponential computational complexity. We also showed that MCPI outperforms an implementation of Audsley approach with reasonable computational complexity.

In Chapter 4 we addressed the problem of static scheduling. As said before, this is motivated by the fact that static policies allow for better interference analysis. The idea is to generate a schedule that consists of two time triggered tables, one per criticality level. We call this approach *Single Time Table per Mode* (STTM). This problem is not trivial, since, in general, a number of tables that grows linearly with the number of HI-critical jobs is needed [BLS10]. Our proposed algorithm can generate STTM tables starting from a non-static scheduling policy (for example a priority based one). We formally proved that our algorithm is optimal for single processor case, in the sense that we successfully find a feasible scheduling table if the original policy generates a feasible schedule as well. The latter also implies an interesting theoretical result: STTM scheduling strategy is optimal for single processor. At the end of the chapter we provided experimental results that for the

multiprocessor case the algorithm fails only under very high workload.

Finally in Chapter 5 we propose a design flow for mixed critical systems. It is based on a novel Model of Computation (MoC), *Fixed Priority Process Network* (FPPN), that allows to describe both reactive-control and data stream processing applications and has the advantage of generating deterministic executions. The proposed methodology can be, however, extended to other MoCs. In our flow both the MoC and the scheduling policy are described in the timed-automata based language BIP (Behaviour Interaction Priorities). We generate the BIP code automatically from high level description in DOL-Critical language. The effectiveness of the approach is shown using a real-life example from avionic industry, a *Flight Management System* on a real hardware platform – MPPA® of Kalray.

6.2 Future work

Regarding the scheduling policies presented in Chapter 3, we plan to complete the work done so far by extending MCEDF to precedence constraints and to prove, if possible, equivalence with MCPI even in this extended case. For MCPI we would like to find a more precise technique to estimate the potential interference relation. The single processor simulation proposed in this thesis is far from being minimal, and we are confident that a more accurate estimation should further increase the performances of MCPI. Also, an alternative way of handling Dhall effect beyond density based separation fixed priority would be desirable, since this technique is not very effective for the finite set of job models.

We would also like to extend the results of Chapter 3 and 4 to non-preemptive case and to multiple levels of criticality.

For the design flow presented in Chapter 5 we are interested in implementing our own scheduling policies, instead of third-party ones, such as those described in Chapter 3 and 4 to test them on real case-studies. Also we are planning to extend the design flow in order to test it with different MoCs, and using it on other multicore computational platforms. Finally we are planning to implement other case-studies.

List of Figures

List of Tables

Contributions

[1] Georgia Giannopoulou, Peter Poplavko, Dario Socci, Pengcheng Huang, Nikolay Stoimenov, Paraskevas Bourgos, Lothar Thiele, Marius Bozga, Saddek Bensalem, Sylvain Girbal, Madeleine Faugere, Romain Soulat, and Benoît Dupont de Dinechin. Dol-bip-critical: A tool chain for the design and correct-by-construction implementation of mixed-criticality multi-core systems. Submitted to *Design Automation for Embedded Systems*.

[2] Peter Poplavko, Dario Socci, Paraskevas Bourgos, Saddek Bensalem, and Marius Bozga. Models for deterministic execution of real-time multiprocessor applications. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1665–1670. EDA Consortium, 2015.

[3] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 93–102. IEEE, 2013.

[4] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Time-triggered mixed-critical scheduler. *Proc. WMC, RTSS*, pages 67–72, 2013.

[5] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Multiprocessor scheduling of precedence-constrained mixed-critical jobs. Technical report, 2015.

[6] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Time-triggered mixed-critical scheduler on single-and multi-processor platforms. 2015.

[7] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. A timed-automata based middleware for time-critical multicore applications. In *ISORCW 2015*, 2015.

[8] Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga, et al. Modeling mixed-critical systems in real-time bip. In *1st workshop on Real-Time Mixed Criticality Systems*, 2013.

Bibliography

[ACS10] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10. ACM, 2010.

[ACW05] Peter Amey, Rod Chapman, and Neil White. Smart certification of mixed criticality systems. In *Reliable Software Technology–Ada-Europe 2005*, pages 144–155. Springer, 2005.

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times a tool for modelling and implementation of embedded systems. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464. Springer, 2002.

[AGSCG11] Bader Alahmad, Sathish Gopalakrishnan, Luca Santinelli, and Liliana Cucu-Grosjean. Probabilities for mixed-criticality problems: Bridging the uncertainty gap. *RTSS 2011 Organization Committee*, page 1, 2011.

[Aud93] N.C. Audsley. *Flexible Scheduling in Hard-Real-Time Systems*. PhD thesis, Dept. of Computer Science, Univ. of York, 1993.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Bar04] Sanjoy K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Trans. Comput.*, 53(6):781–784, June 2004.

[Bar09] Sanjoy Baruah. Mixed criticality schedulability analysis is highly intractable, 2009.

[Bar12] Sanjoy Baruah. Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In *RTNS'12*, pages 11–19. ACM, 2012.

[Bar13] Sanjoy K Baruah. Implementing mixed criticality synchronous reactive systems upon multiprocessor platforms. *The University of North Carolina at Chapel Hill, Tech. Rep*, 2013.

[BBB+09] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russell Urzi. A research agenda for mixed-criticality systems. *Cyber-Physical Systems Week*, 2009.

[BBD11] S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34–43, Nov 2011.

[BBD+12a] S Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Euromicro Conf. on Real-Time Systems*, ECRTS'12, pages 145–154. IEEE, 2012.

[BBD+12b] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Trans. Comput.*, 61(8):1140 –1152, aug. 2012.

[BC13] Sanjoy Baruah and Bipasa Chattopadhyay. Response-time analysis of mixed criticality systems with pessimistic frequency specification. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 237–246. IEEE, 2013.

[BCLS14] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[BD13] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

[BD+14] A Burns, Robert Davis, et al. Adaptive mixed criticality scheduling with deferred preemption. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 21–30. IEEE, 2014.

[BF05] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 9 pp.–329, Dec 2005.

[BF11] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 3–12, 2011.

[BLS10] Sanjoy K. Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium*, RTAS'10, pages 13–22. IEEE, 2010.

[Bra14] B.B. Brandenburg. A synchronous ipc protocol for predictable access to shared resources in mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 196–206, Dec 2014.

[Bur13] A. Burns. The application of the original priority ceiling protocol to mixed criticality systems. In L. George and G. Lipari, editors, *ReTiMiCS, RTCSA*, pages 7–11, 2013.

[BV08] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 147–155, July 2008.

[BZ13] Sanjoy Baruah and Guo Zhishan. Mixed criticality scheduling upon unreliable processors, 2013.

[CBF+11] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard, and Claire Pagetti. Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset. In *19th International Conference on Real-Time and Network Systems*, 2011.

[Col97] Robert Collins. Inside the pentium II math bug - Dan-0411 rocks the industry. *Dr.Dobb's Journal*, 22(8):52, August 1997.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), October 2011.

[DFG+14] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *ERTSS'14*, 2014.

[DL78] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[dNP14] D. de Niz and L.T.X. Phan. Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 111–122, April 2014.

[DRBL74] Robert H Dennard, VL Rideout, E Bassous, and AR Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[Eur11] European Aviation Safety Agency. Easa cm-swceh-001 development assurance of airborne electronic hardware. August 2011. `http://easa.europa.eu/system/files/dfu/certification-memoranda-import-EASA%20CM-SWCEH-001%20Issue%2001%20Rev%2001%20Development%20Assurance%20of%20Airborne%20Electronic%20Hardware.pdf`.

[EY12] Pontus Ekberg and Wang Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Euromicro Conf. on Real-Time Systems*, ECRTS'12, pages 145–154. IEEE, 2012.

[F+10] Julien Forget et al. Scheduling dependent periodic tasks without synchronization mechanisms. In *RTAS'10*, pages 301–310, 2010.

[Fed15] Federal Certification Authorities Software Team (CAST). Cast-32, multicore processors. May 2015. `https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf`.

[FKP07] Elena Fersman, Pavel Krcl, Paul Pettersson, and Wang Yi 0001. Task automata: Schedulability, decidability and undecidability. *Inf. Comput.*, 205(8):1149–1172, 2007.

[FKRvH06] Hauke Fuhrmann, Jens Koch, Jörn Rennhack, and Reinhard von Hanxleden. Model-based system design of time-triggered architectures—an avionics case study. In *25th Digital Avionics Systems Conference (DASC'06)*, Portland, OR, USA, October 2006.

[Fle13] Thomas Fleming. *Extending mixed criticality scheduling*. PhD thesis, University of York, 2013.

[GB13] Patrick Graydon and Iain Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *:*, pages 19–24, 2013.

[GESY11] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium*, RTSS'11, pages 13–23. IEEE, 2011.

[GGDY13] Chuancai Gu, Nan Guan, Qingxu Deng, and Wang Yi. Improving ocbp-based scheduling for mixed-criticality sporadic task systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 247–256, Aug 2013.

[GPS+] Georgia Giannopoulou, Peter Poplavko, Dario Socci, Pengcheng Huang, Nikolay Stoimenov, Paraskevas Bourgos, Lothar Thiele, Marius Bozga, Saddek Bensalem, Sylvain Girbal, Madeleine Faugere, Romain Soulat, and Benoît Dupont de Dinechin. Dol-bip-critical: A tool chain for the design and correct-by-construction implementation of mixed-criticality multi-core systems. Submitted to *Design Automation for Embedded Systems*.

[GRP14] Romain GRATIA, Thomas ROBERT, and Laurent PAUTET. Adaptation of run to mixed-criticality systems. *JRWRTC 2014*, page 25, 2014.

[GSHT] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. DOL-C: Distributed operation layer for mixed-criticality applications, http://www.tik.ee.ethz.ch/ certainty/dolc.html.

[GSHT13] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15. IEEE, 2013.

[GSY15] Zhishan Guo, Luca Santinelli, and Kecheng Yang. Edf schedulability analysis on mixed-criticality systems with permitted failure probability. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2015 IEEE 21st International Conference on*, pages 187–196. IEEE, 2015.

[HGBH09] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2, 2009.

[HGL14] Huang-Ming Huang, Christopher Gill, and Chenyang Lu. Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):126, 2014.

[HKM+12] Jonathan L Herman, Christopher J Kenna, Malcolm S Mollison, James H Anderson, and Daniel M Johnson. Rtos support for multicore mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 197–208. IEEE, 2012.

[HL94] Rhan Ha and J. W S Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. Int. Conf. Distributed Computing Systems*, pages 162–171, Jun 1994.

[JZLP14] Mathieu Jan, Lilia Zaourar, Vincent Legout, and Laurent Pautet. Handling criticality mode change in time-triggered systems through linear programming. In *Ada User Journal, Proc of Workshop on Mixed Criticality for Industrial Systems (WMCIS2014)*, volume 35, pages 138–143, 2014.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, dec 1999.

[KAZ11] Owen R Kelly, Hakan Aydin, and Baoxian Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1051–1059. IEEE, 2011.

[LB10a] Haohan Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 183–192, Nov 2010.

[LB10b] Haohan Li and Sanjoy Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Intern. Conf. on Embedded Software*, EMSOFT '10, pages 99–108. ACM, 2010.

[LB12] Haohan Li and Sanjoy K. Baruah. Outstanding paper award: Global mixed-criticality scheduling on multiprocessors. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012*, 2012.

[LDNRM10] Karthik Lakshmanan, Dionisio De Niz, Ragunathan Rajkumar, and Gines Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 169–178. IEEE, 2010.

[Lee05] Edward A Lee. Absolutely positively on time: what would it take?[embedded computing systems]. *Computer*, 38(7):85–87, 2005.

[LEL+14] Per Lindgren, Johan Eriksson, Marcus Lindner, David J Pereira, and Luís Miguel Pinho. Rtfm-lang static semantics for systems with mixed criticality. *Ada User Journal*, 2014.

[Liu00] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, Inc., 2000.

[LMPC04] R. Le Moigne, O. Pasquier, and J-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *Proceedings of the Conference on*

Design, Automation and Test in Europe - Volume 3, DATE '04, pages 30082–, Washington, DC, USA, 2004. IEEE Computer Society.

[MEA+10] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Int. Conf. Computer and Information Technology*, CIT '10, pages 1864–1871. IEEE, 2010.

[Moo65] Gordon Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), april 1965.

[NLR09] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium*, RTSS'09, pages 291–300. IEEE, 2009.

[Pat12] Risat Mahmud Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 309–320. IEEE, 2012.

[PBS+] Peter Poplavko, Paraskevas Bourgos, Dario Socci, Saddek Bensalem, and Marius Bozga. Multicore code generation for time-critical applications, http://www-verimag.imag.fr/multicore-time-critical-code,470.html.

[PK11] Taeju Park and Soontae Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Intern. Conf. on Embedded software*, EMSOFT '11, pages 253–262. ACM, 2011.

[PSPBB15] Peter Poplavko, Dario Socci, Saddek Paraskevas Bourgos, and Marius Bozga Bensalem. Models for deterministic execution of real-time multiprocessor applications. In *DATE'15*, 2015.

[Sch09] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009:2, 2009.

[SRL90] Lui Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, Sep 1990.

[TCBS13] A. Triki, J. Combaz, S. Bensalem, and J. Sifakis. Model-based implementation of parallel real-time systems. In *FASE'13*. Springer, 2013.

[TFB13] Jens Theis, Gerhard Fohler, and Sanjoy Baruah. Schedule table generation for time-triggered mixed criticality systems. *Proc. WMC, RTSS*, pages 79–84, 2013.

[Ves07] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium*, RTSS'07, pages 239–243. IEEE, 2007.

[YKRB14] Eugene Yip, Matthew Kuo, Partha S Roop, and David Broman. Relaxing the Synchronous Approach for Mixed-Criticality Systems. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[ZGZ13] Qingling Zhao, Zonghua Gu, and Haibo Zeng. Integration of resource synchronization and preemption-thresholds into edf-based mixed-criticality scheduling algorithm. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 227–236, Aug 2013.

Appendix A

List Scheduling

For a given task graph $\mathbf{T}$, 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[1], 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 A.1. For the set jobs we specify an array of arrival times $A[*]$, deadlines $D[*]$, 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 A.2 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 [CSRL01] 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).

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

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, [$\mathbf{J}_{\text{EFF}}$, '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', '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, 'LBL-TERM'], $terminationTime$)
44: **end if**
45: $lastTime \leftarrow time$
46: **end while**

Figure A.1: 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 A.2: Primitive Schedule Operations

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 A.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.

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 A.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[2]. There are $O(k)$ number of events, and hence $O(k)$ main-loop iterations of the list scheduler. Except for visiting 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. □

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

Appendix B

Transformed Fixed Priority Simulation

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 (see Appendix A) 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 4.2.3. 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 presence of non-preemption and/or multiple processors 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, *i.e.,* the HI criticality graph, modified with ASAP and ALAP time, as defined in Section 2.2.3

The pseudo-code of the transformed list scheduling algorithm is given in Fig. B.1.

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 (4.1a, 4.1b, 4.1c) the algorithm keeps track of

Algorithm: *SimulateTranformedListSchedule*
Input: **Boolean** *preemAllowed* **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** $prgs$
Local: **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, [$\mathbf{J}_{\text{HI}}$, 'LBL-ARR'], $A[*]$)
4: *PQueuePushScheduleEvents*(Q_E, S_{LO}, $\mathbf{J}_{\text{HI}}$, 'LBL-LO-RUN', '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_{HI}, $jstat[*][HI]$, $pstat[*][HI]$)
13: $SetAdd(\mathbf{J}^{dis}, J)$
14: **case** 'LBL-LO-STOP'
15: *SchedStop*(J, $time$, S_{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_{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: $ScheduleHighestPrioriorityJob(S_{\text{HI}}, Q_P, preemAllowed, pstat[*][HI], jstat[*][HI], prgs[*][HI])$
28: **define** $lag(J) = (prgs[J][LO] - prgs[J][HI])$
29: $(minLag, J) = \min(lag(J) \mid J \in Running(pstat[*][HI]) \setminus \mathbf{J}^{LO\text{-}term} \wedge jstat[J][LO].proc = \emptyset)$
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 B.1: The Transformed List Scheduling for Generating **HI*** Table, '$\mathcal{T}$(LS-SC)'

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, 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 (4.1b). 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 (4.1a). 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 (4.1a) and (4.1c) 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. Note also that by construction a job may be disabled in the middle of execution only if preemption is allowed. Thus, the algorithm does not interrupt a running job if preemption is not allowed.

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. B.1 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 B.1 (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.

Lemma B.2 (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 A.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. □